Software Testing
Body of Knowledge
for CSTE
# Table of Contents

## Introduction to the Software Testing Certification Program

- Intro.1. Software Certification Overview
  - Intro.1.1. Contact Us
  - Intro.1.2. Program History
  - Intro.1.3. Why Become Certified?
  - Intro.1.4. Benefits of Becoming Certified
- Intro.2. Meeting the Certification Qualifications
  - Intro.2.1. Prerequisites for Candidacy
  - Intro.2.2. Code of Ethics
  - Intro.2.3. Submitting the Initial Application
  - Intro.2.4. Application-Examination Eligibility Requirements
- Intro.3. Scheduling with Pearson VUE to Take the Examination
  - Intro.3.1. Arriving at the Examination Site
- Intro.4. How to Maintain Competency and Improve Value
  - Intro.4.1. Continuing Professional Education
  - Intro.4.2. Advanced Software Testing Designations

## Skill Category 1

### Software Testing Principles and Concepts

- 1.1. Vocabulary
- 1.2. Quality Basics
  - 1.2.1. Quality Assurance Versus Quality Control
  - 1.2.2. Quality, A Closer Look
  - 1.2.3. What is Quality Software?
  - 1.2.4. The Cost of Quality
  - 1.2.5. Software Quality Factors
- 1.3. Understanding Defects
  - 1.3.1. Software Process Defects
  - 1.3.2. Software Process and Product Defects
- 1.4. Process and Testing Published Standards
1.4.1. CMMI® for Development ............................................................... 1-27
1.4.2. TMMi—Test Maturity Model integration ...................................... 1-30
1.4.3. ISO/IEC/IEEE 29119 Software Testing Standard ............................ 1-32
1.5. Software Testing ............................................................................ 1-34
  1.5.1. Principles of Software Testing...................................................... 1-34
  1.5.2. Why Do We Test Software? ......................................................... 1-36
  1.5.3. Developers are not Good Testers ............................................... 1-36
  1.5.4. Factors Affecting Software Testing .............................................. 1-37
  1.5.5. Independent Testing ................................................................. 1-43
  1.6.1. Typical Tasks in the Development Life Cycle ............................. 1-45
  1.6.2. Model Variations ........................................................................ 1-46
  1.6.3. Ad-hoc Development ............................................................... 1-46
  1.6.4. The Waterfall Model ............................................................... 1-47
  1.6.5. V–Model ................................................................................... 1-50
  1.6.6. Incremental Model ................................................................. 1-51
  1.6.7. Iterative Development .............................................................. 1-51
  1.6.8. Variations on Iterative Development ......................................... 1-53
  1.6.9. The Spiral Model ..................................................................... 1-55
  1.6.10. The Reuse Model .................................................................... 1-57
  1.6.11. Creating and Combining Models ............................................. 1-58
  1.6.12. SDLC Models Summary .......................................................... 1-59
  1.6.13. Application Lifecycle Management (ALM) ............................... 1-59
1.7. Agile Development Methodologies ........................................ 1-59
  1.7.1. Basic Agile Concepts ............................................................... 1-60
  1.7.2. Agile Practices ......................................................................... 1-61
  1.7.3. Effective Application of Agile Approaches .................................. 1-62
  1.7.4. Integrating Agile with Traditional Methodologies ..................... 1-63
1.8. Testing Throughout the Software Development Life Cycle (SDLC) ................................................................. 1-63
  1.8.1. Static versus Dynamic Testing .................................................. 1-64
  1.8.2. Verification versus Validation .................................................... 1-65
  1.8.3. Traceability Matrix ................................................................... 1-67
1.9. Testing Schools of Thought and Testing Approaches ... 1-69
  1.9.1. Software Testing Schools of Thought ......................................... 1-69
  1.9.2. Testing Approaches ................................................................. 1-70
1.10. Test Categories and Testing Techniques ............................. 1-72
  1.10.1. Structural Testing ................................................................. 1-72
  1.10.2. Functional Testing ............................................................... 1-75
  1.10.3. Non-Functional Testing .......................................................... 1-77
  1.10.4. Incremental Testing ............................................................... 1-78
  1.10.5. Thread Testing ........................................................................ 1-79
  1.10.6. Regression Testing ............................................................... 1-80
  1.10.7. Testing Specialized Technologies .............................................. 1-81
Skill Category 2

Building the Software Testing Ecosystem . . . . . . 2-1

2.1. Management’s Role . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-2
  2.1.1. Setting the Tone . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-2
  2.1.2. Commitment to Competence . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-3
  2.1.3. The Organizational Structure Within the Ecosystem . . . . . . . . . . . . 2-4
  2.1.4. Meeting the Challenge . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-4

2.2. Work Processes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-5
  2.2.1. What is a Process? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-5
  2.2.2. Components of a Process . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-6
  2.2.3. Tester’s Workbench . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-6
  2.2.4. Responsibility for Building Work Processes . . . . . . . . . . . . . . . . . . . 2-8
  2.2.5. Continuous Improvement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-10
  2.2.6. Analysis and Improvement of the Test Process . . . . . . . . . . . . . . . 2-12
  2.2.7. Test Process Alignment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-23
  2.2.8. SDLC Methodologies Impact on the Test Process . . . . . . . . . . . . 2-24
  2.2.9. The Importance of Work Processes . . . . . . . . . . . . . . . . . . . . . . . . . 2-25

2.3. Test Environment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-26
  2.3.1. What is the Test Environment? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-26
  2.3.2. Why do We Need a Test Environment? . . . . . . . . . . . . . . . . . . . . . . 2-26
  2.3.3. Establishing the Test Environment . . . . . . . . . . . . . . . . . . . . . . . . . . 2-27
  2.3.4. Virtualization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-30
  2.3.5. Control of the Test Environment . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-31

2.4. Test Tools . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-31
  2.4.1. Categories of Test Tools . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-31
  2.4.2. Advantages and Disadvantages of Test Automation . . . . . . . . . . . 2-35
  2.4.3. What Should Be Automated . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-36
  2.4.4. Tool Acquisition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-37
  2.4.5. Last Thoughts on Tool Acquisition . . . . . . . . . . . . . . . . . . . . . . . . . 2-47

2.5. Skilled Team . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-48
  2.5.1. Types of Skills . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-48
  2.5.2. Business Domain Knowledge . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-49
  2.5.3. Test Competency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-50

Skill Category 3

Managing the Test Project . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-1

3.1. Test Administration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-1
  3.1.1. Test Planning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-2
  3.1.2. Estimation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-2
  3.1.3. Developing a Budget for Testing . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-14
3.1.4. Scheduling ................................................................. 3-14
3.1.5. Resourcing ............................................................... 3-15
3.1.6. Customization of the Test Process .............................. 3-18
3.2. Test Supervision ......................................................... 3-19
3.2.1. Communication Skills .............................................. 3-19
3.2.2. Negotiation and Complaint Resolution Skills .............. 3-26
3.2.3. Project Relationships .............................................. 3-31
3.2.4. Motivation, Mentoring, and Recognition .................... 3-33
3.3. Test Leadership ......................................................... 3-34
3.3.1. Chairing Meetings .................................................. 3-34
3.3.2. Team Building ....................................................... 3-35
3.3.3. Quality Management Organizational Structure ............ 3-38
3.3.4. Code of Ethics ....................................................... 3-40
3.4. Managing Change ....................................................... 3-41
3.4.1. Software Configuration Management ......................... 3-41
3.4.2. Software Change Management .................................... 3-42
3.4.3. Software Version Control ......................................... 3-43

Skill Category 4
Risk in the Software Development Life Cycle ................. 4-1
4.1. Risk Concepts and Vocabulary ................................. 4-2
4.1.1. Risk Categories .................................................... 4-2
4.1.2. Risk Vocabulary .................................................... 4-3
4.2. Risk Management ..................................................... 4-5
4.2.1. The Risk Management Process ................................. 4-5
4.3. Risk Assessment ....................................................... 4-7
4.3.1. Risk Assessment - Risk Identification Process ......... 4-8
4.3.2. Risk Assessment - Risk Analysis Process ............... 4-8
4.3.3. The Risk Assessment Method ................................. 4-9
4.4. Risk Control ............................................................ 4-9
4.4.1. Risk Control - Planning ......................................... 4-10
4.4.2. Risk Control - Risk Resolution Strategies ................ 4-11
4.4.3. Risk Control - Monitoring & Tracking .................... 4-11
4.5. Risk Management Plan .............................................. 4-11
4.6. Risks Associated with Software Development ........... 4-12
4.6.1. Improper Use of Technology ................................. 4-12
4.6.2. Repetition of Errors .............................................. 4-13
4.6.3. Cascading of Errors .............................................. 4-13
4.6.4. Inability to Translate User Needs into Business and Technical Requirements .......................... 4-13
4.6.5. Incorrect Entry of Data .......................................... 4-13
# Table of Contents

- 4.6.6. Concentration of Data ................................................. 4-13
- 4.6.7. Inability to React Quickly ......................................... 4-14
- 4.6.8. Inability to Substantiate Processing ................................ 4-14
- 4.6.9. Concentration of Responsibilities .................................. 4-14
- 4.6.10. Erroneous or Falsified Input Data ................................. 4-15
- 4.6.11. Misuse by Authorized End Users .................................. 4-15
- 4.6.12. Uncontrolled System Access ....................................... 4-15
- 4.6.13. Ineffective Security and Privacy Practices for the Application .. 4-15
- 4.6.15. Program Errors ...................................................... 4-16
- 4.6.16. Operating System Flaws ........................................... 4-16
- 4.6.17. Communications System Failure .................................... 4-16
- 4.7. Premature Deployment Risk ............................................ 4-16
- 4.8. Risks Associated with Software Testing ............................. 4-17
  - 4.8.1. Software Testing Risks ............................................... 4-17
  - 4.8.2. Customize Processes based on Risk ................................ 4-20
- 4.9. Risk Scoring Methods .................................................. 4-20
  - 4.9.1. Risk Score ............................................................ 4-20
  - 4.9.2. Two Risk Levels ..................................................... 4-21
  - 4.9.3. Application Level Scoring Method ................................ 4-21
  - 4.9.4. Component Level Scoring Method .................................. 4-24

## Skill Category 5

### Test Planning ............................................................. 5-1

- 5.1. The Test Plan ......................................................... 5-1
  - 5.1.1. Advantages to Utilizing a Test Plan .............................. 5-2
  - 5.1.2. The Test Plan as a Contract and a Roadmap ..................... 5-2
- 5.2. Prerequisites to Test Planning ...................................... 5-2
  - 5.2.1. Objectives of Testing .............................................. 5-3
  - 5.2.2. Acceptance Criteria ................................................ 5-3
  - 5.2.3. Assumptions .......................................................... 5-3
  - 5.2.4. Team Issues .......................................................... 5-3
  - 5.2.5. Constraints ........................................................... 5-4
  - 5.2.6. Understanding the Characteristics of the Application .......... 5-4
- 5.3. Hierarchy of Test Plans .............................................. 5-5
- 5.4. Create the Test Plan ................................................ 5-7
  - 5.4.1. Build the Test Plan ................................................ 5-8
  - 5.4.2. Write the Test Plan ............................................... 5-11
  - 5.4.3. Changes to the Test Plan ......................................... 5-11
  - 5.4.4. Attachments to the Test Plan .................................... 5-18
- 5.5. Executing the Plan ................................................... 5-19
Skill Category 6
Walkthroughs, Checkpoint Reviews, and Inspections 6-1

6.1. Purpose of Reviews 6-2
6.1.1. Emphasize Quality throughout the SDLC 6-2
6.1.2. Detect Defects When and Where they are Introduced 6-2
6.1.3. Opportunity to Involve the End User/Customer 6-3
6.1.4. Permit "Midcourse" Corrections 6-3

6.2. Review Types 6-4
6.2.1. Desk Checks 6-4
6.2.2. Walkthroughs 6-4
6.2.3. Checkpoint Reviews 6-4
6.2.4. Inspections 6-5

6.3. Prerequisites to Reviews 6-6
6.3.1. A System Development Methodology 6-6
6.3.2. Management Support 6-6
6.3.3. Review Process 6-7
6.3.4. Project Team Support 6-7
6.3.5. Training 6-7

6.4. Conducting Walkthroughs 6-7
6.4.1. Select the Walkthrough Team 6-8
6.4.2. Conduct the Walkthrough Meeting 6-8
6.4.3. Walkthrough Meeting Follow-Up 6-8

6.5. Checkpoint Reviews 6-9
6.5.1. Phase I – Planning 6-9
6.5.2. Phase II – Checkpoints 6-9
6.5.3. Sharing Data with Quality Assurance 6-10

6.6. Conducting Inspections 6-10
6.6.1. What are Inspections? 6-10
6.6.2. The Background on Inspections 6-10
6.6.3. Inspection Process 6-10

6.7. Summary 6-13

Skill Category 7
Designing Test Cases 7-1

7.1. Identifying Test Conditions 7-1
7.1.1. Defining Test Conditions from Specifications 7-2
7.1.2. Defining Test Conditions from the Production Environment 7-3
7.1.3. Defining Test Conditions from Test Transaction Types 7-4
7.1.4. Defining Test Conditions from Business Case Analysis .......... 7-21
7.1.5. Defining Test Conditions from Structural Analysis ............. 7-22

**7.2. Test Conditions from Use Cases** .......................... 7-22
  7.2.1. What is a Use Case ........................................ 7-22
  7.2.2. How Use Cases are Created ................................ 7-26
  7.2.3. Use Case Format ........................................... 7-28
  7.2.4. How Use Cases are Applied ............................... 7-30
  7.2.5. Develop Test Cases from Use Cases ...................... 7-30

**7.3. Test Conditions from User Stories** .......................... 7-31
  7.3.1. INVEST in User Stories ..................................... 7-32
  7.3.2. Acceptance Criteria and User Stories ..................... 7-33
  7.3.3. Acceptance Criteria, Acceptance Tests, and User Stories .... 7-33
  7.3.4. User Stories Provide a Perspective ....................... 7-33
  7.3.5. Create Test Cases from User Stories ..................... 7-33

**7.4. Test Design Techniques** ...................................... 7-34
  7.4.1. Structural Test Design Techniques .......................... 7-35
  7.4.2. Functional Test Design Techniques .......................... 7-41
  7.4.3. Experience-based Techniques ............................... 7-60
  7.4.4. Non-Functional Tests ...................................... 7-60

**7.5. Building Test Cases** ........................................... 7-62
  7.5.1. Process for Building Test Cases ............................ 7-63
  7.5.2. Documenting the Test Cases ................................ 7-63

**7.6. Test Coverage** ................................................ 7-65

**7.7. Preparing for Test Execution** ............................... 7-66

**Skill Category 8**

**Executing the Test Process** ....................................... 8-1

  **8.1. Acceptance Testing** ........................................ 8-2
    8.1.1. Incremental Acceptance ................................ 8-2
    8.1.2. Incremental Acceptance Benefits ....................... 8-2
    8.1.3. Fitness for Use .......................................... 8-3
    8.1.4. Acceptance Criteria ..................................... 8-4
    8.1.5. Process for Acceptance Testing ....................... 8-5
    8.1.6. Acceptance Testing Summary ........................... 8-7

  **8.2. IEEE Test Procedure Specification** ........................ 8-7

  **8.3. Test Execution** ............................................. 8-8
    8.3.1. Test Environment .......................................... 8-9
    8.3.2. Test Cycle Strategy ...................................... 8-9
    8.3.3. Test Data .................................................. 8-9
    8.3.4. Use of Tools in Testing .................................. 8-16
    8.3.5. Test Documentation ...................................... 8-16
8.3.6. Perform Tests .................................................. 8-16
8.3.7. Unit Testing ................................................... 8-20
8.3.8. Integration Testing ............................................ 8-22
8.3.9. System Testing .................................................. 8-24
8.3.10. User Acceptance Testing (UAT) ............................... 8-24
8.3.11. Testing COTS Software ....................................... 8-26
8.3.12. Acceptance Test the COTS Software ......................... 8-31
8.3.13. When is Testing Complete? .................................... 8-32

8.4. Testing Controls ................................................... 8-32
  8.4.1. Environmental versus Transaction Processing Controls .......... 8-32
  8.4.2. Environmental or General Controls ............................ 8-32
  8.4.3. Transaction Processing Controls ................................ 8-33

8.5. Recording Test Results ............................................ 8-39
  8.5.1. Deviation from what should be ................................ 8-40
  8.5.2. Effect of a Defect ............................................ 8-42
  8.5.3. Defect Cause ................................................ 8-42
  8.5.4. Use of Test Results ........................................... 8-43

8.6. Defect Management ................................................ 8-43
  8.6.1. Defect Naming ............................................... 8-44
  8.6.2. The Defect Management Process ............................... 8-44

Skill Category 9
Measurement, Test Status, and Reporting ............... 9-1

9.1. Prerequisites to Test Reporting ................................. 9-1
  9.1.1. Define and Collect Test Status Data ........................... 9-2
  9.1.2. Define Test Measures and Metrics used in Reporting ........ 9-3
  9.1.3. Define Effective Test Measures and Metrics ................... 9-5

9.2. Analytical Tools used to Build Test Reports ............... 9-10
  9.2.1. Histograms ................................................ 9-10
  9.2.2. Pareto Charts .............................................. 9-12
  9.2.3. Cause and Effect Diagrams .................................. 9-15
  9.2.4. Check Sheets .............................................. 9-18
  9.2.5. Run Charts ................................................ 9-20
  9.2.6. Control Charts ............................................. 9-22

9.3. Reporting Test Results .......................................... 9-24
  9.3.1. Current Status Test Reports .................................. 9-25
  9.3.2. Final Test Reports .......................................... 9-41
  9.3.3. Guidelines for Report Writing ................................ 9-44
# Skill Category 10

**Testing Specialized Technologies** .......................... 10-1

10.1. **New Technology** ................................................. 10-2
   10.1.2. Testing the Effectiveness of Integrating New Technology . 10-3
   10.1.3. Test the Adequacy of Staff Skills to Use the Technology 10-5

10.2. **Web-Based Applications** ................................. 10-7
   10.2.1. Understand the Basic Architecture .................. 10-7
   10.2.2. Test Related Concerns ................................. 10-9
   10.2.3. Planning for Web-based Application Testing ............ 10-9
   10.2.4. Identify Test Conditions .................................. 10-11
   10.2.5. Web Testing Tools .................................. 10-14

10.3. **Mobile Application Testing** .......................... 10-15
   10.3.1. Characteristics and Challenges of the Mobile Platform . 10-16
   10.3.2. Identifying Test Conditions on Mobile Apps .......... 10-17
   10.3.3. Mobile Test Labs .................................. 10-20
   10.3.4. Mobile App Test Automation .......................... 10-23

10.4. **Cloud Computing** ........................................ 10-23
   10.4.1. Defining the Cloud .................................. 10-23
   10.4.2. Testing in the Cloud .................................. 10-26
   10.4.3. Testing as a Service (TaaS) .......................... 10-28

10.5. **Testing in an Agile Environment** .................... 10-29
   10.5.1. Agile as an Iterative Process .......................... 10-29
   10.5.2. Testers Cannot Rely on having Complete Specifications . 10-29
   10.5.3. Agile Testers must be Flexible ...................... 10-30
   10.5.4. Key Concepts for Agile Testing ...................... 10-30
   10.5.5. Traditional vs. Agile Testing ...................... 10-30
   10.5.6. Agile Testing Success Factors ...................... 10-31

10.6. **DevOps** ...................................................... 10-32
   10.6.1. DevOps Continuous Integration ...................... 10-32
   10.6.2. DevOps Lifecycle .................................. 10-33
   10.6.3. Testers Role in DevOps .................................. 10-33
   10.6.4. DevOps and Test Automation .......................... 10-34

10.7. **The Internet of Things** ............................... 10-34
   10.7.1. What is a Thing? .................................. 10-34
   10.7.2. IPv6 as a Critical Piece .............................. 10-34
   10.7.3. Impact of IoT .................................. 10-35
   10.7.4. Testing and the Internet of Things .................. 10-35
Skill Category A
Vocabulary .................................................. A-1

Skill Category B
Test Plan Example .............................. B-1

Skill Category C
Test Transaction Types Checklists ............... C-1
  C.1. Field Transaction Types Checklist .................. C-2
  C.2. Records Testing Checklist ............................. C-4
  C.3. File Testing Checklist ................................ C-5
  C.4. Relationship Conditions Checklist .................. C-6
  C.5. Error Conditions Testing Checklist .................. C-8
  C.6. Use of Output Conditions Checklist ............... C-10
  C.7. Search Conditions Checklist .......................... C-11
  C.8. Merging/Matching Conditions Checklist ............. C-12
  C.9. Stress Conditions Checklist .......................... C-14
  C.10. Control Conditions Checklist ......................... C-15
  C.11. Attributes Conditions Checklist ...................... C-18
  C.12. Stress Conditions Checklist .......................... C-20
  C.13. Procedures Conditions Checklist ..................... C-21

Skill Category D
References ........................................................ D-1
Introduction to the Software Testing Certification Program

The Software Testing Certification program (CAST, CSTE, and CMST) was developed by leading software testing professionals as a means of recognizing software testers who demonstrate a predefined level of testing competency. The Software Testing Certification program is directed by the International Software Certification Board (ISCB), an independent Board and administered by the QAI Global Institute (QAI). The program was developed to provide value to the profession, the individual, the employer, and co-workers.

The CAST, CSTE, and CMST certifications test the level of competence in the principles and practices of testing and control in the Information Technology (IT) profession. These principles and practices are defined by the ISCB as the Software Testing Body of Knowledge (STBOK). The ISCB will periodically update the STBOK to reflect changing software testing and control, as well as changes in computer technology. These updates should occur approximately every three years.

<table>
<thead>
<tr>
<th>Software Certification Overview</th>
<th>Intro-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meeting the Certification Qualifications</td>
<td>Intro-7</td>
</tr>
<tr>
<td>Scheduling with Pearson VUE to Take the Examination</td>
<td>Intro-13</td>
</tr>
<tr>
<td>How to Maintain Competency and Improve Value</td>
<td>Intro-13</td>
</tr>
</tbody>
</table>

Be sure to check the Software Certifications Web site for up-to-date information on the Software Testing Certification program at:

www.softwarecertifications.org

Using this product does not constitute, nor imply, the successful passing of the certification examination.
Intro.1 Software Certification Overview

Software Certifications is recognized worldwide as the standard for IT testing professionals. Certification is a big step, a big decision. Certification identifies an individual as a test leader and earns the candidate the respect of colleagues and managers. It is formal acknowledgment that the IT recipient has an overall understanding of the disciplines and skills represented in a comprehensive Software Testing Common Body of Knowledge (STBOK) for a respective software discipline.

The Software Testing Certification programs demonstrate the following objectives to establish standards for initial qualification and continuing improvement of professional competence. The certification programs help to:

1. Define the tasks (skill categories) associated with software testing duties in order to evaluate skill mastery.
2. Demonstrate an individual’s willingness to improve professionally.
3. Acknowledge attainment of an acceptable standard of professional competency.
4. Aid organizations in selecting and promoting qualified individuals.
5. Motivate personnel having software testing responsibilities to maintain their professional competency.
6. Assist individuals in improving and enhancing their organization’s software testing programs (i.e., provide a mechanism to lead a professional).

In addition to Software Testing Certification, the ISCB also offer the following software certifications.

Software Testers
- Certified Associate in Software Testing (CAST)
- Certified Software Tester (CSTE)
- Certified Manager of Software Testing (CMST)

Software Quality Analysts
- Certified Associate in Software Quality (CASQ)
- Certified Software Quality Analyst (CSQA)
- Certified Manager of Software Quality (CMSQ)

Software Business Analysts
- Certified Associate in Business Analysis (CABA)
- Certified Software Business Analyst (CSBA)
Intro.1.1  Contact Us

Software Certifications  
Phone: (407)-472-8100  
Fax: (407)-363-1112

CSTE questions? E-mail: certify@softwarecertifications.org

Intro.1.2  Program History

QAI was established in 1980 as a professional association formed to represent the software testing industry. The first certification began development in 1985 and the first formal examination process was launched in 1990. Today, Software Certifications, administered by QAI, is global. Since its inception, the ISCB has certified over 50,000 IT professionals in 50+ countries worldwide.

Intro.1.3  Why Become Certified?

As the IT industry becomes more competitive, management must be able to distinguish professional and skilled individuals in the field when hiring. Certification demonstrates a level of understanding in carrying out software testing principles and practices that management can depend upon.

Acquiring a CAST, CSTE, or CMST certification indicates a foundation, professional practitioner, or managerial level of competence in software testing respectively. Software Testers become members of a recognized professional group and receive recognition for their competence by businesses and professional associates, potentially more rapid career advancement, and greater acceptance in the role as advisor to management.

Intro.1.4  Benefits of Becoming Certified

As stated above, the Software Testing certifications were developed to provide value to the profession, the individual, the employer, and co-workers. The following information is data collected from CSTEIs in the IT industry – a real testimonial to the benefits and reasons to make the effort to become a certified.
**Intro.1.4.1 Value Provided to the Profession**

Software testing is often viewed as a software project task, even though many individuals are full-time testing professionals. The Software Testing Certification program was designed to recognize software testing professionals by providing:

- **Software Testing Body of Knowledge (STBOK)**
  The ISCB defines the skills upon which the software testing certification is based. The current STBOK includes 10 skill categories fully described in this book – see Skill Category 1 through Skill Category 10.

- **Examination Process to Evaluate Competency**
  The successful candidate must pass an examination that is based on the STBOK. You must receive a grade of 70% or higher. The CAST examination consists of 100 multiple choice questions; the CSTE examination consists of 100 multiple choice and 12 short essays; and the CMST examination consists of 12 short essays.

- **Code of Ethics**
  The successful candidate must agree to abide by a professional Code of Ethics as specified by the ISCB. See “Code of Ethics” on page 9 for an explanation of the ethical behaviors expected of all certified professionals.

**Intro.1.4.2 Value Provided to the Individual**

The individual obtaining the CSTE certification receives the following values:

- **Recognition by Peers of Personal Desire to Improve**
  Approximately seventy-five percent (75%) of all CSTEs stated that a personal desire for self-improvement and peer recognition was the main reason for obtaining the CSTE certification. Fifteen percent (15%) were required by their employer to sit for the examination, and ten percent (10%) were preparing themselves for an improved testing-related position.
  Many CSTEs indicated that while their employer did not require CSTE certification, it was strongly encouraged.

- **Increased Confidence in Personal Capabilities**
  Eighty-five percent (85%) of the CSTEs stated that passing the examination increased their confidence to perform their job more effectively. Much of that confidence came from studying for the examination.

- **Recognition by IT Management for Professional Achievement**
  Most CSTEs stated that their management greatly respects those who put forth the personal effort needed for self-improvement. IT organizations recognized and rewarded individuals in the following ways:
• Thirteen percent (13%) received an immediate average one-time bonus of $610, with a range of $250 to $2,500.
• Twelve percent (12%) received an immediate average salary increase of 10%, with a range of 2% to 50%.

Non-monetary recognitions were:
• Thirty-six percent (36%) were recognized in staff meetings.
• Twenty percent (20%) in newsletters or e-mail.
• Many received rewards, management visits or calls, and lunch with the boss.

Within the first 18 months after receiving the CSTE certification:
• Twenty-seven percent (27%) received an average salary increase of 23%, with a range of 2% to 100%.
• Twenty-three percent (23%) were promoted, 25% received a better assignment and 13% a new assignment.

**Intro.1.4.3 Value Provided to the Employer**

With the need for increased software testing and reliability, companies employing certified testers provide value in these ways:

**Intro.1.4.3.1 Increased Confidence by IT Users and Customers**

IT users and customers expressed confidence in IT to effectively build or acquire software when certified testing practitioners were involved.

**Intro.1.4.3.2 Improved Processes to Build/Acquire/Maintain, Operate and Measure Software**

Certified Testers use their knowledge and skills to continuously improve the IT work processes. They know what to measure, how to measure it, and then prepare an analysis to aid in the decision-making process.

**Intro.1.4.3.3 Independent Assessment of Testing Competencies**

The Software Testing Certification program is directed by the ISCB. Through examination and recertification, they provide an independent assessment of one’s testing competencies, based on a continuously strengthening Software Testing Body of Knowledge.

**Intro.1.4.3.4 Testing Competencies Maintained Through Recertification**

Yesterday’s testing competencies are inadequate for today’s challenges. Recertification is a process that helps assure one’s skills remain current. The recertification process requires testers to obtain 120 hours of testing-related training per three year recertification cycle in topics specified by the ISCB.
From an IT director’s perspective, this is employee-initiated testing training. Most, if not all testers, do this training during their personal time. IT organizations gain three benefits from recertification: 1) employees initiate improvement; 2) testing practitioners obtain competencies in testing methods and techniques; and 3) employees train during personal time.

Intro.1.4.3.5 Value Provided to Co-Workers

The drive for self-improvement is a special trait that manifests itself in providing these values to co-workers:

Intro.1.4.3.6 Mentoring the Testing Staff

Forty-five percent (45%) of the CSTEs mentor their testing colleagues by conducting training classes, encouraging staff to become certified, and acting as a resource to the staff on sources of IT testing-related information.

Intro.1.4.3.7 Testing Resource to “IT” Staff

CSTEs and CMSTs are recognized as experts in testing and are used heavily for advice, counseling, and for recommendations on software construction and testing.

Intro.1.4.3.8 Role Model for Testing Practitioners

CSTEs and CMSTs are the IT role models for individuals with testing responsibilities to become more effective in performing their job responsibilities.

Intro.1.4.4 How to Improve Testing Effectiveness through Certification

A “driver” for improved IT effectiveness is the integration of the Software Testing certification program in your “IT” career development plan. This can be accomplished by:

- Creating an awareness of the Software Testing Certification and its benefits to your testing practitioners.
- Requiring or encouraging your testing practitioners to become certified.
- Recognizing and rewarding successful candidates.
- Supporting recertification as a means of maintaining testing competency.

-----

QAI, as administrators of the Software Testing Certification, will assist you in this effort.

See www.qaiglobalinstitute.com for detailed information.
Intro.2 Meeting the Certification Qualifications

To become certified in Software Testing, every candidate must first:

1. Satisfy all of the prerequisites required prior to applying for candidacy – educational and professional prerequisites, and recommendations for preparing for the examination.
2. Subscribe to the Code of Ethics as described on page Intro-9.
3. Complete the Certification Candidacy Online Application. See “Submitting the Initial Application” on page Intro-11 for information on all the materials needed to submit your application.

Intro.2.1 Prerequisites for Candidacy

Before you submit your application, first check that you satisfy the educational and professional prerequisites described below and understand what is expected of Certified Software Testers after certification.

Intro.2.1.1 Educational and Professional Prerequisites

Intro.2.1.1.1 CAST

To qualify for candidacy, each applicant must meet one of the “rule of 3’s” credentials listed below:

1. A three- or four-year degree from an accredited college-level institution.
2. A two-year degree from an accredited college-level institution and one year of experience in the information services field.
3. Three years of experience in the information services field.

Intro.2.1.1.2 CSTE

To qualify for candidacy, each applicant must meet one of the “rule of 6’s” credentials listed below:

1. A four year degree from an accredited college-level institution and two years of experience in the information services field.
2. A three year degree from an accredited college-level institution and three years of experience in the information services field.
3. A two year degree from an accredited college-level institution and four years of experience in the information services field.
4. Six years of experience in the information services field.
Intro.2.1.3 CMST

To qualify for candidacy, each applicant must meet one of the “rule of 8’s” credentials listed below:

1. A four-year degree from an accredited college-level institution and four years of experience in the information services field.
2. A two-year degree from an accredited college-level institution and six years of experience in the information services field.
3. Eight years of experience in the information services field.

Intro.2.1.2 Expectations of the Certified Professional

Knowledge within a profession doesn't stand still. Having passed the certification examination, a certificant has demonstrated knowledge of the designation's STBOK at the point in time of the examination. In order to stay current in the field, as knowledge and techniques mature, the certificant must be actively engaged in professional practice, and seek opportunities to stay aware of, and learn, emerging practices.

The certified tester is required to submit 120 credit hours of Continuing Professional Education (CPE) every three years to maintain certification or take an examination for recertification. Any special exceptions to the CPE requirements are to be directed to Software Certifications. Certified professionals are generally expected to:

- Attend professional conferences to stay aware of activities and trends in the profession.
- Take education and training courses to continually update skills and competencies.
- Develop and offer training to share knowledge and skills with other professionals and the public.
- Publish information in order to disseminate personal, project, and research experiences.
- Participate in the profession through active committee memberships and formal special interest groups.

The certified tester is expected not only to possess the skills required to pass the certification examination but also to be a change agent: someone who can change the culture and work habits of individuals (or someone who can act in an advisory position to upper management) to make quality in software testing happen.

Intro.2.1.2.1 Professional Skill Proficiency Responsibilities

In preparing yourself for the profession of IT software testing and to become more effective in your current job, you need to become aware of the three C’s of today's workplace:
• **Change** – The speed of change in technology and in the way work is performed is accelerating. Without continuous skill improvement, you will become obsolete in the marketplace.

• **Complexity** – Information technology is becoming more complex, not less complex. Thus, achieving quality, with regard to software testing in the information technology environment, will become more complex. You must update your skill proficiency in order to deal with this increased complexity.

• **Competition** – The ability to demonstrate mastery of multiple skills makes you a more desirable candidate for any professional position. While hard work does not guarantee your success, few, if any, achieve success without hard work. A software testing certification is one form of achievement. A software testing certification is proof that you’ve mastered a basic skill set recognized worldwide in the information technology arena.

---

### Intro.2.1.2.2 Develop a Lifetime Learning Habit

Become a lifelong learner in order to perform your current job effectively and remain marketable in an era of the three C’s. You cannot rely on your current knowledge to meet tomorrow’s job demands. The responsibility for success lies within your own control.

---

**Perhaps the most important single thing you can do to improve yourself professionally and personally is to develop a lifetime learning habit.**

**REMEMBER: “If it is going to be—it’s up to me.”**

---

### Intro.2.2 Code of Ethics

An applicant for certification must subscribe to the following Code of Ethics that outlines the ethical behaviors expected of all certified professionals. Software Certifications includes processes and procedures for monitoring certificant’s adherence to these policies. Failure to adhere to the requirements of the Code is grounds for decertification of the individual by the International Software Certifications Board.

#### Intro.2.2.1 Purpose

A distinguishing mark of a profession is acceptance by its members of responsibility to the interests of those it serves. Those certified must maintain high standards of conduct in order to effectively discharge their responsibility.
Intro.2.2.2 Responsibility

This Code of Ethics is applicable to all certified by the ISCB. Acceptance of any certification designation is a voluntary action. By acceptance, those certified assume an obligation of self-discipline beyond the requirements of laws and regulations.

The standards of conduct set forth in this Code of Ethics provide basic principles in the practice of information services testing. Those certified should realize that their individual judgment is required in the application of these principles.

Those certified shall use their respective designations with discretion and in a dignified manner, fully aware of what the designation denotes. The designation shall also be used in a manner consistent with all statutory requirements.

Those certified who are judged by the ISCB to be in violation of the standards of conduct of the Code of Ethics shall be subject to forfeiture of their designation.

Intro.2.2.3 Professional Code of Conduct

Software Certifications certificate holders shall:

1. Exercise honesty, objectivity, and diligence in the performance of their duties and responsibilities.

2. Exhibit loyalty in all matters pertaining to the affairs of their organization or to whomever they may be rendering a service. However, they shall not knowingly be party to any illegal or improper activity.

3. Not engage in acts or activities that are discreditable to the profession of information services testing or their organization.

4. Refrain from entering any activity that may be in conflict with the interest of their organization or would prejudice their ability to carry out objectively their duties and responsibilities.

5. Not accept anything of value from an employee, client, customer, supplier, or business associate of their organization that would impair, or be presumed to impair, their professional judgment and integrity.

6. Undertake only those services that they can reasonably expect to complete with professional competence.

7. Be prudent in the use of information acquired in the course of their duties. They shall not use confidential information for any personal gain nor in any manner that would be contrary to law or detrimental to the welfare of their organization.

8. Reveal all material facts known to them that, if not revealed, could either distort reports of operation under review or conceal unlawful practices.

9. Continually strive for improvement in their proficiency, and in the effectiveness and quality of their service.
10. In the practice of their profession, shall be ever mindful of their obligation to maintain the high standards of competence, morality, and dignity promulgated by this Code of Ethics.

11. Maintain and improve their professional competency through continuing education.

12. Cooperate in the development and interchange of knowledge for mutual professional benefit.


**Intro.2.2.4 Grounds for Decertification**

Revocation of a certification, or decertification, results from a certificant failing to reasonably adhere to the policies and procedures of Software Certifications as defined by the ISCB. The ISCB may revoke certification for the following reasons:

- Falsifying information on the initial application and/or a CPE reporting form,
- Failure to abide by and support the Software Certifications Code of Ethics,

**Intro.2.3 Submitting the Initial Application**

A completed Certification Candidacy Application must be submitted on-line at www.softwarecertifications.org/portal. The ISCB strongly recommends that you submit the application only if you have:

- Satisfied all of the prerequisites for candidacy as stated on page Intro-7.
- Subscribed to the Code of Ethics as described on page Intro-9.
- Reviewed the STBOK and identified those areas that require additional studying.

The entire STBOK is provided in Skill Category 1 through Skill Category 10. A comprehensive list of related references is listed in the appendices.

- Current experience in the field covered by the certification designation.
- Significant experience and breadth to have mastered the basics of the entire STBOK.
- Prepared to take the required examination and therefore ready to schedule and take the examination.

It should not be submitted by individuals who:

- Have not met all of the requirements stated above.
- Are not yet working in the field but who have an interest in obtaining employment in the field (CSTE and CMST).
- Are working in limited areas of the field but would like to expand their work roles to include broader responsibilities (CSTE and CMST).
- Are working in IT but have only marginal involvement or duties related to the certification (CSTE and CMST).
- Are interested in determining if this certification program will be of interest to them.
Candidates for certification who rely on only limited experience, or upon too few or specific study materials, typically do not successfully obtain certification. Many drop out without ever taking the examination. Fees in this program are nonrefundable.

Do not apply for CSTE or CMST unless you feel confident that your work activities and past experience have prepared you for the examination process.

Applicants already holding a certification from the ISCB must still submit a new application when deciding to pursue an additional certification. For example, an applicant already holding a CSQA or CSBA certification must still complete the application process if pursuing the CSTE certification.

**Intro.2.3.1 Updating Your On-Line Profile**

It is critical that candidates keep their on-line profile up-to-date. Many candidates change their residence or job situations during their certification candidacy. If any such changes occur, it is the candidate's responsibility to login to the Software Certification Customer Portal and update their profile as appropriate.

**Intro.2.4 Application-Examination Eligibility Requirements**

The Certification Candidacy begins the date the application fee is processed in the Customer Portal. The candidate then has 12 months from that date to take the initial examination or the candidacy will officially expire. If the application is allowed to expire the individual must reapply for candidacy and pay the current application fee to begin the certification candidacy again.

If the examination is taken inside that 12-month period, then another year is added to the original application length and two more attempts, if required. Candidates for certification must pass a two-part examination in order to obtain certification. The examination tests the candidate's knowledge and practice of the competency areas defined in the STBOK. Candidates who do not successfully pass the examination may re-take the examination up to two times by logging into the Software Certification’s Customer Portal and selecting the retake option and paying all required fees.

Technical knowledge becomes obsolete quickly; therefore the board has established these eligibility guidelines. The goal is to test on a consistent and comparable knowledge base worldwide. The eligibility requirements have been developed to encourage candidates to prepare and pass all portions of the examination in the shortest time possible.
Intro.3  Scheduling with Pearson VUE to Take the Examination

When you have met all of the prerequisites as described above, you are ready to schedule and take the Software Testing examination.

To schedule the Software Testing Certification examination, every candidate must:

- Satisfy all of the qualifications as described in “Meeting the Certification Qualifications” starting on page Intro-7. Be certain that you are prepared and have studied the STBOK and the vocabulary in Appendix A.
- After completing your on-line application you will receive within 24 hours an acknowledgment from Pearson VUE Testing Centers that you are eligible to take the exam at a Pearson VUE site. You will follow the instructions on that acknowledgment email for selecting a testing center location, date and time of your exam.

Intro.3.1  Arriving at the Examination Site

Candidates should arrive at the examination location at least 30 minutes before the scheduled start time of the examination. To check-in at the testing center, candidates must have with them two forms of identification, one of which must be a photo ID. You will receive an email from Pearson VUE regarding arrival instructions.

Intro.3.1.1  No-shows

Candidates who fail to appear for a scheduled examination – initial or retake – are marked as NO SHOW and must submit an on-line Examination Re-sit request to apply for a new examination date. If a candidate needs to change the date and/or time of their certification exam, they must log in directly to the Pearson VUE site to request the change. All changes must be made 24 hours before the scheduled exam or a re-sit fee will be required.

Intro.4  How to Maintain Competency and Improve Value

Maintaining your personal competency is too important to leave to the sole discretion of your employer. In today’s business environment you can expect to work for several different organizations, and to move to different jobs within your own organization. In order to be adequately prepared for these changes you must maintain your personal competency in your field of expertise.
**Intro.4.1 Continuing Professional Education**

Most professions recognize that continuing professional education is required to maintain competency of your skills. There are many ways to get this training, including attending professional seminars and conferences, on-the-job training, attending professional meetings, taking e-learning courses, and attending professional association meetings.

You should develop an annual plan to improve your personal competencies. Getting 120 hours of continuing professional education will enable you to recertify your Software Testing designation.

**Intro.4.2 Advanced Software Testing Designations**

You can use your continuing professional education plan to improve and demonstrate your value to your employer. Your employer may have difficulty assessing improved competencies attributable to the continuing professional education you are acquiring. However, if you can use that continuing education effort to obtain an advanced certification, you can demonstrate to your employer your increased value to the organization by acquiring an advanced certification.

**Intro.4.2.1 What is the Certification Competency Emphasis?**

The drivers for improving performance in IT are the quality assurance and quality control (testing) professionals. Dr. W. Edward Deming recognized this “do-check” partnership of quality professionals in his “14 points” as the primary means for implementing the change needed to mature. Quality control identifies the impediments to quality and quality assurance facilitates the fix. Listed below is the certification level, emphasis of each certification, and how you can demonstrate that competency.

- **CAST**
  Demonstrate competency in knowing what to do.
  Study for, and pass, a one-part examination designed to evaluate the candidate’s knowledge of the principles and concepts incorporated into the STBOK.

- **CSTE**
  Demonstrate competency in knowing what to do and how to do it.
  Study for, and pass, a two-part examination designed to evaluate the candidate’s knowledge of the principles and concepts incorporated into the STBOK, plus the ability to relate those principles and concepts to the challenges faced by IT organizations.

- **CMST**
  Demonstrate competency in knowing how to solve management level challenges.
Candidates must demonstrate their ability to develop real solutions to challenges in their IT organizations, by proposing a solution to a real-world management problem.
The “basics” of software testing are represented by the vocabulary of testing, testing approaches, methods and techniques, as well as, the materials used by testers in performing their test activities.

<table>
<thead>
<tr>
<th>Vocabulary</th>
<th>1-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality Basics</td>
<td>1-2</td>
</tr>
<tr>
<td>Understanding Defects</td>
<td>1-19</td>
</tr>
<tr>
<td>Process and Testing Published Standards</td>
<td>1-26</td>
</tr>
<tr>
<td>Software Testing</td>
<td>1-34</td>
</tr>
<tr>
<td>Software Development Life Cycle (SDLC) Models</td>
<td>1-45</td>
</tr>
<tr>
<td>Agile Development Methodologies</td>
<td>1-59</td>
</tr>
<tr>
<td>Testing Throughout the Software Development Life Cycle (SDLC)</td>
<td>1-63</td>
</tr>
<tr>
<td>Testing Schools of Thought and Testing Approaches</td>
<td>1-69</td>
</tr>
<tr>
<td>Test Categories and Testing Techniques</td>
<td>1-72</td>
</tr>
</tbody>
</table>
1.1 Vocabulary

A unique characteristic of a profession is its vocabulary. The profession’s vocabulary represents the knowledge of the profession and its ability to communicate with others about the profession’s knowledge. For example, in the medical profession one hundred years ago doctors referred to “evil spirits” as a diagnosis. Today the medical profession has added words such as cancer, AIDS, and stroke, which communicate knowledge.

This Software Testing Body of Knowledge (STBOK) defines many terms used by software testing professionals today. To aid in preparing for the certification exam, key definitions have been noted at the beginning of sections as shown in Figure 1-1. It is suggested you create a separate vocabulary list and write down the definitions as they are called out in the text. It is also a good practice to use an Internet search engine to search the definitions and review other examples of those terms. However, some variability in the definition of words may exist, so for the purpose of preparing for the examination, a definition given in the STBOK is the correct usage as recognized on the examination.

Appendix A of the STBOK is a glossary of software testing terms. However, learning them as used in the appropriate context is the best approach.

1.2 Quality Basics

The term “quality” will be used throughout the STBOK. This should come as no surprise as a primary goal of the software test professional is to improve the quality of software applications. In fact, many software testing professionals have titles such as Software Quality Engineer or Software Quality Analyst. Before we begin the discussion of software testing, we will first describe some “quality basics.”

1.2.1 Quality Assurance Versus Quality Control

There is often confusion in the IT industry regarding the difference between quality control and quality assurance. Many “quality assurance” groups, in fact, practice quality control. Quality methods can be segmented into two categories: preventive methods and detective methods. This distinction serves as the mechanism to distinguish quality assurance activities from quality control.
activities. This discussion explains the critical difference between control and assurance, and how to recognize a Quality Control practice from a Quality Assurance practice.

Quality has two working definitions:

- Producer’s Viewpoint – The product meets the requirements.
- Customer’s Viewpoint – The product is “fit for use” or meets the customer’s needs.

There are many “products” produced from the software development process in addition to the software itself, including requirements, design documents, data models, GUI screens, and programs. To ensure that these products meet both requirements and user needs, quality assurance and quality control are both necessary.

### 1.2.1.1 Quality Assurance

Quality assurance is an activity that establishes and evaluates the processes that produce products. If there is no need for process, there is no role for quality assurance. Quality assurance is a staff function, responsible for implementing the quality plan defined through the development and continuous improvement of software development processes. Quality assurance activities in an IT environment would determine the need for, acquire, or help install:

- System development methodologies
- Estimation processes
- System maintenance processes
- Requirements definition processes
- Testing processes and standards

Once installed, quality assurance would measure these processes to identify weaknesses, and then correct those weaknesses to continually improve the process.

### 1.2.1.2 Quality Control

Quality control is the process by which product quality is compared with applicable standards and actions are taken when nonconformance is detected. Quality control is a line function, and the work is done within a process to ensure that the work product conforms to standards and requirements.

---

**Testing is a Quality Control Activity.**

---

Quality control activities focus on identifying defects in the actual products produced. These activities begin at the start of the software development process with reviews of requirements and continue until all testing is complete.
It is possible to have quality control without quality assurance. For example, a test team may be in place to conduct system testing at the end of development, regardless of whether the organization has a quality assurance function in place.

The following statements help differentiate between quality control and quality assurance:

- Quality assurance helps establish processes.
- Quality assurance sets up measurement programs to evaluate processes.
- Quality assurance identifies weaknesses in processes and improves them.
- Quality assurance is a management responsibility, frequently performed by a staff function.
- Quality assurance is concerned with the products across all projects where quality control is product line focused.
- Quality assurance is sometimes called quality control over quality control because it evaluates whether quality control is working.
- Quality assurance personnel should never perform quality control unless it is to validate quality control.
- Quality control relates to a specific product or service.
- Quality control verifies whether specific attribute(s) are in, or are not in, a specific product or service.
- Quality control identifies defects for the purpose of correcting defects.

Both quality assurance and quality control are separate and distinct from the internal audit function. Internal Auditing is an independent appraisal activity within an organization for the review of operations, and is a service to management. It is a managerial control that functions by measuring and evaluating the effectiveness of other controls.

1.2.2 Quality, A Closer Look

The definition of “quality” is a factor in determining the scope of software testing. Although there are multiple quality definitions in existence, it is important to note that most contain the same core components:

- Quality is based upon customer satisfaction.
- Your organization must define quality before it can be achieved.
- Management must lead the organization through any quality improvement efforts.

There are five perspectives of quality – each of which should be considered as important to the customer:
1. Transcendent – I know it when I see it

2. Product Based – Possesses desired features

3. User Based – Fitness for use

4. Development and Manufacturing Based – Conforms to requirements

5. Value Based – At an acceptable cost

Peter R. Scholtes\(^1\) introduces the contrast between effectiveness and efficiency. Quality organizations must be both effective and efficient.

Patrick Townsend\(^2\) examines quality in fact and quality in perception as shown in Table 1-1. Quality in fact is usually the supplier’s point of view, while quality in perception is the customer’s. Any difference between the former and the latter can cause problems between the two.

<table>
<thead>
<tr>
<th>QUALITY IN FACT</th>
<th>QUALITY IN PERCEPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doing the right thing.</td>
<td>Delivering the right product.</td>
</tr>
<tr>
<td>Doing it the right way.</td>
<td>Satisfying our customer’s needs.</td>
</tr>
<tr>
<td>Doing it right the first time.</td>
<td>Meeting the customer’s expectations.</td>
</tr>
<tr>
<td>Doing it on time.</td>
<td>Treating every customer with integrity, courtesy, and respect.</td>
</tr>
</tbody>
</table>

*Table 1-1 Townsend's Quality View*

An organization’s quality policy must define and view quality from their customer's perspectives. If there are conflicts, they must be resolved.

### 1.2.3 What is Quality Software?

As discussed earlier, there are two important definitions of quality software:

*The producer’s view of quality software means meeting requirements.*

*Customer’s/User’s view of quality software means fit for use.*

---

These two definitions are not inconsistent. Meeting requirements is the producer’s definition of quality; it means that the producer develops software in accordance with requirements. The fit for use definition is a user’s definition of software quality; it means that the software developed by the producer meets the user’s need regardless of the software requirements.

1.2.3.1 The Two Software Quality Gaps

In most IT groups, there are two gaps as illustrated in Figure 1-2. These gaps represent the different perspectives of software quality as seen by the producer and the customer.

![Figure 1-2 The Two Software Quality Gaps](image)

The first gap is the producer gap. It is the gap between what was specified to be delivered, meaning the documented requirements and internal IT standards, and what was actually delivered. The second gap is between what the producer actually delivered compared to what the customer expected.

A significant role of software testing is helping to close the two gaps. The IT quality function must first improve the processes to the point where IT can produce the software according to requirements received and its own internal standards. The objective of the quality function closing the producer’s gap is to enable an IT function to provide consistency in what it can produce. This is referred to as the “McDonald’s effect.” This means that when you go into any McDonald’s in the world, a Big Mac should taste the same. It doesn’t mean that you as a customer like the Big Mac or that it meets your needs but rather that McDonald’s has now produced consistency in its delivered product.
To close the customer’s gap, the IT quality function must understand the true needs of the user. This can be done by the following:

- Customer surveys
- JAD (joint application development) sessions – the producer and user come together and negotiate and agree upon requirements
- More user involvement while building information products
- Implementing Agile development strategies

Continuous process improvement is necessary to close the user gap so that there is consistency in producing software and services that the user needs. Software testing professionals can participate in closing these “quality” gaps.

### 1.2.3.2 What is the Quality Message?

The Random House College Dictionary defines excellence as "superiority; eminence.” Excellence, then, is a measure or degree of quality. These definitions of quality and excellence are important because it is a starting point for any management team contemplating the implementation of a quality policy. They must agree on a definition of quality and the degree of excellence they want to achieve.

The common thread that runs through today's quality improvement efforts is the focus on the customer and, more importantly, customer satisfaction. The customer is the most important person in any process. Customers may be either internal or external. The question of customer satisfaction (whether that customer is located in the next workstation, building, or country) is the essence of a quality product. Identifying customers' needs in the areas of what, when, why, and how are an essential part of process evaluation and may be accomplished only through communication.

The internal customer is the person or group that receives the results (outputs) of any individual's work. The outputs may include a product, a report, a directive, a communication, or a service. Customers include peers, subordinates, supervisors, and other units within the organization. To achieve quality the expectations of the customer must be known.

External customers are those using the products or services provided by the organization. Organizations need to identify and understand their customers. The challenge is to understand and exceed their expectations.

An organization must focus on both internal and external customers and be dedicated to exceeding customer expectations.

### 1.2.3.3 Improving Software Quality

There are many strategic and tactical activities that help improve software quality. Listed below are several such activities that can help the IT development team and specifically the quality function to improve software quality.
• Explicit software quality objectives: Making clear which qualities are most important
• Explicit quality assurance activities: Ensuring software quality is not just an afterthought to grinding out ‘code’
• Testing strategy: Planning and conducting both static testing (reviews, inspections) and dynamic testing (unit, integrations, system and user acceptance testing)
• Software engineering guidelines: Specifying recommendations/rules/standards for requirements analysis, design, coding and testing
• Informal technical reviews: Reviewing specifications, design, and code alone or with peers
• Formal technical reviews: Conducting formal reviews at well-defined milestones (requirements/architecture, architecture/detailed design, detailed design/coding, and coding/testing)
• External audits: Organizing technical reviews conducted by outside personnel, usually commissioned by management
• Development processes: Using development processes with explicit risk management
• Change control procedures: Using explicit procedures for changing requirements, design, and code; documenting the procedures and checking them for consistency
• Measurement of results: Measuring effects of quality assurance activities

1.2.4 The Cost of Quality

When calculating the total costs associated with the development of a new application or system, four cost components must be considered. The Cost of Quality (CoQ), as seen in Figure 1-3, is all the costs that occur beyond the cost of producing the product “right the first time.” Cost of Quality is a term used to quantify the total cost of prevention and appraisal, and costs associated with the failure of software.
The three categories of costs associated with producing quality products are:

- **Prevention Costs**
  Resources required to prevent errors and to do the job right the first time. These normally require up-front expenditures for benefits that will be derived later. This category includes money spent on establishing methods and procedures, training workers, acquiring tools, and planning for quality. Prevention resources are spent before the product is actually built.

- **Appraisal Costs**
  Resources spent to ensure a high level of quality in all development life cycle stages which includes conformance to quality standards and delivery of products that meet the user’s requirements/needs. Appraisal costs include the cost of in-process reviews, dynamic testing, and final inspections.

- **Failure Costs**
  All costs associated with defective products that have been delivered to the user and/or moved into production. Failure costs can be classified as either “internal” failure costs or “external” failure costs. Internal failure costs are costs that are caused by products or services not conforming to requirements or customer/user needs and are found before deployment of the application to production or delivery of the product to external customers. Examples of internal failure costs are: rework, re-testing, delays within the life cycle, and lack of certain quality factors such as flexibility. Examples of external failure costs include: customer complaints, lawsuits, bad debt, losses of revenue, and the costs associated with operating a Help Desk.

Collectively the Preventive Costs and Appraisal Costs are referred to as the “Costs of Control (Costs of Conformance).” They represent the costs of “good quality.” Failure Costs are...
described as the “Costs of Failure of Control (Costs of Non-Conformance).” Failure Costs represent the cost of “poor quality.”

The iceberg diagram illustrated in Figure 1-4 is often used to depict how the more visible CoQ factors make up only a portion of the overall CoQ costs. When viewing the cost of quality from a broader vantage point the true costs are revealed.

![Iceberg Diagram](image)

Figure 1-4 Iceberg Diagram

The Cost of Quality will vary from one organization to the next. The majority of costs associated with the Cost of Quality are associated with failure costs, both internal and external. Studies have shown that on many IT projects the cost of quality can make up as much as 50% of the overall costs to build a product. These studies have shown that of the 50% Cost of Quality Costs, 3% are Preventive Costs, 7% appraisal Costs, and 40% failure Costs.

The concept of “quality is free” goes to the heart of understanding the costs of quality. If you can identify and eliminate the causes of problems early, it reduces rework, warranty costs, and inspections which logically follows that creating quality goods and services does not cost money, it saves money.
Software Testing Principles and Concepts

Figure 1-5 Preventive Costs Return on Investment

The IT quality assurance group must identify the costs within these three categories, quantify them, and then develop programs to minimize the totality of these three costs. Applying the concepts of continuous testing to the systems development process can reduce the Cost of Quality.

1.2.5 Software Quality Factors

Software quality factors are attributes of the software that, if they are wanted and not present, pose a risk to the success of the software and thus constitute a business risk. For example, if the software is not easy to use, the resulting processing may be done incorrectly. Identifying the software quality factors and determining their priority enables the test process to be logically constructed.

This section addresses the problem of identifying software quality factors that are in addition to the functional, performance, cost, and schedule requirements normally specified for software development. The fact that the goals established are related to the quality of the end product should, in itself, provide some positive influence on the development process.

The software quality factors should be documented in the same form as the other system requirements. Additionally, a briefing emphasizing the intent of the inclusion of the software quality factors is recommended.

1.2.5.1 Defining the Software Quality Factors

Figure 1-6 illustrates the Diagram of Software Quality Factors as described by Jim McCall. McCall produced this model for the US Air Force as a means to help “bridge the gap” between users and developers. He mapped the user view with the developer’s priority.
McCall identified three main perspectives for characterizing the quality attributes of a software product. These perspectives are:

- Product operations (basic operational characteristics)
- Product revision (ability to change)
- Product transition (adaptability to new environments)

<table>
<thead>
<tr>
<th>Quality Categories</th>
<th>Quality Factors</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Operation</td>
<td>Correctness</td>
<td>Extent to which a program satisfies its specifications and fulfills the user’s objective.</td>
</tr>
<tr>
<td></td>
<td>Reliability</td>
<td>Extent to which a program can be expected to perform its intended function with required precision.</td>
</tr>
<tr>
<td></td>
<td>Efficiency</td>
<td>The amount of computing resources and code required by a program to perform a function.</td>
</tr>
<tr>
<td></td>
<td>Integrity</td>
<td>Extent to which access to software or data by unauthorized persons can be controlled.</td>
</tr>
<tr>
<td></td>
<td>Usability</td>
<td>Effort required to learn, operate, prepare input, and interpret output of a program.</td>
</tr>
<tr>
<td>Product Revision</td>
<td>Maintainability</td>
<td>Effort required to locate and fix an error in an operational program.</td>
</tr>
<tr>
<td></td>
<td>Testability</td>
<td>Effort required testing a program to ensure that it performs its intended function.</td>
</tr>
<tr>
<td></td>
<td>Flexibility</td>
<td>Effort required to modify an operational program.</td>
</tr>
</tbody>
</table>
Software Testing Principles and Concepts

Table 1-2  Software Quality Factors

The quality factors represent the behavioral characteristics of a system.

<table>
<thead>
<tr>
<th>Quality Categories</th>
<th>Quality Factors</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Transition</td>
<td>Portability</td>
<td>Effort required to transfer software from one configuration to another.</td>
</tr>
<tr>
<td>Reusability</td>
<td></td>
<td>Extent to which a program can be used in other applications.</td>
</tr>
<tr>
<td>Interoperability</td>
<td></td>
<td>Effort required to couple one system with another.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quality Categories</th>
<th>Quality Factors</th>
<th>Broad Objectives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Operation</td>
<td>Correctness</td>
<td>Does it do what the customer wants?</td>
</tr>
<tr>
<td></td>
<td>Reliability</td>
<td>Does it do it accurately all the time?</td>
</tr>
<tr>
<td></td>
<td>Efficiency</td>
<td>Does it quickly solve the intended problem?</td>
</tr>
<tr>
<td></td>
<td>Integrity</td>
<td>Is it secure?</td>
</tr>
<tr>
<td></td>
<td>Usability</td>
<td>Is it easy to use?</td>
</tr>
<tr>
<td>Product Revision</td>
<td>Maintainability</td>
<td>Can it be fixed?</td>
</tr>
<tr>
<td></td>
<td>Testability</td>
<td>Can it be tested?</td>
</tr>
<tr>
<td></td>
<td>Flexibility</td>
<td>Can it be changed?</td>
</tr>
<tr>
<td>Product Transition</td>
<td>Portability</td>
<td>Can it be used on another machine?</td>
</tr>
<tr>
<td></td>
<td>Reusability</td>
<td>Can parts of it be reused?</td>
</tr>
<tr>
<td></td>
<td>Interoperability</td>
<td>Can it interface with another system?</td>
</tr>
</tbody>
</table>

Table 1-3  The Broad Objectives of Quality Factors

1.2.5.2  Software Quality Criteria

McCall extended the model an additional layer by defining for each quality factor one or more quality criteria (a way of measurement); in this way, an overall quality assessment could be made of a given software product by evaluating the criteria for each factor. A quality criterion is an attribute of a quality factor that is related to software development. For example, modularity is an attribute of the architecture of a software system. Highly modular code allows designers to put cohesive components in one module, thereby increasing the maintainability of the system. Below is a list of these quality characteristics:
1. Access audit: Ease with which software and data can be checked for compliance with standards or other requirements.

2. Access control: Provisions for control and protection of the software and data.

3. Accuracy: Precision of computations and output.

4. Communication commonality: Degree to which standard protocols and interfaces are used.

5. Completeness: Degree to which a full implementation of the required functionalities has been achieved.

6. Communicativeness: Ease with which inputs and outputs can be assimilated.

7. Conciseness: Compactness of the source code, in terms of lines of code.

8. Consistency: Use of uniform design and implementation techniques and notation throughout a project.


10. Error tolerance: Degree to which continuity of operation is ensured under adverse conditions.

11. Execution efficiency: Run time efficiency of the software.

12. Expandability Degree to which storage requirements or software functions can be expanded.


14. Hardware independence: Degree to which the software is independent on the underlying hardware.

15. Instrumentation: Degree to which the software provides for measurement of its use or identification of errors.


17. Operability: Ease of operation of the software.


19. Simplicity: Ease with which the software can be understood.

20. Software system independence: Degree to which the software is independent of its software environment - nonstandard language constructs, operating system, libraries, database management system, etc.

21. Software efficiency: Run time storage requirements of the software.
22. Traceability: Ability to link software components to requirements.

23. Training: Ease with which new users can use the system.

Figure 1-7 shows the mapping of quality characteristics to their respective quality factors.

![Quality Criteria Map]

Each quality factor is positively influenced by a set of quality criteria, and the same quality criterion impacts a number of quality factors. For example, simplicity impacts reliability, usability, and testability.

It is important to note that improving one quality factor may or may not mean related quality factors or criteria will improve. Some quality factors positively impact others; for example, an effort to improve the correctness of a system will increase its reliability. However, in some cases an effort is made to improve one quality factor, another quality factor may be degraded. For example, portable code may be less efficient.

The ISO 25010:2011 Standard defines two models relating to software product quality. Each model has a set of unique characteristics and sub-characteristics.

The *product quality* model is composed of eight characteristics which are further subdivided into sub-characteristics. The eight characteristics relate to static properties of software and dynamic properties of the computer system. The model is applicable to both computer systems and software products. (See Table 1-4)

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional suitability</td>
<td>The degree to which the product provides functions that meet stated and implied needs when the product is used under specified conditions</td>
</tr>
<tr>
<td></td>
<td>Sub-characteristics:</td>
</tr>
<tr>
<td></td>
<td>Suitability</td>
</tr>
<tr>
<td></td>
<td>Accuracy</td>
</tr>
<tr>
<td></td>
<td>Interoperability</td>
</tr>
<tr>
<td></td>
<td>Security</td>
</tr>
<tr>
<td></td>
<td>Compliance</td>
</tr>
<tr>
<td>Reliability</td>
<td>The degree to which a system or component performs specified functions under specified conditions for a specified period of time</td>
</tr>
<tr>
<td></td>
<td>Sub-characteristics:</td>
</tr>
<tr>
<td></td>
<td>Maturity</td>
</tr>
<tr>
<td></td>
<td>Fault Tolerance</td>
</tr>
<tr>
<td></td>
<td>Recoverability</td>
</tr>
<tr>
<td></td>
<td>Compliance</td>
</tr>
<tr>
<td><strong>Characteristic</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------</td>
</tr>
</tbody>
</table>
| Operability        | The degree to which the product has attributes that enable it to be understood, learned, used and attractive to the user, when used under specified conditions  
  Sub-characteristics:  
  Appropriateness  
  Recognizability  
  Ease of use  
  Learnability  
  Attractiveness  
  Technical accessibility  
  Compliance |
| Performance efficiency | The performance relative to the amount of resources used under stated conditions  
  Sub-characteristics:  
  Time Behavior  
  Resource Utilization  
  Compliance |
| Security           | The degree of protection of information and data so that unauthorized persons or systems cannot read or modify them and authorized persons or systems are not denied access to them  
  Sub-characteristics:  
  Confidentiality  
  Integrity  
  Non-repudiation  
  Accountability  
  Authenticity  
  Compliance |
| Compatibility      | The degree to which two or more systems or components can exchange information and/or perform their required functions while sharing the same hardware or software environment  
  Sub-characteristics:  
  Replaceability  
  Co-existence  
  Interoperability  
  Compliance |
The quality in use model is composed of five characteristics (some of which are further subdivided into sub-characteristics) that relate to the outcome of interaction when a product is used in a particular context of use. This system model is applicable to the complete human-computer system, including both computer systems in use and software products in use.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effectiveness</td>
<td>The accuracy and completeness with which users achieve specified goals</td>
</tr>
<tr>
<td>Efficiency</td>
<td>The resources expended in relation to the accuracy and completeness with which users achieve goals</td>
</tr>
</tbody>
</table>
1.3 Understanding Defects

A defect is an undesirable state. There are two types of defects: process and product. For example, if a Test Plan Standard is not followed, it would be a process defect. However, if the Test Plan did not contain a Statement of Usability as specified in the Requirements documentation it would be a product (the test plan) defect.
1.3.1 Software Process Defects

Ideally, the software development process should produce the same results each time the process is executed. For example, if we follow a process that produced one function-point-of-logic in 100 person hours, we would expect that the next time we followed that process, we would again produce one function-point-of-logic in 100 hours. However, if we follow the process the second time and it took 110 hours to produce one function-point-of-logic, we would state that there is “variability” in the software development process. Variability is the “enemy” of quality – the concepts behind maturing a software development process are to reduce variability.

The concept of measuring and reducing variability is commonly called statistical process control (SPC).

To understand SPC we need to first understand the following:

• What constitutes an in control process
• What constitutes an out of control process
• What are some of the steps necessary to reduce variability within a process

Testers need to understand process variability, because the more variability in the process the greater the need for software testing. Following is a brief tutorial on processes and process variability.

1.3.1.1 What Does It Mean For a Process To Be In or Out of Control?

The amount of variation in a process is quantified with summary statistics; typically, the standard deviation is used. A process is defined as stable if its parameters (i.e., mean and standard deviation) remain constant over time; it is then said to be in a state of statistical control. Figure 1-8 illustrates a stable process. Such a process is predictable, i.e., we can predict, within known limits and with a stated degree of belief, future process values. Accepted practice uses a prediction interval three standard deviation distances in width around the population mean (μ ± 3σ) in establishing the control limits.
Continuous process improvement through the use of quantitative methods and employee involvement sets quality management apart from other attempts to improve productivity. Continuous process improvement is accomplished by activating teams and providing them with quantitative methods such as SPC techniques and supporting them as they apply these tools. We will further discuss the concept of variation, common and special causes of variation.

Variation is present in all processes. The natural change occurring in organizational life moves systems and processes towards increasing variation. Statistical methods help us collect and present data in ways that facilitate the evaluation of current theories and the formation of new theories. These tools are the only methods available for quantifying variation. Since the key to quality is process consistency, variation (the lack of consistency) must be understood before any process can be improved. Statistical methods are the only way to objectively measure variability. There is no other way!

One of the challenges in implementing quality management is to get those working in the process thinking in terms of sources of variation. How much of the observed variation can be attributed to measurements, material, machines, methods, people, and the environment?

Consistency in all the processes, from conception through delivery, of a product or service is the cornerstone of quality. Paradoxically, the route to quality is not just the application of SPC and the resulting control charts. Managers must change the way they manage. They must use statistical methods in making improvements to management processes as well as all other processes in the organization.

### 1.3.1.2 Common Cause of Variability

Common causes of variation are typically due to a large number of small random sources of variation. The sum of these sources of variation determines the magnitude of the process’s inherent variation due to common causes; the process’s control limits and current process capability can then be determined. Figure 1-9 illustrates an out of control process.
Brian Joiner has provided several thoughts on common causes of variation:

- Process inputs and conditions that regularly contribute to the variability of process outputs.
- Common causes contribute to output variability because they themselves vary.
- Each common cause typically contributes a small portion to the total variation in process outputs.
- The aggregate variability due to common causes has a “nonsystematic,” random-looking appearance.
- Because common causes are “regular contributors,” the “process” or “system” variability is defined in terms of them.

Joiner also outlined a strategy for reducing common causes of variation:

- Talk to lots of people including local employees, other managers, and staff from various functions.
- Improve measurement processes if measuring contributes too much to the observed variation.
- Identify and rank categories of problems by Pareto analysis (a ranking from high to low of any occurrences by frequency).
- Stratify and desegregate your observations to compare performance of sub-processes.
- Investigate cause-and-effect relations. Run experiments (one factor and multifactor).

### 1.3.1.3 Special Cause of Variability

Special causes of variation are not typically present in the process. They occur because of special or unique circumstances. If special causes of variation exist, the process is unstable or

---

   From an address given at a Deming User’s Group Conference in Cincinnati, OH.
unpredictable. A state of statistical control is established when all special causes of variation
have been eliminated.

The elimination of the special cause may happen without any specific action on the part of the
process owner. For example, a power outage from a hurricane causing a special cause of
variation in a process may be resolved simply by the power being restored after the storm. No
particular change to the process is made. If the special cause, while unpredictable, is a
potential threat, then contingency planning may be necessary to eliminate a future process
threat.

Joiner has summarized special causes of variation as follows:

- Process inputs and conditions that sporadically contribute to the variability of process
  outputs.
- Special causes contribute to output variability because they themselves vary.
- Each special cause may contribute a “small” or “large” amount to the total variation in
  process outputs.
- The variability due to one or more special causes can be identified by the use of
  control charts.
- Because special causes are “sporadic contributors,” due to some specific
  circumstances, the “process” or “system” variability is defined without them.

Joiner then presents this strategy for eliminating special causes of variation:

- Work to get very timely data so that special causes are signaled quickly – use early
  warning indicators throughout your operation.
- Immediately search for the cause when the control chart gives a signal that a special
  cause has occurred. Find out what was different on that occasion from other occasions.
- Do not make fundamental changes in that process.
- Instead, seek ways to change some higher-level systems to prevent that special cause
  from recurring.

1.3.1.4 Reducing Variability

Management is responsible for leading the effort to reduce common cause variation. These
higher-level improvements to the process usually require process or system changes. It is now
widely recognized that as much as 94% of problems in any organization are system problems
and the responsibility of management to solve. The concept of statistical control allows us to
determine which problems are in the process (due to common causes of variation) and which
are external to the process (due to special causes of variation).

Reducing variation due to common causes is process improvement and the real essence of
continuous process improvement.

As previously mentioned, variation due to special causes must be identified and removed to
create a stable process. However, a stable process may not be an acceptable process. If its
variation, due to common causes, results in operation of the process beyond specifications, the
process is called “incapable.” The process must be improved, i.e., variation due to common cause must be reduced or the process retargeted or both. Figure 1-10 illustrates the transition of a process from incapable to capable.

![Diagram of process capability]

**Figure 1-10 Making a Process Capable**

Deming defines tampering as “action taken on a stable system in response to variation within statistical control, in an effort to compensate for this variation – the results of which will inevitably increase the variation and will increase cost from here on out.” Tampering is any adjustment to a process (typically by operator or machine) in response to variation due to common causes (i.e., that variation between the control limits). By definition, process variation (due to common causes) is expected and is not a reason for adjusting or changing the process (tampering). Management that does not understand variation, time and time again asks for an explanation or corrective action when confronted with variation due to common causes.

### 1.3.1.5 Do Testers Need to Understand Statistical Process Control?

Testing is a measurement process. It attempts to measure the implemented software against both specifications and user needs. Statistical process control is a measurement tool.

The more you know about the process used to develop software, the more effective the testing process will become. For example, if you know that the requirements process has significant variability, meaning there is a high probability that the requirements as defined are not correct, you should then focus testing efforts on determining the “correctness” of the requirements as viewed by the customer. Software testing does not add a lot of value to the business if all they are doing is validating that incorrect requirements are implemented correctly.
1.3.2 Software Process and Product Defects

As previously stated, there are two types of defects: process defects and product defects. It is often stated that the “quality of the software product is directly related to the quality of the process used to develop and maintain it.” The manifestation of software process defects are product defects. Testing focuses on discovering and eliminating product defects or variances from what is expected. Testers need to identify two types of product defects:

- Variance from specifications - A defect from the perspective of the developer of the product
- Variance from what is desired - A defect from a user (or customer) perspective

Typical software process and product defects include:

- IT improperly interprets requirements
  IT staff misinterprets what the user wants, but correctly implements what the IT people believe is wanted.

- Users specify the wrong requirements
  The specifications given to IT are erroneous.

- Requirements are incorrectly recorded
  Specifications are recorded improperly.

- Design specifications are incorrect
  The application system design does not achieve the system requirements, but the design as specified is implemented correctly.

- Program specifications are incorrect
  The design specifications are incorrectly interpreted, making the program specifications inaccurate; however, it is possible to properly code the program to achieve the specifications.

- Errors in program coding
  The program is not coded according to the program specifications.

- Data entry errors
  Data entry staff incorrectly inputs information.

- Testing errors
  Tests either falsely detect an error or fail to detect one.

- Mistakes in error correction
  Your implementation team makes errors in implementing solutions.

- The corrected condition causes another defect

In the process of correcting a defect, the correction process itself injects additional defects into the application system.
1.4 Process and Testing Published Standards

In the early days of computing, experience showed that some software development processes were much more effective than others. As the software industry grew, the need for standards within the software engineering discipline became apparent. Many global “standards” organizations like the International Organization of Standardization (ISO) prescribe standards to improve the quality of the software. Listed in Table 1-6 are some of the relevant standards for software development, quality assurance, and testing. Sections 1.4.1 to 1.4.3 detail three specifically; CMMI, TMMi, and ISO 29119.

<table>
<thead>
<tr>
<th>Standard</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMMI-Dev</td>
<td>A process improvement model for software development.</td>
</tr>
<tr>
<td>TMMI</td>
<td>A process improvement model for software testing.</td>
</tr>
<tr>
<td>ISO/IEC/IEEE 29119</td>
<td>A set of standards for software testing.</td>
</tr>
<tr>
<td>ISO/IEC 12119</td>
<td>A standard that establishes requirements for software packages and instructions on how to test a software package against those requirements.</td>
</tr>
<tr>
<td>IEEE 829</td>
<td>A standard for the format of documents used in different stages of software testing.</td>
</tr>
<tr>
<td>IEEE 1061</td>
<td>Defines a methodology for establishing quality requirements, identifying, implementing, analyzing, and validating the process and product of software quality metrics.</td>
</tr>
<tr>
<td>IEEE 1008</td>
<td>A standard for unit testing.</td>
</tr>
<tr>
<td>IEEE 1012</td>
<td>A standard for Software Verification and Validation.</td>
</tr>
<tr>
<td>IEEE 1028</td>
<td>A standard for software inspections</td>
</tr>
<tr>
<td>IEEE 1044</td>
<td>A standard for the classification of software anomalies.</td>
</tr>
<tr>
<td>IEEE 1044-1</td>
<td>A guide to the classification of software anomalies.</td>
</tr>
<tr>
<td>IEEE 830</td>
<td>A guide for developing system requirements specifications.</td>
</tr>
<tr>
<td>IEEE 730</td>
<td>A standard for software quality assurance plans.</td>
</tr>
</tbody>
</table>
1.4.1 CMMI® for Development

Over five thousand organizations in 94 countries use SEI’s maturity model to improve their software development process. CMMI does not provide a single process, but rather the CMMI framework models what to do to improve processes, not define the processes. Specifically, CMMI for Development is designed to compare an organization’s existing development processes to proven best practices developed by members of industry, government, and academia. Through this comparison, organizations identify possible areas for improvement. Table 1-7 illustrates CMMI’s five levels of maturity.

<table>
<thead>
<tr>
<th>Level</th>
<th>Focus</th>
<th>Process Areas</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>5 Optimizing</strong></td>
<td>Continuous process improvement</td>
<td>Organization Innovation and Deployment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Causal Analysis and Resolution</td>
</tr>
<tr>
<td><strong>4 Quantitatively Managed</strong></td>
<td>Quantitative management</td>
<td>Organizational Process Performance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Quantitative Project Management</td>
</tr>
</tbody>
</table>

**Table 1-6 List of Standards**

<table>
<thead>
<tr>
<th>Standard</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE 1061</td>
<td>A standard for software quality metrics and methodology.</td>
</tr>
<tr>
<td>IEEE 12207</td>
<td>A standard for software life cycle processes and life cycle data.</td>
</tr>
<tr>
<td>BS 7925-1</td>
<td>A vocabulary of terms used in software testing.</td>
</tr>
<tr>
<td>BS 7925-2</td>
<td>A standard for software component testing.</td>
</tr>
</tbody>
</table>
1.4.1.1  **Level 1 – Initial**

Ad hoc means unstructured, inconsistent levels of performance. At the ad hoc level, tasks are not performed the same way by different people or different teams. For example, one system development team may use part of the system development methodology, but improvise other parts; another team may select different parts of the same system development methodology to use, and decide not to perform tasks done by a different team.

At this level, management manages people and jobs. Management will establish goals or objectives for individuals and teams and manage to those objectives and goals with minimal concern about the means used to achieve the goals. This level is normally heavily schedule driven, and those that meet the schedules are rewarded. Since there are not standards against which to measure deliverables, people’s performance is often dependent upon their ability to convince management that the job they have done is excellent. This causes the environment to be very political. Both management and staff become more concerned with their personal agenda than with meeting their organization’s mission.

The emphasis needed to move from Level 1 to Level 2 is discipline and control. The work processes need to be defined, the people need to be trained in the work processes, controls need to be implemented to ensure compliance to the work processes, and products produced that meet predefined standards.

1.4.1.2  **Level 2 – Managed**

There are two major objectives to be achieved at Level 2. The first is to instill discipline in the culture of the information organization so that, through the infrastructure, training, and leadership of management, individuals will want to follow defined processes. The second objective is to reduce variability in the processes by defining them to a level that permits relatively constant outputs. At this level, processes are defined with minimal regard to skills needed to perform the process AND with minimal regard to the impact on other processes. At
Level 2, the work processes are defined; management manages those processes, and uses validation and verification techniques to check compliance to work procedures and product standards. Having the results predefined through a set of standards enables management to measure people’s performance against meeting those standards. Education and training are an important component of Level 2 as is building an infrastructure that involves the entire staff in building and improving work processes.

The emphasis that needs to be put into place to move to Level 3 is defining and building the information group’s core competencies.

1.4.1.3 Level 3 – Defined

At maturity level 3, processes are well characterized and understood, and are described in standards, procedures, tools, and methods. The organization’s set of standard processes, which is the basis for maturity level 3, is established and improved over time. These standard processes are used to establish consistency across the organization. A critical distinction between level 2 and level 3 is that at maturity level 3, the processes are typically described more rigorously than at maturity level 2. A defined process clearly states the purpose, inputs, entry criteria, activities, roles, measures, verification steps, outputs, and exit criteria. At maturity level 3, processes are managed more proactively using an understanding of the interrelationships of process activities and detailed measures of the process, its work products, and its services.

The managerial emphasis that is needed to move to level 4 is quantitative measurement. Measurement is only a practical initiative when the processes are stabilized and focused on achieving management’s desired results.

1.4.1.4 Level 4 – Quantitatively Managed

This level has two objectives. The first is to develop quantitative standards for the work processes based on performance of the Level 3 stabilized processes. The second objective is to provide management the dashboards and skill sets needed to manage quantitatively. The result is predictable work processes. Knowing the normal performance of a work process, management can easily identify problems through variation from the quantitative standards and address problems quickly to keep projects on schedule and budget. This level of predictability is one that uses measurement to manage process as opposed to using measurement to evaluate individual performance. At this level, management can become coaches to help people address their day-to-day challenges in performing work processes in a predictable manner.

Management recognizes that obstacles and problems are normal in professional activities, and through early identification and resolution, professional work processes can be as predictable as manufacturing work processes.
1.4.1.5 Level 5 – Optimizing

At maturity level 5, an organization continually improves its processes based on a quantitative understanding of its business objectives and performance needs. The organization uses a quantitative approach to understand the variation inherent in the process and the causes of process outcomes.

1.4.1.6 Testers Need to Understand Process Maturity

Testers face a much greater challenge when testing software developed by maturity level 1 organizations, than they do when testing software developed by higher maturity levels. Some people have categorized level 1 organizations as “Test and Fix” organizations. At this level, testing and rework consumes more than 50% of the total software development effort. As software development processes mature, two things happen: more testing occurs earlier in the application life cycle and the aggregate amount of testing is reduced.

1.4.2 TMMi—Test Maturity Model integration

In section 1.4.1 the Capability Maturity Model Integration (CMMI) was described as a standard for software process improvement. One of the drawbacks in the CMMI is the limited amount of attention given to testing processes. In response to that, a process improvement standard for testing was developed. The Test Maturity Model integration (TMMi) is a detailed model for test process improvement and is positioned as being complementary to the CMMI. Like the CMMI, the TMMi was developed as a staged model. The staged model uses predefined sets of process areas to define an improvement path for an organization. This improvement path is described at each maturity level. Table 1-8 illustrates the five maturity levels of TMMi.

<table>
<thead>
<tr>
<th>Level</th>
<th>Focus</th>
<th>Process Areas</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 Optimizing</td>
<td>Continuous process improvement based on quantitative data</td>
<td>Defect Prevention, Test Process Optimization, Quality Control</td>
</tr>
<tr>
<td>4 Measured</td>
<td>Testing is a thoroughly defined, well-founded, and measurable process</td>
<td>Test Management, Software Quality Evaluation, Advanced Peer Reviews</td>
</tr>
<tr>
<td>3 Defined</td>
<td>Testing is fully integrated into the development lifecycle</td>
<td>Test Organization, Test Training Program, Test Lifestyle and Integration, Non-functional Testing, Peer Reviews</td>
</tr>
</tbody>
</table>
Table 1-8  TMMi’s Five Maturity Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Focus</th>
<th>Process Areas</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Testing becomes a managed process</td>
<td>Test Policy and Strategy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Test Planning</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Test Monitoring and Control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Test Design and Execution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Test Environment</td>
</tr>
<tr>
<td>1</td>
<td>Competent people and heroics</td>
<td></td>
</tr>
</tbody>
</table>

1.4.2.1  Level 1 – Initial

At TMMi level 1, testing is an undefined process and primarily carried out by the developer as part of the debugging process. At this level, many organizations simply do not understand or, in many cases, care about rigorous testing or the need for a dedicated testing effort. Like CMMI level 1 organizations, success depends on the competence and heroics of the people and not the use of proven processes. If there was a clearly stated objective for testing, it would simply be to get product out the door without major failures. When and if there is a testing function, a lack of resources, tools, and well educated staff is common. At TMMi level 1, there are no defined process areas with organizations tending to over commit and deliver products late, over budget and with lower than expected quality.

1.4.2.2  Level 2 – Managed

At TMMi level 2, testing becomes a managed process and is clearly separated from debugging. The process discipline reflected by maturity level 2 helps to ensure that existing practices are retained during times of stress. However, testing is still perceived by many stakeholders as being a project phase that follows coding.

At this level, a consistent company-wide or program-wide test strategy is established. Test plans are developed based on a product risk assessment and define what testing is required, when, how, and by whom. Risk management techniques are used to identify the product risks based on documented requirements. Testing is monitored and controlled to ensure it is going according to plan and actions can be taken if deviations occur. The status of the work products and the delivery of testing services are visible to management. Test design techniques are applied for deriving and selecting test cases from specifications. Testing is multi-leveled with component, integration, system and acceptance test levels. For each identified test level, there are specific testing objectives defined in the organization-wide or program-wide test strategy.

1.4.2.3  Level 3 – Defined

At TMMi level 3, life cycle testing is institutionalized. The organization’s set of standard test processes, which is the basis for maturity level 3, is established and improved over time. A
test organization and a specific test training program exist, and testing is perceived as being a profession. A critical distinction between TMMi maturity level 2 and 3 is the scope of the standards, process descriptions, and procedures. At maturity level 2, these may be quite different in each specific instance, e.g., on a particular project. At maturity level 3, these are tailored from the organization’s set of standard processes to suit a particular project or organizational unit and, therefore, are more consistent except for the differences allowed by the tailoring guidelines.

1.4.2.4 Level 4 - Measured

At this level in the TMMi, the goals of level 2 and 3 have been accomplished such that quantitative standards can be implemented. In TMMi level 4 organizations, testing is a thoroughly defined, well-founded, and measurable process. Testing is perceived as evaluation; it consists of all life cycle activities concerned with checking products and related work products.

Static testing in the form of reviews and inspections is considered part of the test process and is used to measure product quality early in the life cycle. Peer reviews as a defect detection technique are transformed into a product quality measurement technique.

1.4.2.5 Level 5 – Optimization

The maturing process that leads to level 5 has created an organizational infrastructure for testing that supports a completely defined and measured process. At TMMi maturity level 5, an organization is capable of continually improving its processes based on a quantitative understanding of statistically controlled processes. The testing methods and techniques are optimized and this is a continuous focus on fine tuning and process improvement. An optimized test process, as defined by the TMMi is one that is:

- managed, defined, measured, efficient, and effective
- statistically controlled and predictable
- focused on defect prevention
- supported by automation as much is deemed an effective use of resources
- able to support technology transfer from the industry to the organization
- able to support re-use of test assets
- focused on process change to achieve continuous improvement

1.4.3 ISO/IEC/IEEE 29119 Software Testing Standard

ISO 29119 is a set of standards for software testing that can be used within any software development life cycle or organization. There are currently five standards within the 29119 family.
1.4.3.1 ISO 29119–1: Concepts and Definitions

Part 1 of the ISO 29119 standard introduces the vocabulary on which all standards in the 29119 series are built and provides examples of the application of each concept in practice. Part 1 is informative and provides definitions, a description of the concepts of software testing, and ways to apply the processes, documents, and techniques defined in the 29119 series.

1.4.3.2 ISO 29119–2: Test Processes

Part 2 of the ISO 29119 defines a generic process model for software testing that can be used within any software development life cycle. The model specifies test processes that can be used to govern, manage, and implement software testing in any organization, project, or testing activity. The testing process is based on a three-layer process model that covers:

- Organizational Test Specifications (e.g. Organizational Test Policy, Organizational Test Strategy)
- Test Management
- Dynamic Testing

A risk-based approach to testing is used throughout the standard. Risk-based testing is a best-practice approach to strategizing and managing testing, as it allows testing to be prioritized and focused on the most important features and quality attributes of each system under test.

1.4.3.3 ISO 29119–3: Test Documentation

Part 3 defines templates for test documentation that cover the entire software testing life cycle. All templates align with the test process defined in ISO/IEC/IEEE 29119-2. The IEEE 829 Test Documentation standard was used as the basis for this standard, and as such, ISO/IEC/IEEE 29119-3 supersedes IEEE 829.

1.4.3.4 ISO 29119–4: Test Techniques

Part 4 covers software test design techniques. These techniques are independent of any specific software development life cycle model. The test design techniques that are presented in this standard can be used to derive test cases that can be used to collect evidence that the requirements of each system under test have been met and/or that defects are present. A risk-based approach to testing should be used to determine the set of techniques that are applicable in specific situations and to prioritize derived test cases and test procedures.

1.4.3.5 ISO 29119–5: Keyword Driven Testing

Part 5 of the standard defines a standard supporting Keyword-Driven Testing. Keyword-Driven Testing is a way of describing test cases by using a predefined set of Keywords. These Keywords are names which are associated with a set of actions that are required to perform a
specific step in a test case. By using keywords to describe test steps instead of natural language, test cases can be easier to understand, to maintain, and to automate. Keyword testing is explained in further detail in section 1.9.2.5.

1.5 Software Testing

Testing is the process of evaluating a deliverable with the intent of finding errors.

Testing is NOT:

- A stage/phase of the project
- Just finding broken code
- A final exam
- “Debugging”

1.5.1 Principles of Software Testing

Testing principles are important to test specialists/engineers because they provide the foundation for developing testing knowledge and acquiring testing skills. They also provide guidance for defining testing activities as performed in the practice of a test specialist.

A principle can be defined as:

1. A general or fundamental law, doctrine, or assumption
2. A rule or code of conduct
3. The laws or facts of nature underlying the working of an artificial device

A number of testing principles have been suggested over the past 40 years. These offer general guidelines common for all types of testing.

- **Testing shows presence of defects**
  The first principle states that testing can show that defects are present, but cannot prove that there are no defects. In other words, testing reduces the probability of undiscovered defects remaining in the software, but, even if no defects are found, it is not proof of correctness.

- **Exhaustive testing is impossible**
  Testing everything (all combinations of inputs and preconditions) is not feasible except for the most trivial cases. This implies that instead of spending scarce resources (both time and money) on exhaustive testing, organizations should use risk analysis and priorities to focus their testing efforts.

- **Early testing**
Testing activities should start as early as possible in the software or system development life cycle and should be focused on defined objectives.

- **Defect clustering**
  Research shows that a small number of modules generally contain most of the defects discovered during pre-release testing or are responsible for most of the operational failures. This indicates that software defects are usually found in clusters.

- **Pesticide paradox**
  If the same tests are repeated over and over again, their effectiveness reduces and eventually the same set of test cases will no longer find any new defects. This is called the “pesticide paradox.” To overcome this “pesticide paradox,” the test cases need to be regularly reviewed and revised, and new and different tests need to be written to exercise different parts of the software or system to potentially find more defects.

- **Testing is context dependent**
  No single test plan fits all organizations and all systems. Testing needs to be done differently in different contexts. For example, safety-critical software needs to be tested differently from an e-commerce site.

- **Absence-of-errors fallacy**
  Absence of errors does not mean that the software is perfect. Finding and fixing defects is of no help if the system build is unusable and does not fulfill the users’ needs and expectations.

- **Testing must be traceable to the requirements**
  Quality is understood as meeting customer requirements. One important principle for testing, therefore, is that it should be related to the requirements; testing needs to check that each requirement is met. You should, therefore, design tests for each requirement and should be able to trace back your test cases to the requirements being test—establish ‘requirement traceability.’ Testers should understand what is meant by, “all testing must be traceable to the overall user requirements.”

- **Testing needs to be planned for and conducted early in the software process**
  One common mistake made by IT organizations is to think of testing only after coding is complete. Testing must begin from the earliest stages of the application life cycle.

- **The data on the defects detected should be used to focus future testing efforts**
  Testing requires effort; therefore, it makes sense to focus this effort on areas that have more errors. An important principle of testing is that the testing resources should be used to uncover the largest possible number of errors.

- **Testing should be done incrementally**
  Software usually consists of a number of modules that interface with each other to provide the overall functionality. Some testers have the tendency to test software only after it is fully coded and integrated. The rationale for this is that if coding is done properly, there should be very few errors in it. Therefore, there is no need to waste time testing parts of the software separately. This approach is called “big-bang” testing. However, it is problematic because it is very difficult to isolate the sources of the errors encountered, as well as detect smaller errors, when the software is tested in
one shot. An important testing principle, therefore, is that testing should be done in incremental steps.

- **Testing should focus on exceptions**
  Testing aims at detecting as many errors as possible. To test effectively, we, therefore, need to take into account the human tendency of making mistakes. It has been found that while most programmers code correctly for typical processing, they make mistakes in code dealing with aberrant conditions, such as erroneous data entry or an unexpected data combination. Testing should, therefore, focus on such exceptions in the program so that these errors are detected.

### 1.5.2 Why Do We Test Software?

The simple answer as to why we test software is that the overall software development process is unable to build defect-free software. If the development processes were perfect, meaning no defects were produced, testing would not be necessary.

Let’s compare the manufacturing process of producing boxes of cereal to the process of making software. We find that, as is the case for most food manufacturing companies, testing each box of cereal produced is unnecessary. However, making software is a significantly different process than making a box of cereal. Cereal manufacturers may produce 50,000 identical boxes of cereal a day, while each software process is unique. This uniqueness introduces defects and thus making testing software necessary.

### 1.5.3 Developers are not Good Testers

Testing by the individual who developed the work has not proven successful in many organizations. The disadvantages of a person checking their own work are as follows:

- Misunderstandings will not be detected because the programmer will assume that what they heard was correct.
- Improper use of the development process may not be detected because the individual may not understand the process.
- The individual may be “blinded” into accepting erroneous system specifications and coding because he falls into the same trap during testing that led to the introduction of the defect in the first place.
- Software developers are optimistic in their ability to do defect-free work and thus sometimes underestimate the need for extensive testing.

Without a formal division between development and test, an individual may be tempted to improve the system structure and documentation, rather than allocate that time and effort to the test.
1.5.4 Factors Affecting Software Testing

Software testing varies from organization to organization. Many factors affect testing. The major factors are:

- People relationships
- Scope of testing
- Understanding the value of life cycle testing
- Poor test planning
- Testing constraints

Each of these factors will be discussed individually to explain how the role of testing in an IT organization is determined.

1.5.4.1 People Relationships

With the introduction of the Certified Software Test Engineer (CSTE) certification program in the early 1990’s, software testing began the long journey towards being recognized as a profession with specialized skill sets and qualifications. Over the last two decades, some organizations have come to appreciate that testing must be conducted throughout the application life cycle and that new development frameworks such as Agile must be considered if quality software products are to be delivered in today’s fast changing industry.

Unfortunately, while much progress has been made over the last 20+ years, many organizations still have unstructured testing processes and leave the bulk of software testing as the last activity in the development process.

The word “testing” conjures up a variety of meanings depending upon an individual’s frame of reference. Some people view testing as a method or process by which they add value to the development cycle; they may even enjoy the challenges and creativity of testing. Other people feel that testing tries a person’s patience, fairness, ambition, credibility, and capability. Testing can actually affect a person’s mental and emotional health if you consider the office politics and interpersonal conflicts that are often present.

Some attitudes that have shaped a negative view of testing and testers are:

- Testers hold up implementation, FALSE
- Giving testers less time to test will reduce the chance that they will find defects, FALSE
- Letting the testers find problems is an appropriate way to debug, FALSE
- Defects found in production are the fault of the testers, FALSE; and
- Testers do not need training; only programmers need training, FALSE!

Although testing is a process, it is very much a dynamic one in that the product and process will change somewhere with each application under test. There are several variables that affect the testing process, including the development process itself, software risk, customer/user participation, the testing process, the tester’s skill set, use of tools, testing budget and
resource constraints, management support, and morale and motivation of the testers. It is obvious that the people side of software testing has long been ignored for the more process-related issues of test planning, test tools, defect tracking, and so on.

Testers should perform a self-assessment to identify their own strengths and weaknesses as they relate to people-oriented skills. They should also learn how to improve the identified weaknesses, and build a master plan of action for future improvement. Essential testing skills include test planning, using test tools (automated and manual), executing tests, managing defects, risk analysis, test measurement, designing a test environment, and designing effective test cases. Additionally, a solid vocabulary of testing is essential. A tester needs to understand what to test, who performs what type of test, when testing should be performed, how to actually perform the test, and when to stop testing.

1.5.4.2 Scope of Testing

The scope of testing is the extensiveness of the test process. A narrow scope may be limited to determining whether or not the software specifications were correctly implemented. The scope broadens as more responsibilities are assigned to software testers. Among the broader scope of software testing are these responsibilities:

1. Finding defects early in the software development process, when they can be corrected at significantly less cost, than detecting them later in the software development process.

2. Removing defects of all types prior to the software going into production, when it is significantly cheaper, than when the software is operational

3. Identifying weaknesses in the software development process so that those processes can be improved and thus mature the software development process. Mature processes produce software more effectively and efficiently.

In defining the scope of software testing each IT organization must answer the question, “Why are we testing?”

1.5.4.3 Understanding the Value of Life Cycle Testing

The traditional view of the development life cycle places testing just prior to operation and maintenance, as illustrated in Table 1-9. All too often, testing after coding is the only method used to determine the adequacy of the system. When testing is constrained to a single phase and confined to the later stages of development, severe consequences can develop. It is not unusual to hear of testing consuming 50 percent of the project budget. All errors are costly, but the later in the life cycle that the discovered error is found, the more costly the error. An error discovered in the latter parts of the life cycle must be paid for four different times. The first cost is developing the program erroneously, which may include writing the wrong specifications, coding the system wrong, and documenting the system improperly. Second, the system must be tested to detect the error. Third, the wrong specifications and coding must be removed and the proper specifications, coding, and documentation added. Fourth, the system must be retested to determine that it is now correct.
If lower cost and higher quality systems are the goals of the IT organization, verification must not be isolated to a single phase in the development process but rather incorporated into each phase of development.

Studies have shown that the majority of system errors occur in the requirements and design phases. These studies show that approximately two-thirds of all detected system errors can be attributed to errors made prior to coding. This means that almost two-thirds of the errors are specified and coded into programs before they can be detected by validation (dynamic testing).

<table>
<thead>
<tr>
<th>Life Cycle Phase</th>
<th>Testing Activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>- Determine verification approach</td>
</tr>
<tr>
<td></td>
<td>- Determine adequacy of requirements</td>
</tr>
<tr>
<td></td>
<td>- Develop Test Plan</td>
</tr>
<tr>
<td></td>
<td>- Generate functional test cases/data based on requirements</td>
</tr>
<tr>
<td>Design</td>
<td>- Determine consistency of design with requirements</td>
</tr>
<tr>
<td></td>
<td>- Determine adequacy of design</td>
</tr>
<tr>
<td></td>
<td>- Generate structural and functional test cases/data based on design</td>
</tr>
<tr>
<td></td>
<td>- Refine test cases written in the requirements phase</td>
</tr>
<tr>
<td>Program (build/construction)</td>
<td>- Determine consistency of code with the design</td>
</tr>
<tr>
<td></td>
<td>- Determine adequacy of implementation</td>
</tr>
<tr>
<td></td>
<td>- Generate and refine structural and functional test cases/data</td>
</tr>
<tr>
<td>Test</td>
<td>- Test application system</td>
</tr>
<tr>
<td>Installation</td>
<td>- Place tested system into production</td>
</tr>
<tr>
<td>Maintenance</td>
<td>- Modify and retest</td>
</tr>
</tbody>
</table>

Table 1-9 Life Cycle Testing Activities

The recommended testing process is presented in Table 1-9 as a life cycle chart showing the verification activities for each phase. The success of conducting verification throughout the development cycle depends upon the existence of clearly defined and stated products to be produced at each development stage. The more formal and precise the statement of the development product, the more amenable it is to the analysis required to support verification. A more detailed discussion of Life Cycle Testing is found later in this skill category.

1.5.4.4 Poor Test Planning

Variability in test planning is a major factor affecting software testing today. A plan should be developed that defines how testing should be performed (see Skill Category 4). With a test
plan, testing can be considered complete when the plan has been accomplished. The test plan is a contract between the software stakeholders and the testers.

### 1.5.4.5 Testing Constraints

Anything that inhibits the tester’s ability to fulfill their responsibilities is a constraint. Constraints include:

- Limited schedule and budget
- Lacking or poorly written requirements
- Limited tester skills
- Lack of independence of the test team

Each of these four constraints will be discussed individually.

#### 1.5.4.5.1 Budget and Schedule Constraints

Budget and schedule constraints may limit the ability of a tester to complete their test plan. Embracing a life cycle testing approach can help alleviate budget and schedule problems.

The cost of defect identification and correction increases exponentially as the project progresses. Figure 1-11 illustrates how costs dramatically increase the later in the life cycle you find a defect. A defect discovered during requirement and design is the cheapest to fix. So, let’s say it costs x; based on this, a defect corrected during the system test phase costs 10x to fix. A defect corrected after the system goes into production costs 100x. Clearly, identifying and correcting defects early is the most cost-effective way to reduce the number of production level defects.

![Figure 1-11 Relative Cost versus the Project Phase](image)

*Figure 1-11 Relative Cost versus the Project Phase*
Testing should begin during the first phase of the life cycle and continue throughout the life cycle. It’s important to recognize that life cycle testing is essential to reducing the overall cost of producing software.

Let’s look at the economics of testing. One information services manager described testing in the following manner, “too little testing is a crime – too much testing is a sin.” The risk of under testing is directly translated into system defects present in the production environment. The risk of over testing is the unnecessary use of valuable resources in testing computer systems where the cost of testing far exceeds the value of detecting the defects.

Most problems associated with testing occur from one of the following causes:

- Failing to define testing objectives
- Testing at the wrong phase in the life cycle
- Using ineffective test techniques

The cost-effectiveness of testing is illustrated in Figure 1-12. As the cost of testing increases, the number of undetected defects decreases. The left side of the illustration represents an under test situation in which the cost of testing is less than the resultant loss from undetected defects.

![Figure 1-12  Testing Cost Curve](image)

At some point, the two lines cross and an over test condition begins. In this situation, the cost of testing to uncover defects exceeds the losses from those defects. A cost-effective perspective means testing until the optimum point is reached, which is the point where the value received from testing no longer exceeds the cost of testing.

Few organizations have established a basis to measure the effectiveness of testing. This makes it difficult to determine the cost effectiveness of testing. Without testing standards, the effectiveness of the process cannot be evaluated in sufficient detail to enable the process to be measured and improved.
The use of a standardized testing methodology provides the opportunity for a cause and effect relationship to be determined and applied to the methodology. In other words, the effect of a change in the methodology can be evaluated to determine whether that effect resulted in a smaller or larger number of defects being discovered. The establishment of this relationship is an essential step in improving the test process. The cost-effectiveness of a testing process can be determined when the effect of that process can be measured. When the process can be measured, it can be adjusted to improve its cost-effectiveness for the organization.

1.5.4.5.2 Lack Of or Poorly Written Requirements

If requirements are lacking or poorly written, then the test team must have a defined method for uncovering and defining test objectives.

A test objective is simply a testing “goal.” It is a statement of what the test team or tester is expected to accomplish or validate during a specific testing activity. Test objectives, usually defined by the test manager or test team leader during requirements analysis, guide the development of test cases, test scripts, and test data. Test objectives enable the test manager and project manager to gauge testing progress and success, and enhance communication both within and outside the project team by defining the scope of the testing effort.

Each test objective should contain a statement of the objective, and a high-level description of the expected results stated in measurable terms. The users and project team must prioritize the test objectives. Usually the highest priority is assigned to objectives that validate high priority or high-risk requirements defined for the project. In cases where test time is short, test cases supporting the highest priority objectives would be executed first.

Test objectives can be easily derived from using the system requirements documentation, the test strategy, and the outcome of the risk assessment. A couple of techniques for uncovering and defining test objectives, if the requirements are poorly written, are brainstorming and relating test objectives to the system inputs, events, or system outputs. Ideally, there should be less than 100 high level test objectives for all but the very largest systems. Test objectives are not simply a restatement of the system’s requirements, but the actual way the system will be tested to assure that the system objective has been met. Completion criteria define the success measure for the tests.

As a final step, the test team should perform quality control on the test objective process using a checklist or worksheet to ensure that the process to set test objectives was followed, or reviewing them with the system users.

1.5.4.5.3 Limited Tester Skills

Testers should be competent in all skill areas defined in the Software Testing Body of Knowledge (STBOK). Lack of the skills needed for a specific test assignment constrains the ability of the testers to effectively complete that assignment. Tester skills will be discussed in greater detail in Skill Category 2.
1.5.5 Independent Testing

The primary responsibility of individuals accountable for testing activities is to ensure that quality is measured accurately. Often, just knowing that the organization is measuring quality is enough to cause improvements in the applications being developed. In the loosest definition of independence, just having a tester or someone in the organization devoted to test activities is a form of independence.

The roles and reporting structure of test resources differs across and within organizations. These resources may be business or systems analysts assigned to perform testing activities, or they may be testers who report to the project manager. Ideally, the test resources will have a reporting structure independent from the group designing or developing the application in order to assure that the quality of the application is given as much consideration as the development budget and timeline.

Misconceptions abound regarding the skill set required to perform testing, including:

- Testing is easy
- Anyone can perform testing
- No training or prior experience is necessary

In truth, to test effectively, an individual must:

- Thoroughly understand the system
- Thoroughly understand the technology the system is being deployed upon (e.g., client/server, Internet technologies, or mobile introduce their own challenges)
- Possess creativity, insight, and business knowledge
- Understand the development methodology used and the resulting artifacts

While much of this discussion focuses on the roles and responsibilities of an independent test team, it is important to note that the benefits of independent testing can be seen in the unit testing stage. Often, successful development teams will have a peer perform the unit testing on a program or class. Once a portion of the application is ready for integration testing, the same benefits can be achieved by having an independent person plan and coordinate the integration testing.

Where an independent test team exists, they are usually responsible for system testing, the oversight of acceptance testing, and providing an unbiased assessment of the quality of an application. The team may also support or participate in other phases of testing as well as executing special test types such as performance and load testing.

An independent test team is usually comprised of a test manager or team leader and a team of testers. The test manager should join the team no later than the start of the requirements definition stage. Key testers may also join the team at this stage on large projects to assist with test planning activities. Other testers can join later to assist with the creation of test cases and scripts, and right before system testing is scheduled to begin.
The test manager ensures that testing is performed, that it is documented, and that testing techniques are established and developed. They are responsible for ensuring that tests are designed and executed in a timely and productive manner, as well as:

- Test planning and estimation
- Designing the test strategy
- Reviewing analysis and design artifacts
- Chairing the Test Readiness Review
- Managing the test effort
- Overseeing acceptance tests

Testers are usually responsible for:

- Developing test cases and procedures
- Test data planning, capture, and conditioning
- Reviewing analysis and design artifacts
- Testing execution
- Utilizing automated test tools for regression testing
- Preparing test documentation
- Defect tracking and reporting

Other testers joining the team will primarily focus on test execution, defect reporting, and regression testing. These testers may be junior members of the test team, users, marketing or product representatives.

The test team should be represented in all key requirements and design meetings including:

- JAD or requirements definition sessions
- Risk analysis sessions
- Prototype review sessions

They should also participate in all inspections or walkthroughs for requirements and design artifacts.
1.6 Software Development Life Cycle (SDLC) Models

Software Development Life Cycle models describe how the software development phases combine together to form a complete project. The SDLC describes a process used to create a software product from its initial conception to its release. Each model has its advantages and disadvantages, so certain models may be employed depending on the goal of the application project. Testers will work on many different projects using different development models. A common misconception is that projects must follow only one methodology; for instance, if a project uses a Waterfall approach, it is only Waterfall. This is wrong. The complexities of projects and the variety of interconnected modules will often be developed using best practices found in a variety of models. The tester will need to tailor the best testing approach to fit the model or models being used for the current project. There are many software development models. These models are:

- Ad-Hoc
- Waterfall
- V-Model
- Incremental Model
- Iterative Development Model
- Prototype/RAD Model
- Spiral Model
- Reuse Model

1.6.1 Typical Tasks in the Development Life Cycle

Professional system developers, testers, and the customers they serve share a common goal of building information systems that effectively support business objectives. In order to ensure that cost-effective, quality systems are developed which address an organization’s business needs, the development team employs some kind of system development model to direct the project’s life cycle. Typical activities performed include the following:4

- System conceptualization
- System requirements and benefits analysis
- Project adoption and project scoping
- System design
- Specification of software requirements

---

• Architectural design
• Detailed design
• Unit development
• Unit testing
• System integration & testing
• Installation at site
• Site testing and acceptance
• Training and documentation
• Implementation
• Maintenance

1.6.2 Model Variations

While nearly all system development efforts engage in some combination of the above tasks, they can be differentiated by the feedback and control methods employed during development and the timing of activities. Most system development models in use today have evolved from three primary approaches: Ad-hoc Development, Waterfall Model, and the Iterative process.

1.6.3 Ad-hoc Development

Early systems development often took place in a rather chaotic and haphazard manner, relying entirely on the skills and experience of the individual staff members performing the work. Today, many organizations still practice Ad-hoc Development either entirely or for a certain subset of their development (e.g. small projects).

The Software Engineering Institute (SEI) at Carnegie Mellon University\(^5\) points out that with Ad-hoc Process Models, “process capability is unpredictable because the software process is constantly changed or modified as the work progresses. Schedules, budgets, functionality, and product quality are generally (inconsistent). Performance depends on the capabilities of individuals and varies with their innate skills, knowledge, and motivations. There are few stable software processes in evidence, and performance can be predicted only by individual rather than organizational capability.”\(^6\)

---

5. Information on the Software Engineering Institute can be found at http://www.sei.cmu.edu.

“Even in undisciplined organizations, however, some individual software projects produce excellent results. When such projects succeed, it is generally through the heroic efforts of a dedicated team, rather than through repeating the proven methods of an organization with a mature software process. In the absence of an organization-wide software process, repeating results depends entirely on having the same individuals available for the next project. Success that rests solely on the availability of specific individuals provides no basis for long-term productivity and quality improvement throughout an organization.”\textsuperscript{7}

\section*{1.6.4 The Waterfall Model}

The Waterfall Model is the earliest method of structured system development. Although it has come under attack in recent years for being too rigid and unrealistic when it comes to quickly meeting customer’s needs, the Waterfall Model is still widely used. It is attributed with providing the theoretical basis for other SDLC models because it most closely resembles a “generic” model for software development.

\textsuperscript{7} Ibid.
The Waterfall Model consists of the following steps:

- **System Conceptualization** – Refers to the consideration of all aspects of the targeted business function or process, with the goal of determining how each of those aspects relates with one another, and which aspects will be incorporated into the system.

- **Systems Analysis** – Refers to the gathering of system requirements, with the goal of determining how these requirements will be accommodated in the system. Extensive communication between the customer and the developer is essential.

- **System Design** – Once the requirements have been collected and analyzed, it is necessary to identify in detail how the system will be constructed to perform necessary tasks. More specifically, the System Design phase is focused on the data requirements (what information will be processed in the system?), the software construction (how will the application be constructed?), and the interface construction (what will the system look like? What standards will be followed?).

- **Coding** – Also known as programming, this step involves the creation of the system software. Requirements and systems specifications from the System Design step are translated into machine readable computer code.

- **Testing** – As the software is created and added to the developing system, testing is performed to ensure that it is working correctly and efficiently. Testing is generally focused on two areas: internal efficiency and external effectiveness. The goal of external effectiveness testing is to verify that the software is functioning according to system design, and that it is performing all necessary functions or sub-functions. The goal of internal testing is to make sure that the computer code is efficient, standardized, and well documented. Testing can be a labor-intensive process, due to its iterative nature.

---

There are a variety of potential deliverables from each life cycle phase. The primary deliverables for each Waterfall phase are shown in Table 1-10 along with “What is tested” and “Who performs the testing.”

<table>
<thead>
<tr>
<th>Development Phase</th>
<th>Deliverable</th>
<th>What is Tested</th>
<th>Who Performs Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Conceptualization</td>
<td>Statement of need</td>
<td>Feasibility of System</td>
<td>Business Analysts, Product Owner, Testers</td>
</tr>
<tr>
<td>Systems Analysis</td>
<td>Statement of user requirements</td>
<td>Completeness and accuracy of requirements in describing user need</td>
<td>Business Analysts, Product Owner, Testers</td>
</tr>
<tr>
<td>System Design</td>
<td>System design specification</td>
<td>Completeness and accuracy of translation of requirements into design</td>
<td>Design Analysts, DBA, Developers, Testers</td>
</tr>
<tr>
<td>Coding</td>
<td>Application software</td>
<td>Design specifications translated into code at module level</td>
<td>Developers</td>
</tr>
<tr>
<td>Dynamic Testing</td>
<td>Tested system, error reports, final test report</td>
<td>Requirements, design specifications, applications software</td>
<td>Developers, Testers, Users</td>
</tr>
<tr>
<td>Maintenance</td>
<td>System changes</td>
<td>Requirements, design specifications, application software</td>
<td>Developers, Testers, Users</td>
</tr>
</tbody>
</table>

Table 1-10  System Deliverables Tested in the Traditional Waterfall Model

1.6.4.1  Problems/Challenges associated with the Waterfall Model

Although the Waterfall Model has been used extensively over the years in the production of many quality systems, it is not without its problems. Criticisms fall into the following categories:

- Real projects rarely follow the sequential flow that the model proposes.
- At the beginning of most projects there is often a great deal of uncertainty about requirements and goals, and it is therefore difficult for customers to identify these criteria on a detailed level. The model does not accommodate this natural uncertainty very well.
- Developing a system using the Waterfall Model can be a long, painstaking process that does not yield a working version of the system until late in the process.
1.6.5 V-Model

The V-Model is considered an extension of the Waterfall Model. The purpose of the “V” shape is to demonstrate the relationships between each phase of specification development and its associated dynamic testing phase. This model clearly shows the inverse relationship between how products move from high level concepts to detailed program code; then, dynamic testing begins at the detailed code phase and progresses to the high level user acceptance test phase.

On the left side of the “V,” often referred to as the specifications side, verification test techniques are employed (to be described later in this skill category). These verification tests test the interim deliverables and detect defects as close to point of origin as possible. On the right hand side of the “V,” often referred to as the testing side, validation test techniques are used (described later in this skill category). Each of the validation phases test the counter opposed specification phase to validate that the specification at that level has been rendered into quality executable code.

![The V-Model](image)

*Figure 1-15 The V-Model*

The V-Model enables teams to significantly increase the number of defects identified and removed during the development life cycle by integrating verification tests into all stages of development. Test planning activities are started early in the project, and test plans are detailed in parallel with requirements. Various verification techniques are also utilized throughout the project to:

- Verify evolving work products
- Test evolving applications by walking through scenarios using early prototypes
- Removing defects in the stage of origin (phase containment) results in:
  - Shorter time to market
  - Lower error correction costs
  - Fewer defects in the production system
- Early test planning yields better test plans that can be used to validate the application against requirements

Regardless of the development methodology used, understanding the V-model helps the tester recognize the dependence of related phases within the life cycle.

### 1.6.6 Incremental Model

The incremental method is in many ways a superset of the Waterfall Model. Projects following the Incremental approach subdivide the requirements specifications into smaller buildable projects (or modules). Within each of those smaller requirements subsets, a development life cycle exists which includes the phases described in the Waterfall approach. The goal is to produce a working portion of the application demonstrating real functionality early in the broader SDLC. Each subsequent “increment” adds additional functionality to the application. Successive rounds continue until the final product is produced. Several of the development models are variants of the Incremental model including Spiral and RAD.

![Figure 1-16 The Incremental Model](image)

### 1.6.7 Iterative Development

The problems with the Waterfall Model and its variants created a demand for a new method of developing systems which could provide faster results, require less up-front information, and offer greater flexibility. With Iterative Development, the project is divided into small parts.
This allows the development team to demonstrate results earlier on in the process and obtain valuable feedback from system users. Often, each iteration is actually a mini-Waterfall process with the feedback from one phase providing vital information for the design of the next phase. In a variation of this model, the software products which are produced at the end of each step (or series of steps) can go into production immediately as incremental releases.

![Iterative Development Diagram](image)

**Figure 1-17 Iterative Development**

1.6.7.1 Problems/Challenges associated with the Iterative Model\(^9\)

While the Iterative Model addresses many of the problems associated with the Waterfall Model, it does present new challenges.

The user community needs to be actively involved throughout the project. Even though this involvement is a positive for the project, it is demanding on the time of the staff and can cause project delays.

Communication and coordination skills take center stage in project development.

Informal requests for improvement after each phase may lead to confusion -- a controlled mechanism for handling substantive requests needs to be developed.

---

The Iterative Model can lead to “scope creep,” since user feedback following each phase may lead to increased customer demands. As users see the system develop, they may realize the potential of other system capabilities which would enhance their work.

### 1.6.8 Variations on Iterative Development

A number of SDLC models have evolved from the Iterative approach. All of these methods produce some demonstrable software product early on in the process in order to obtain valuable feedback from system users or other members of the project team. Several of these methods are described below.

#### 1.6.8.1 Prototyping

The Prototyping Model was developed on the assumption that it is often difficult to know all of your requirements at the beginning of a project. Typically, users know many of the objectives that they wish to address with a system, but they do not know all the nuances of the data, nor do they know the details of the system features and capabilities. The Prototyping Model allows for these circumstances and offers a development approach that yields results without first requiring all the information.

When using the Prototyping Model, the developer builds a simplified version of the proposed system and presents it to the customer for consideration as part of the development process. The customer in turn provides feedback to the developer, who goes back to refine the system requirements to incorporate the additional information. Often, the prototype code is thrown away and entirely new programs are developed once requirements are identified.

There are a few different approaches that may be followed when using the Prototyping Model:

- Creation of the major user interfaces without any substantive coding in the background in order to give the users a “feel” for what the system will look like
- Development of an abbreviated version of the system that performs a limited subset of functions; development of a paper system (depicting proposed screens, reports, relationships etc.)
- Use of an existing system or system components to demonstrate some functions that will be included in the developed system

#### 1.6.8.2 Prototyping steps

Prototyping is comprised of the following steps:

---

• **Requirements Definition/Collection** – Similar to the Conceptualization phase of the Waterfall Model, but not as comprehensive. The information collected is usually limited to a subset of the complete system requirements.

• **Design** – Once the initial layer of requirements information is collected, or new information is gathered, it is rapidly integrated into a new or existing design so that it may be folded into the prototype.

• **Prototype Creation/Modification** – The information from the design is rapidly rolled into a prototype. This may mean the creation/modification of paper information, new coding, or modifications to existing coding.

• **Assessment** – The prototype is presented to the customer for review. Comments and suggestions are collected from the customer.

• **Prototype Refinement** – Information collected from the customer is digested and the prototype is refined. The developer revises the prototype to make it more effective and efficient. **Additional iterations may be done as necessary.**

• **System Implementation** – In most cases, the system is rewritten once requirements are understood. Sometimes, the Iterative process eventually produces a working system that can be the cornerstone for the fully functional system.

### 1.6.8.3 Problems/Challenges associated with the Prototyping Model

Criticisms of the Prototyping Model generally fall into the following categories:

• Prototyping can lead to false expectations. Prototyping often creates a situation where the customer mistakenly believes that the system is “finished” when in fact it is not. More specifically, when using the Prototyping Model, the pre-implementation versions of a system are really nothing more than one-dimensional structures. The necessary, behind-the-scenes work such as database normalization, documentation, testing, and reviews for efficiency have not been done. Thus the necessary underpinnings for the system are not in place.

• Prototyping can lead to poorly designed systems. Because the primary goal of Prototyping is rapid development, the design of the system can sometimes suffer because the system is built in a series of “layers” without a global consideration of the integration of all other components. While initial software development is often built to be a “throwaway,” attempting to retroactively produce a solid system design can sometimes be problematic.

### 1.6.8.4 Rapid Application Development (RAD)

Rapid Application Development (RAD), a variant of prototyping, is another form of iterative development. The RAD model is designed to build and deliver application prototypes to the client while in the iterative process. With less emphasis placed on detailed requirements upfront, the user continuously interacts with the development team during the user design phase. The process is continuous and interactive which allows the user to understand, modify, and eventually approve a working model. The approved model moves to the construction
phase where the user still continues to participate. At this time, traditional phases of coding, unit, integrations and system test take place. The four phases of RAD are:

1. Requirements Planning phase
2. User Design phase
3. Construction phase
4. Cutover phase

![Rapid Application Development Model](image)

**Figure 1-18 Rapid Application Development Model**

### 1.6.9 The Spiral Model

The Spiral Model was designed to include the best features from the Waterfall and Prototyping Models, and introduces a new component – risk-assessment. The term “spiral” is used to describe the process that is followed as the development of the system takes place. Similar to the Prototyping Model, an initial version of the system is developed, and then repeatedly modified based on input received from customer evaluations. Unlike the Prototyping Model, however, the development of each version of the system is carefully designed using the steps involved in the Waterfall Model. With each iteration around the spiral (beginning at the center and working outward), progressively more complete versions of the system are built.

---

Risk assessment is included as a step in the development process as a means of evaluating each version of the system to determine whether or not development should continue. If the customer decides that any identified risks are too great, the project may be halted. For example, if a substantial increase in cost or project completion time is identified during one phase of risk assessment, the customer or the developer may decide that it does not make sense to continue with the project, since the increased cost or lengthened time frame may make continuation of the project impractical or unfeasible.

1.6.9.1 The Spiral Model steps

The Spiral Model is made up of the following steps:

- **Project Objectives** – Similar to the system conception phase of the Waterfall Model. Objectives are determined, possible obstacles are identified and alternative approaches are weighed.

- **Risk Assessment** – Possible alternatives are examined by the developer, and associated risks/problems are identified. Resolutions of the risks are evaluated and weighed in the consideration of project continuation. Sometimes prototyping is used to clarify needs.

- **Engineering & Production** – Detailed requirements are determined and the software piece is developed.

---

• **Planning and Management** – The customer is given an opportunity to analyze the results of the version created in the Engineering step and to offer feedback to the developer.

**1.6.9.2 Problems/Challenges associated with the Spiral Model**

The risk assessment component of the Spiral Model provides both developers and customers with a measuring tool that earlier Process Models did not have. The measurement of risk is a feature that occurs every day in real-life situations, but (unfortunately) not as often in the system development industry. The practical nature of this tool helps to make the Spiral Model a more realistic Process Model than some of its predecessors.

**1.6.10 The Reuse Model**

The premise behind the Reuse Model is that systems should be built using existing components, as opposed to custom-building new components. The Reuse Model is clearly suited to Object-Oriented computing environments, which have become one of the premiere technologies in today’s system development industry.

Within the Reuse Model, libraries of software modules are maintained that can be copied for use in any system. These components are of two types: procedural modules and database modules.

When building a new system, the developer will “borrow” a copy of a module from the system library and then plug it into a function or procedure. If the needed module is not available, the developer will build it, and store a copy in the system library for future usage. If the modules are well engineered, the developer, with minimal changes, can implement them.

**1.6.10.1 The Reuse Model steps**

The Reuse Model consists of the following steps:

- **Definition of Requirements** – Initial system requirements are collected. These requirements are usually a subset of the complete system requirements.

- **Definition of Objects** – The objects, which can support the necessary system components, are identified.

- **Collection of Objects** – The system libraries are scanned to determine whether or not the needed objects are available. Copies of the needed objects are downloaded from the system.

- **Creation of Customized Objects** – Objects that have been identified as needed, but that are not available in the library are created.

- **Prototype Assembly** – A prototype version of the system is created and/or modified using the necessary objects.
• **Prototype Evaluation** – The prototype is evaluated to determine if it adequately addresses customer needs and requirements.

• **Requirements Refinement** – Requirements are further refined as a more detailed version of the prototype is created.

• **Objects Refinement** – Objects are refined to reflect the changes in the requirements.

### 1.6.10.2 Problems/Challenges Associated with the Reuse Model

A general criticism of the Reuse Model is that it is limited for use in object-oriented development environments. Although this environment is rapidly growing in popularity, it is currently used in only a minority of system development applications.

### 1.6.11 Creating and Combining Models

In many cases, parts and procedures from various SDLC models are integrated to support system development. This occurs because most models were designed to provide a framework for achieving success only under a certain set of circumstances. When the circumstances change beyond the limits of the model, the results from using it are no longer predictable. When this situation occurs it is sometimes necessary to alter the existing model to accommodate the change in circumstances, or adopt or combine different models to accommodate the new circumstances.

The selection of an appropriate model hinges primarily on two factors: organizational environment and the nature of the application. Frank Land, from the London School of Economics, suggests that suitable approaches to system analysis, design, development, and implementation be based on the relationship between the information system and its organizational environment. Four categories of relationships are identified:

• **The Unchanging Environment** – Information requirements are unchanging for the lifetime of the system (e.g. those depending on scientific algorithms). Requirements can be stated unambiguously and comprehensively. A high degree of accuracy is essential. In this environment, formal methods (such as the Waterfall or Spiral Models) would provide the completeness and precision required by the system.

• **The Turbulent Environment** – The organization is undergoing constant change and system requirements are always changing. A system developed on the basis of the conventional Waterfall Model would be, in part, already obsolete by the time it is implemented. Many business systems fall into this category. Successful methods would include those which incorporate rapid development, some throwaway code (such as in Prototyping), the maximum use of reusable code, and a highly modular design.

• **The Uncertain Environment** – The requirements of the system are unknown or uncertain. It is not possible to define requirements accurately ahead of time because the situation is new or the system being employed is highly innovative. Here, the
development methods must emphasize learning. Experimental Process Models, which
take advantage of prototyping and rapid development, are most appropriate.

- **The Adaptive Environment** – The environment may change in reaction to the system
  being developed, thus initiating a changed set of requirements. Teaching systems and
  expert systems fall into this category. For these systems, adaptation is key, and the
  methodology must allow for a straightforward introduction of new rules.

### 1.6.12 SDLC Models Summary

The evolution of system development models has reflected the changing needs of computer
customers. As customers demanded faster results, more involvement in the development
process, and the inclusion of measures to determine risks and effectiveness, the methods for
developing systems evolved. In addition, the software and hardware tools used in the industry
changed (and continue to change) substantially. Faster networks and hardware supported the
use of smarter and faster operating systems that paved the way for new languages and
databases, and applications that were far more powerful than any predecessors. These and
numerous other changes in the system development environment simultaneously spawned the
development of more practical new models and the demise of older models that were no
longer useful.

### 1.6.13 Application Lifecycle Management (ALM)

Application lifecycle management (ALM) is often discussed as if it is another software
development lifecycle framework. In reality, ALM is quite different from SDLC. The ALM is
a superset which could include one or more SDLCs. ALM is about managing the entire
application lifecycle from the initial application definition, through the development lifecycle,
to application maintenance, and eventually application retirement.

### 1.7 Agile Development Methodologies

As the preceding discussion makes clear, there are strengths and weaknesses associated with
all of the methodologies. In the mid 1990s a number of alternative development solutions
appeared to address some of the perceived shortcomings, especially the lack of flexibility.
These approaches, which include Scrum, Crystal Clear, Adaptive Software Development,
Feature Driven Development, Dynamic Systems Development Methodology (DSDM) and
probably the best known Extreme Programming (XP), collectively have come to be referred to
as Agile Methodologies.
1.7.1 Basic Agile Concepts

As the various methodologies emerged, there were similarities and differences between them. In an effort to bring some cohesion and “critical mass” to the Agile movement, there were a number of conferences and workshops. In 2001, a number of the key figures in the Agile movement met in an effort to define a lighter, faster way of creating software which was less structural and more people focused. The result was a document which has become known as the Agile Manifesto; it articulates the key principles of the Agile Development Framework.

Principles behind the Agile Manifesto

*We follow these principles:*

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity—the art of maximizing the amount of work not done—is essential.

The best architectures, requirements, and designs emerge from self-organizing teams.

14. Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas
At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

### 1.7.2 Agile Practices

Despite the reputation for being undisciplined, effective Agile approaches are anything but that. The structure and the flow is very different from traditional approaches, and the way products are derived is very different, even if the name is the same.

- **Detailed Planning** – is done for each iteration, this may be called a “timebox” or a “sprint” or merely a cycle or iteration. Requirements are gathered from the customer as “stories” of desired capabilities. In a bullpen style discussion they are analyzed in depth to understand what is needed and what will be required to provide that functionality. This process combines both a verification step for Requirements and a group high level design.

  Design is kept as simple as possible while still achieving the desired functionality. Future potential use is not considered. While the concept of a product architecture may exist, it does not drive the inclusion of functionality not requested by the customer or essential to meet an immediate customer need.

  Work to be done is carefully estimated using a standard unit of measure (often called points). The amount of work, measured in units, that can be completed within a given amount of time (cycle, iteration, etc.) is tracked over time, and is known as velocity. Once established, velocity is relatively fixed and it determines how much functionality can be delivered per cycle.

- **Test Driven Development** – To ensure that Requirements are fully understood, the test cases are developed and run before the code is written. This process helps to identify things that will need to be changed for the new functionality to work properly.

- **Refactor Relentlessly** – Refactoring is the term for changing existing code to work properly with the new requirements. This part of the process is one of the most contentious for those accustomed to traditionally architected systems which strive to “pre-plan” all possible interfaces and accesses. Failure to effectively and aggressively refactor will result in a steady increase in the testing effort combined with a significant decline in productivity.

- **Continuous Integration** – As individual units of work are completed, they are added to the existing base and integration tested. New test cases developed for the specific functionality are installed and become a part of the future test base for all other developers. Updates which fail must be removed and repaired, along with the associated test cases so that the work of others will not be jeopardized. Typical

---

15. 2001, Per the above authors this declaration may be freely copied in any form, but only in its entirety through this notice.
development cycles lasting from one to three weeks will often see twice daily integration activity.

- **Paired Programming** – To obtain the benefits of reviews and inspections, as well as to facilitate the dispersion of the knowledge base, programmers work in pairs. The pairs may change partners often to promote the collective ownership of code. This is also supported by the strict adherence to a set of uniform coding standards that makes code understandable and modifiable by any experienced team member.

- **Onsite Customer** – One of the major issues addressed by the Agile approach is the need for continuous involvement on the part of the intended product user or a designated representative. This individual is a part of the team and co-located, making access a matter of walking across the room. The onsite requirement prevents delays and confusion resulting from the inability to access the customer or business partner at critical times in the development process. It also addresses the testing and requirements issue.

### 1.7.3 Effective Application of Agile Approaches

The Agile approach works well in projects where it is difficult to obtain solid requirements due to an unstable environment especially those in which the requirements will continue to emerge as the product is used. For organizations that see themselves as “nimble” or “responsive” or “market-driven” and that view the necessary refactoring as an acceptable price to pay for being quick to market, Agile works well.

Agile development teams are generally small; Kent Beck\(^{16}\) suggests 10 or fewer. Projects requiring more people should be broken down into teams of the recommended size. Beck’s suggestion has led people to think that Agile only works well for small projects, and it often excels in this area. Many organizations are experimenting with the use of Agile on larger projects; in fact, Beck’s original project was very large.

One of the key “efficiencies” achieved through the use of Agile methodologies is the elimination of much of the documentation created by the traditional processes. The intent is for programs to be self documenting with extensive use of commentary.

This lack of documentation is one of the drawbacks for many organizations considering Agile, especially those which impact the health and safety of others. Large, publicly traded corporations involved in international commerce are finding the lack of external documentation can cause problems when complying with various international laws that require explicit documentation of controls on financial systems.

Agile development is less attractive in organizations that are highly structured with a “command and control” orientation. There is less incentive and less reward for making the organizational and cultural changes required for Agile when the environment exists for developing a stable requirements base.

\(^{16}\) Beck, Kent; *Extreme Programming Explained*
1.7.4 Integrating Agile with Traditional Methodologies

As the challenges and benefits of employing Agile methodologies become more widely understood and accepted, there is a move toward selective integration. Organizations are targeting projects with a positive Agile benefit profile and applying that methodology, even while maintaining a substantial portfolio of traditional waterfall or iterative projects.

This approach allows the organization to respond rapidly to a crisis or opportunity by quickly deploying an entry level product and then ramping up the functionality in a series of iterations. Once the initial result has been achieved, it is possible to either continue with the Agile development, or consider the production product as a “super-prototype” that can either be expanded or replaced.

1.8 Testing Throughout the Software Development Life Cycle (SDLC)

Life cycle testing involves continuous testing of the system throughout the development process. Full life testing incorporates both verification tests and validation tests. Verification and validation will be discussed in length in the following sections. Life cycle testing cannot occur until a formalized life cycle approach has been adopted. Life cycle testing is dependent upon the completion of predetermined deliverables at specified points in the development life cycle. If significant variability exists in the development processes, it is very difficult to effectively test both executable and non-executable deliverables.

The “V-model” as discussed in section 1.6.5 is ideal for describing both verification and validation test processes in the SDLC. Regardless of the development life cycle model used, the basic need for verification and validation tests remain.
1.8.1 Static versus Dynamic Testing

Two terms that tend to be used when describing life cycle testing processes are static and dynamic testing. Static testing is performed on non-executable deliverables in the SDLC. Static testing is designed to both identify product defects and process defects. From the product perspective, examples of static software testing includes code analysis, code reviews and requirements walkthroughs. The code is not executed during static testing. As a part of the static testing process, the process is checked to ensure that the procedures as documented have been followed and the process is in compliance with the standard.
1.8.2 Verification versus Validation

Verification ensures that the system (software, hardware, documentation, and personnel) complies with an organization’s standards and processes, relying on review or non-executable methods. Validation physically ensures that the system operates according to the desired specifications by executing the system functions through a series of tests that can be observed and evaluated.

Keep in mind that verification and validation techniques can be applied to every element of the applicant system. You’ll find these techniques in publications dealing with the design and implementation of user manuals and training courses.

1.8.2.1 Verification Testing

Verification requires several types of reviews, including requirements reviews, design reviews, code walkthroughs, code inspections, and test reviews. The system tester should be involved in these reviews to find defects before they are built into the system. Table 1-11 shows examples of verification. The list is not exhaustive, but it does show who performs the task and what the deliverables are. A detailed discussion of walkthroughs, checkpoint reviews, and inspections is found in Skill Category 6.

<table>
<thead>
<tr>
<th>Verification Example</th>
<th>Performed by</th>
<th>Explanation</th>
<th>Deliverable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements Reviews</td>
<td>Business analysts, Development team, Test team, Users</td>
<td>The study and discussion of the computer system requirements to ensure they meet stated user needs and are feasible.</td>
<td>Reviewed statement of requirements, ready to be translated into a system design.</td>
</tr>
<tr>
<td>Design Reviews</td>
<td>Business analysts, Development team, Test team, Users</td>
<td>The study and discussion of the application system design to ensure it will support the system requirements.</td>
<td>System design, ready to be translated into computer programs, hardware configurations, documentation, and training.</td>
</tr>
<tr>
<td>Code Walkthroughs</td>
<td>Development team</td>
<td>An informal analysis of the program source code to find defects and verify coding techniques.</td>
<td>Computer software ready for testing or more detailed inspections by the developer.</td>
</tr>
</tbody>
</table>
Validation is accomplished simply by executing a real-life function (if you wanted to check to see if your mechanic had fixed the starter on your car, you’d try to start the car). Examples of validation are shown in Table 1-12. As in the table above, the list is not exhaustive.

### Table 1-11 Computer System Verification Examples

<table>
<thead>
<tr>
<th>Verification Example</th>
<th>Performed by</th>
<th>Explanation</th>
<th>Deliverable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Inspections</td>
<td>Development team</td>
<td>A formal analysis of the program source code to find defects as defined by meeting computer system design specifications. Usually performed by a team composed of developers and subject matter experts.</td>
<td>Computer software ready for testing by the developer.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Validation Example</th>
<th>Performed by</th>
<th>Explanation</th>
<th>Deliverable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit Testing</td>
<td>Developers</td>
<td>The testing of a single program, module, or unit of code. Usually performed by the developer of the unit. Validates that the software performs as designed.</td>
<td>Software unit ready for testing with other system components, such as other software units, hardware, documentation, or users.</td>
</tr>
<tr>
<td>Integration Testing</td>
<td>Developers with support from an independent test team</td>
<td>The testing of related programs, modules, or units of code. Validates that multiple parts of the system interact according to the system design.</td>
<td>Portions of the system ready for testing with other portions of the system.</td>
</tr>
</tbody>
</table>
### Table 1-12 Computer System Validation Examples

<table>
<thead>
<tr>
<th>Validation Example</th>
<th>Performed by</th>
<th>Explanation</th>
<th>Deliverable</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Testing</td>
<td>Independent Test Team</td>
<td>The testing of an entire computer system. This kind of testing can include functional and structural testing, such as stress testing. Validates the system requirements.</td>
<td>A tested computer system, based on what was specified to be developed or purchased.</td>
</tr>
<tr>
<td>User Acceptance Testing</td>
<td>Users with support from an independent test team</td>
<td>The testing of a computer system or parts of a computer system to make sure it will solve the customer’s problem regardless of what the system requirements indicate.</td>
<td>A tested computer system, based on user needs.</td>
</tr>
</tbody>
</table>

Determining when to perform verification and validation relates to the development model used. In waterfall, verification tends to be phase-end with validation during the unit, integration, system and UAT processes.

### 1.8.3 Traceability Matrix

One key component of a life cycle test approach is verifying at each step of the process the inputs to a stage are correctly translated and represented in the resulting artifacts. Requirements, or stakeholder needs, are one of these key inputs that must be traced throughout the rest of the software development life cycle.

The primary goal of software testing is to prove that the user or stakeholder requirements are actually delivered in the final product developed. This can be accomplished by tracing these requirements, both functional and non-functional, into analysis and design models, test plans and code to ensure they’re delivered. This level of traceability also enables project teams to track the status of each requirement throughout the development and test process.

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Func Rqmt 1.1</th>
<th>Func Rqmt 1.2</th>
<th>Func Rqmt 1.3</th>
<th>Func Rqmt 1.x</th>
<th>Func Rqmt x.x</th>
<th>Technical Rqmt 1.1</th>
<th>Technical Rqmt 1.2</th>
<th>Technical Rqmt 1.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Cases</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
If a project team is developing a web-based application, the requirements or stakeholder needs will be traced to use cases, activity diagrams, class diagrams and test cases or scenarios in the analysis stage of the project. Reviews for these deliverables will include a check of the traceability to ensure that all requirements are accounted for.

In the design stage of the project, the tracing will continue to design and test models. Again, reviews for these deliverables will include a check for traceability to ensure that nothing has been lost in the translation of analysis deliverables. Requirements mapping to system components drives the test partitioning strategies. Test strategies evolve along with system mapping. Test cases to be developed need to know where each part of a business rule is mapped in the application architecture. For example, a business rule regarding a customer phone number may be implemented on the client side as a GUI field edit for high performance order entry. In another it may be implemented as a stored procedure on the data server so the rule can be enforced across applications.

When the system is implemented, test cases or scenarios will be executed to prove that the requirements were implemented in the application. Tools can be used throughout the project to help manage requirements and track the implementation status of each one.
1.9 Testing Schools of Thought and Testing Approaches

Within any profession there exists different ways to approach solving a problem. A primary objective of a test professional is to solve the problem of poor software quality (which could be measured in a variety of ways from defects found in production to user dissatisfaction). Individuals learn to solve problems from the moment the cognitive switch is turned on. Problem solving skills are honed as we mature through the collection of experiences gained by learning, doing, failing, and succeeding. Taking those evolutionary problem solving capabilities into the software testing profession, they are further influenced by new experiences, our peer relationships, existing institutionalized problem solving approaches, policies, and standards. At any given point in time, problem solving maturity is a product of the accumulation of these experiences. Eventually, the natural process of individuals seeking like individuals creates groups with similar ideas.

1.9.1 Software Testing Schools of Thought

Just as there are various models for the SDLC, there are different “schools of thought” within the testing community. A school of thought simply defined as “a belief (or system of beliefs) shared by a group.” Dr. Cem Kaner, Bret Pettichord, and James Bach are most often cited in regard to the “software testing schools.” The first substantive discussion about these schools was by Bret Pettichord (2003) who described the following five schools (initially four with Agile added later).

1.9.1.1 Analytical School

The Analytical school sees testing as rigorous and technical. This school places emphasis on modeling or other more theoretical/analytical methods for assessing the quality of the software.

1.9.1.2 Factory School

The Factory school emphasizes reduction of testing tasks to basic routines or very repetitive tasks. Outsourcing aligns well with the Factory school approach.

1.9.1.3 Quality (control) School

In the Quality school the emphasis is on process and relies heavily on standards. Testing is a disciplined sport and in the Quality school the test team may view themselves as the gatekeepers who protects the user from the poor quality software.
1.9.1.4 **Context-driven School**

In the Context-driven school the emphasis is on adapting to the circumstances under which the product is developed and used. In this school the focus is on product and people rather than process.

1.9.1.5 **Agile School**

The Agile school emphasizes the continuous delivery of working product. In this school testing is code focused testing by programmers. In the Agile school testing is focused on automated unit tests as used in test-driven development or test-first development.

The introduction of the schools of software testing was not a moment in time event but rather as the profession matured, diversify of thought evolved, a relatively discrete number of similar approaches emerged. One school of thought is not suggested as better than the next, nor are they competitive, but rather a problem solving approach to test software. While individuals may align themselves with one school or another, the important issue is to recognize that some approaches may serve a test project better than another. This would not be based on the personal alignment of the individual but rather the nuances of the project be it a legacy “big iron” systems or mobile device applications.

1.9.2 **Testing Approaches**

The discussion of schools of thought described five high-level approaches to software testing. Additionally to the schools of software testing, there are other ways the software testing approach can be delineated. These approaches do not specifically relate to static or dynamic testing nor are they specific to a development model or testing school. Shown here are five different approaches to testing software applications:

- Requirements-based Testing
- Risk-based Testing
- Model-based Testing
- Exploratory Testing
- Keyword-driven Testing

1.9.2.1 **Requirements-based Testing (RBT)**

Requirements-based testing is self-definitional. RBT focuses on the quality of the Requirements Specification and requires testing throughout the development life cycle. Specifically, RBT performs static tests with the purpose of verifying that the requirements meet acceptable standards defined as: complete, correct, precise, unambiguous, and clear, consistent, relevant, testable, and traceable. Also, RBT focuses on designing a necessary and
sufficient set of test cases from those requirements to ensure that the design and code fully meet those requirements.

### 1.9.2.2 Risk-based Testing

A goal of software testing is to reduce the risk associated with the deployment of an automated system (the software application). Risk-based testing prioritizes the features and functions to be tested based on the likelihood of failure and the impact of a failure should it occur.

Risk-based testing requires the professional tester to look at the application from as many viewpoints as possible on the likelihood of failure and the impact of a failure should it occur.

1. Make a list of risks. This process should include all stakeholders and must consider both process risks and product risks.
2. Analyze then prioritize the list of risks.
3. Assign test resources based on risk analysis.
4. Design and execute tests that evaluate each risk.
5. With each iteration and the removal of defects (reduced risk), reevaluate and re-prioritize tests.

### 1.9.2.3 Model-based Testing (MBT)

In Model-based Testing test cases are based on a simple model of the application. Generally, models are used to represent the desired behavior of the application being tested. The behavioral model of the application is derived from the application requirements and specification. It is not uncommon that the modeling process itself will reveal inconsistencies and deficiencies in the requirements and is an effective static test process. Test cases derived from the model are functional tests, so model-based testing is generally viewed as black-box testing.

### 1.9.2.4 Exploratory Testing (ET)

The term “Exploratory Testing” was coined in 1983 by Dr. Cem Kaner. Dr. Kaner defines exploratory testing as “a style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the quality of his/her work by treating test-related learning, test design, test execution, and test result interpretation as mutually supportive activities that run in parallel throughout the project.” Exploratory Testing is aligned with the Context Driven testing school of thought.

Exploratory testing has always been performed by professional testers. The Exploratory Testing style is quite simple in concept; the tester learns things that, together with experience and creativity, generates new good tests to run. Exploratory testing seeks to find out how the
software actually works and to ask questions about how it will handle difficult and easy cases. The quality of the testing is dependent on the tester’s skill of inventing test cases and finding defects. The more the tester knows about the product and different test methods, the better the testing will be.

Exploratory testing is not a test technique but rather a style of testing used throughout the application life cycle. According to Dr. Kaner and James Bach, exploratory testing is more a mindset, “a way of thinking about testing,” than a methodology. As long as the tester is thinking and learning while testing and subsequent tests are influenced by the learning, the tester is performing exploratory testing.

1.9.2.5 Keyword-driven Testing (KDT)

Keyword-driven testing, also known as table-driven testing or action word based testing, is a testing methodology whereby tests are driven wholly by data. Keyword-driven testing uses a table format, usually a spreadsheet, to define keywords or action words for each function that will be executed. In Keyword-driven tests the data items are not just data but also the names of specific functions being tested and their arguments which are then executed as the test runs.

Keyword-driven testing is well suited for the non-technical tester. KDT also allows automation to be started earlier in the SDLC and has a high degree of reusability.

1.10 Test Categories and Testing Techniques

Tests can be classified according to whether they are derived from a description of the program’s function, from the program’s structure, or from the implementation of the quality attributes and characteristics of the system. Both structural, functional, and non-functional tests should be performed to ensure adequate testing. Structural test sets tend to uncover errors that occur during “coding” of the program; functional test sets tend to uncover errors that occur in implementing requirements or design specifications; and non-functional tests tend to uncover poor design and coding.

1.10.1 Structural Testing

Structural testing can be categorized into two groups, Structural System Testing or White-box Testing.

1.10.1.1 Structural System Testing

Structural System Testing is designed to verify that the developed system and programs work. The objective is to ensure that the
product designed is structurally sound and will function correctly. It attempts to determine that the technology has been used properly and that when all the component parts are assembled they function as a cohesive unit. Structural System Testing could be more appropriately labeled as testing tasks rather than techniques, as Structural System Testing provides the facility for determining that the implemented configuration and its interrelationship of parts function so that they can perform the intended tasks. These test tasks are not designed to ensure that the application system is functionally correct, but rather that it is structurally sound. Examples of structural system testing tasks are shown in Table 1-14.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stress</td>
<td>Determine system performs with expected volumes.</td>
<td>• Sufficient disk space allocated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Communication lines adequate</td>
</tr>
<tr>
<td>Execution</td>
<td>System achieves desired level of proficiency</td>
<td>• Transaction turnaround time adequate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Software/hardware use optimized</td>
</tr>
<tr>
<td>Recovery</td>
<td>System can be returned to an operational status after a failure.</td>
<td>• Induce failure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Evaluate adequacy of backup data</td>
</tr>
<tr>
<td>Compliance (to Process)</td>
<td>System is developed in accordance with standards and procedures.</td>
<td>• Standards followed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Documentation complete</td>
</tr>
</tbody>
</table>

Table 1-14 Structural Testing Techniques

1.10.1.1.1 Stress Testing

Stress testing is designed to determine if the system can function when subjected to large volumes of work. The areas that are stressed include input transactions, internal tables, disk space, output, communications, computer capacity, and interaction with people. If the application functions adequately under stress, it can be assumed that it will function properly with normal volumes of work.

1.10.1.1.2 Execution Testing

Execution testing determines whether the system achieves the desired level of proficiency in a production status. Execution testing can verify response times, turnaround times, as well as design performance. The execution of a system can be tested in whole or in part using the actual system or a simulated model of a system.

1.10.1.1.3 Recovery Testing

Recovery is the ability to restart operations after the integrity of the application has been lost. The process normally involves reverting to a point where the integrity of the system is known, and then processing transactions up to the point of failure. The time required to recover operations is affected by the number of restart points, the volume of applications run on the
The importance of recovery will vary from application to application.

1.10.1.4 Compliance Testing

Compliance testing verifies that the application was developed in accordance with information technology standards, procedures, and guidelines. The methodologies are used to increase the probability of success, to enable the transfer of people in and out of the project with minimal cost, and to increase the maintainability of the application system. The type of testing conducted varies on the phase of the system development life cycle. However, it may be more important to compliance test adherence to the process during requirements than at later stages in the life cycle because it is difficult to correct applications when requirements are not adequately documented.

1.10.1.2 White-box Testing

White-box testing assumes that the path of logic in a unit or program is known. White-box testing consists of testing paths, branch by branch, to produce predictable results. The following are white-box testing techniques:

1.10.1.2.1 Statement Coverage

Execute all statements at least once.

1.10.1.2.2 Decision Coverage

Execute each decision direction at least once.

1.10.1.2.3 Condition Coverage

Execute each decision with all possible outcomes at least once.

1.10.1.2.4 Decision/Condition Coverage

Execute all possible combinations of condition outcomes in each decision. Treat all iterations as two-way conditions exercising the loop zero times and one time.

1.10.1.2.5 Multiple Condition Coverage

Invoke each point of entry at least once.
1.10.2 Functional Testing

Like Structural Testing, Functional Testing can test the system level functionality as well as application functionality. Functional tests at the application level are frequently referred to as Black-box Test techniques.

1.10.2.1 Functional System Testing

Functional system testing ensures that the system requirements and specifications are achieved. The process involves creating test conditions for use in evaluating the correctness of the application. Examples of Functional System Testing are shown in Table 1-15.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>System performs as specified.</td>
<td>• Prove system requirements</td>
</tr>
<tr>
<td>Error Handling</td>
<td>Errors can be prevented or detected and then corrected.</td>
<td>• Error introduced into test</td>
</tr>
<tr>
<td>Intersystem</td>
<td>Data is correctly passed from system to system.</td>
<td>• Intersystem parameters changed</td>
</tr>
<tr>
<td>Control</td>
<td>Controls reduce system risk to an acceptable level.</td>
<td>• File reconciliation procedures work</td>
</tr>
<tr>
<td>Parallel</td>
<td>Old system and new system are run and the results compared to detect unplanned differences.</td>
<td>• Old and new systems</td>
</tr>
</tbody>
</table>

Table 1-15 Functional System Testing Techniques

1.10.2.1.1 Requirements Testing

Requirements testing verifies that the system can perform its function correctly and that the correctness can be sustained over a continuous period of time. Unless the system can function correctly over an extended period of time, management will not be able to rely upon the system. The system can be tested for correctness throughout the life cycle, but it is difficult to test the reliability until the program becomes operational. Requirements testing is primarily performed through the creation of test conditions and functional checklists. Test conditions are generalized during requirements and become more specific as the SDLC progresses leading to the creation of test data for use in evaluating the implemented application system.
1.10.2.1.2 Error Handling

Error-handling testing determines the ability of the application system to properly process incorrect transactions. Error-handling testing requires a group of knowledgeable people to anticipate what can go wrong with the application system in an operational setting. Error-handling testing should test the error condition is properly corrected. This requires error-handling testing to be an iterative process in which errors are first introduced into the system, then corrected, then reentered into another iteration of the system to satisfy the complete error-handling cycle.

1.10.2.1.3 Interconnection Testing

Application systems are frequently interconnected to other application systems. The interconnection may be data coming into the system from another application, leaving for another application, or both. Frequently, multiple applications—sometimes called cycles or functions—are involved. For example, there is a revenue function or cycle that interconnects all of the income-producing applications such as order entry, billing, receivables, shipping and returned goods. Intersystem testing is designed to ensure that the interconnections between applications functions correctly.

1.10.2.1.4 Control Testing

Approximately one-half of the total system development effort is directly attributable to controls. Controls include data validation, file integrity, audit trail, backup and recovery, documentation, and the other aspects of systems related to integrity. Control testing techniques are designed to ensure that the mechanisms that oversee the proper functioning of an application system work. A more detailed discussion of Control Testing is found in Skill Category 7.

1.10.2.1.5 Parallel Testing

Parallel testing requires that the same input data be run through two versions of the same application. Parallel testing can be done with the entire application or with a segment of the application. Sometimes a particular segment, such as the day-to-day interest calculation on a savings account, is so complex and important that an effective method of testing is to run the new logic in parallel with the old logic.

1.10.2.2 Black-box Testing Techniques

Black-box testing focuses on testing the function of the program or application against its specification. Specifically, this technique determines whether combinations of inputs and operations produce expected results. The following are black-box testing techniques. Each technique is discussed in detail in Skill Category 7.
1.10.2.2.1 Equivalence Partitioning

In Equivalence Partitioning, the input domain of a system is partitioned into classes of representative values so that the number of test cases can be limited to one-per-class, which represents the minimum number of test cases that must be executed.

1.10.2.2.2 Boundary Value Analysis

Boundary Value Analysis is a data selection technique in which test data is chosen from the “boundaries” of the input or output domain classes, data structures, and procedure parameters. Choices often include the actual minimum and maximum boundary values, the maximum value plus or minus one, and the minimum value plus or minus one.

1.10.2.2.3 Decision Table Testing

A technique useful for analyzing logical combinations of conditions and their resultant actions to minimize the number of test cases needed to test the program’s logic.

1.10.2.2.4 State Transition Testing

An analysis of the system to determine a finite number of different states and the transitions from one state to another. Tests are then based on this analysis.

1.10.2.2.5 All-pairs (pairwise) Testing

A combinatorial method that for each pair of input parameters tests all possible discrete combinations of those parameters.

1.10.2.2.6 Cause-Effect Graphing

Cause-Effect Graph is a technique that graphically illustrates the relationship between a given outcome and all the factors that influence the outcome.

1.10.2.2.7 Error Guessing

Error Guessing is a test data selection technique for picking values that seem likely to cause defects. This technique is based on the intuition and experience of the tester.

1.10.3 Non-Functional Testing

Non-functional testing validates that the system quality attributes and characteristics have been considered during the development process. Section 1.2.5 described a variety of software quality factors and software quality characteristics. The ISO 25010:2011 standard named additional quality objectives. Non-functional testing would include test cases that test application characteristics like those in the list below:
1.10.4 Incremental Testing

Incremental testing is a disciplined method of testing the interfaces between unit-tested programs as well as between system components. It involves adding unit-tested programs to a given module or component one by one, and testing each resultant combination. This is not to be confused with the Incremental Development Model. There are two types of incremental testing:

• **Top-down**
  
  Begin testing from the top of the module hierarchy and work down to the bottom using interim stubs to simulate lower interfacing modules or programs. Modules are added in descending hierarchical order.

![Figure 1-21](image)

*Figure 1-21 Stubs simulate the behaviors of a lower level module*

• **Bottom-up**

  Begin testing from the bottom of the hierarchy and work up to the top. Modules are added in ascending hierarchical order. Bottom-up testing requires the development of driver modules, which provide the test input, that call the module or program being tested, and display test output.
Figure 1-22 Drivers call the module(s) being tested

Within the context of incremental testing, test stubs and drivers are often referred to as a test harness. This is not to be confused with the term test harness used in the context of test automation.

### 1.10.5 Thread Testing

This test technique, which is often used during early integration testing, demonstrates key functional capabilities by testing a string of units that accomplish a specific function in the application. Thread testing and incremental testing are usually utilized together. For example, units can undergo incremental testing until enough units are integrated and a single business function can be performed, threading through the integrated components.

When testing client/server applications, these techniques are extremely critical. An example of an effective strategy for a simple two-tier client/server application could include:

1. Unit and bottom-up incremental testing of the application server components
2. Unit and incremental testing of the GUI (graphical user interface) or client components
3. Testing of the network
4. Thread testing of a valid business transaction through the integrated client, server, and network
Table 1-16 illustrates how the various techniques can be used throughout the standard test stages.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>White-Box</td>
</tr>
<tr>
<td>Unit Testing</td>
<td>X</td>
</tr>
<tr>
<td>String/Integration Testing</td>
<td>X</td>
</tr>
<tr>
<td>System Testing</td>
<td></td>
</tr>
<tr>
<td>Acceptance Testing</td>
<td></td>
</tr>
</tbody>
</table>

Table 1-16 Testing techniques and standard test stages

It is important to note that when evaluating the paybacks received from various test techniques, white-box or program-based testing produces a higher defect yield than the other dynamic techniques when planned and executed correctly.

1.10.6 Regression Testing

Regression Testing has been singled out not as an afterthought but because regression testing doesn’t fit into any of the aforementioned categories. Regression testing isn’t an approach, a style, or a testing technique. Regression testing is a “decision.” It is a decision to re-test something that has already been tested. The purpose of regression testing is to look for defects that may have been inadvertently introduced or manifested as an unintended consequence of other additions or modifications to the application code, operating system, or other impacting program. Simply stated, the purpose of regression testing is to make sure unchanged portions of the system work as they did before a change was made.

When should regression testing be performed?

- New releases of packaged software are received
- Application software is enhanced or any changes made
- Support software changes (OS, utilities, object libraries)
- Either side of a system interface is changed
- Changes to configuration
- Whenever changes are made after a testing stage is completed
1.10.7 Testing Specialized Technologies

In Skill Category 10, specialized testing techniques are described for such specialized technologies as:

- Internet Applications
- Agile Environment
- Mobile Applications
- Cloud Computing
- DevOps
- Internet of Things
Building the Software Testing Ecosystem

Ecosystem - a community of organisms together with their physical environment, viewed as a system of interacting and interdependent relationships.¹

If the pages of this STBOK talked only to the issues of writing a test case, developing a test plan, recording a defect, or testing within an Agile framework, the most important success factor would have been overlooked, the software testing ecosystem. Like any thriving ecosystem, a thriving testing ecosystem requires balanced interaction and interdependence. Not unlike the three-legged stool of people, process, and technology, if one leg is missing, the stool will not fulfill its objective. A successful software testing ecosystem must include the right organizational policies, procedures, culture, attitudes, rewards, skills, and tools.

<table>
<thead>
<tr>
<th>Skill Category</th>
<th>2</th>
</tr>
</thead>
</table>

| Management’s Role          | 2-2 |
| Work Processes             | 2-5 |
| Test Environment           | 2-26 |
| Test Tools                 | 2-31 |
| Skilled Team               | 2-48 |

¹ The American Heritage® Science Dictionary, Copyright © 2002. Published by Houghton Mifflin. All rights reserved.
2.1 Management’s Role

It starts at the top! The tone for the organization starts with the person in the leadership role. Whether the top is defined as the executive suite, a department head or a team lead, success depends first and foremost on humble, curious, and enlightened leadership. Without that the value of the ecosystem is greatly diminished.

2.1.1 Setting the Tone

An essential component of the testing ecosystem is the tone set by management. The “tone” is representative of the ecosystem that management has established which influences the way people work. The role of management is to create an ecosystem where people matter, quality matters, and where integrity and ethical behavior are the way things are done. Team member behavior is reflective of management’s behavior. Management’s tone is reflected in the following ways:

- Integrity and Ethical Values
- Incentives for Unethical Behavior
- Moral Behavior Role Model

2.1.1.1 Integrity and Ethical Values

An organization’s objectives and the way they are achieved are based on preferences, value judgments, and management styles. Those preferences and value judgments, which are translated into standards of behavior, reflect management’s integrity and its commitment to ethical values.

Integrity is a prerequisite for ethical behavior in all aspects of an enterprise’s activities. A strong corporate ethical climate at all levels is vital to the well-being of the organization, all of its constituencies, and the public at large. Such a climate contributes substantially to the effectiveness of the ecosystem.

Ethical behavior and management integrity are products of the “corporate culture.” Corporate culture includes ethical and behavioral standards, how they are communicated, and how they are reinforced in practice. Official policies specify what management wants to happen.
Corporate culture determines what actually happens and which rules are obeyed, bent, or ignored.

### 2.1.1.2 Incentives for Unethical Behavior

Individuals may engage in dishonest, illegal, or unethical acts simply because their organizations give them strong incentives to do so. Emphasis on “results,” particularly in the short term, fosters an environment in which the price of failure becomes very high.

Incentives for engaging in unethical behavior include:
- Pressure to meet unrealistic performance targets
- High performance-dependent rewards

Removing or reducing incentives can go a long way toward diminishing undesirable behavior. Setting realistic performance targets is a sound motivational practice; it reduces counterproductive stress as well as the incentive for fraudulent reporting that unrealistic targets create. Similarly, a well-controlled reporting system can serve as a safeguard against a temptation to misstate results.

### 2.1.1.3 Moral Behavior Role Model

The most effective way of transmitting a message of ethical behavior throughout the organization is by example. People imitate their leaders. Employees are likely to develop the same attitudes about what’s right and wrong as those shown by top management. Knowledge that the CEO has “done the right thing” ethically when faced with a tough business decision sends a strong message to all levels of the organization.

Setting a good example is not enough. Top management should verbally communicate the entity’s values and behavioral standards to employees. This verbal communication must be backed up by a formal code of corporate conduct. The formal code is “a widely used method of communicating to employees the company’s expectations about duty and integrity.” Codes address a variety of behavioral issues, such as integrity and ethics, conflicts of interest, illegal or otherwise improper payments, and anti-competitive arrangements. While codes of conduct can be helpful, they are not the only way to transmit an organization’s ethical values to employees, suppliers, and customers.

### 2.1.2 Commitment to Competence

A critical component to the thriving ecosystem is a competent team. Competence should reflect the knowledge and skills needed to accomplish tasks that define the individual’s job. The role of management must be to specify the competence levels for particular jobs and to translate those levels into requisite knowledge and skills. The necessary knowledge and skills may in turn depend on individual’s intelligence, training, and experience. Among the many
factors considered in developing knowledge and skill levels are the nature and degree of judgment to be applied to a specific job.

### 2.1.3 The Organizational Structure Within the Ecosystem

An entity’s organizational structure provides the framework within which its activities for achieving objectives are planned, executed, controlled, and monitored. Significant aspects of establishing a relevant structure include defining key areas of authority and responsibility and establishing appropriate lines of reporting. The appropriateness of an organizational structure depends, in part, on its size, maturity, and operating philosophy. A large organization may require very formal reporting lines and responsibilities. Such formality might impede the flow of information in a smaller organization. Whatever the structure, activities will be organized to carry out the strategies designed to achieve particular objectives of the ecosystem.

#### 2.1.3.1 Assignment of Authority and Responsibility

Management has an important function assigning authority and responsibility for operating activities and establishment of reporting relationships and authorization protocols. It involves the degree to which individuals and teams are encouraged to use initiative in addressing issues and solving problems, as well as limits of their authority. There is a growing tendency to push authority downward in order to bring decision-making closer to front-line personnel. An organization may take this tactic to become more market-driven or quality focused—perhaps to eliminate defects, reduce cycle time or increase customer satisfaction. To do so, the organization needs to recognize and respond to changing priorities in market opportunities, business relationships, and public expectations.

Alignment of authority and accountability is often designed to encourage individual initiatives, within limits. Delegation of authority, or “empowerment,” means surrendering central control of certain business decisions to lower echelons—to the individuals who are closest to everyday business transactions. A critical challenge is to delegate only to the extent required to achieve objectives. This requires ensuring that risk acceptance is based on sound practices for identification and minimization of risk including sizing risks and weighing potential losses versus gains in arriving at good business decisions.

Another challenge is ensuring that all personnel understand the objectives of the ecosystem. It is essential that each individual know how his or her actions interrelate and contribute to achievement of the objectives.

### 2.1.4 Meeting the Challenge

It is essential that personnel be equipped for new challenges as issues that organizations face change and become more complex—driven in part by rapidly changing technologies and increasing competition. Education and training, whether classroom instruction, self-study, or
on-the-job training, must prepare an entity’s people to keep pace and deal effectively with the evolving environment. They will also strengthen the organization’s ability to effect quality initiatives. Hiring of competent people and one-time training are not enough; the education process must be ongoing.

2.2 Work Processes

Once management commits to creating an ecosystem conducive to effective software testing, test work processes need to be created. To that end, players within the ecosystem must establish, adhere to, and maintain the work processes. It is the tester’s responsibility to follow these test work processes, and management’s responsibility to ensure that processes are changed if the processes do not work.

It is also critical that the work processes represent sound policies, standards, and procedures. It must be emphasized that the purposes and advantages of standards exist only when sound work processes are in place. If the processes are defective or out of date, the purposes will not be met. Poor standards can, in fact, impede quality and reduce productivity.

2.2.1 What is a Process?

A process can be defined as a set of activities that represent the way work is performed. The outcome from a process is usually a product or service. Both software development and software testing are processes. Table 2-1 illustrates a few examples of processes and their outcomes.

<table>
<thead>
<tr>
<th>Examples of Processes</th>
<th>Deliverables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyze Business Needs</td>
<td>Needs statement</td>
</tr>
<tr>
<td>Daily Scrum</td>
<td>Understanding of what work has been done and what work remains</td>
</tr>
<tr>
<td>Conduct JAD Session</td>
<td>JAD notes</td>
</tr>
<tr>
<td>Test Planning</td>
<td>Test plan</td>
</tr>
<tr>
<td>Unit Test</td>
<td>Defect-free unit</td>
</tr>
</tbody>
</table>

Table 2-1 Process Example and Deliverables
2.2.2 Components of a Process

A process has six basic components. It is important to have a common understanding of these definitions. The components are:

- **Policy** - Managerial desires and intents concerning either processes or products.
- **Standards** - The measure used to evaluate products and identify nonconformance. The basis upon which adherence to policies is measured.
- **Inputs** - The data and information required from the “suppliers” to be transformed by a process into the required deliverable of that process.
- **Procedures** - Describe how work must be done and how methods, tools, techniques, and people are applied to perform a process. There are Do procedures and Check procedures. Procedures indicate the “best way” to meet standards.
- **Deliverables** - Any product or service produced by a process. Deliverables can be interim or external. Interim deliverables are produced within the process but never passed on to another process. External deliverables may be used by one or more processes. Deliverables serve as both inputs to and outputs from a process.
- **Tools** - Any resources that are not consumed in converting the input into the deliverable.

Policies provide direction, standards are the rules or measures by which the implemented policies are measured, and the procedures are the means used to meet or comply with the standards. These definitions show the policy at the highest level, standards established next, and procedures last. However, the worker sees a slightly different view, which is important in explaining the practical view of standards.

2.2.3 Tester’s Workbench

An illustration of a process, sometimes referred to as a Process Workbench, is shown in Figure 2-1.
The objective of the workbench is to produce the defined output products (deliverables) in a defect-free manner. The procedures and standards established for each workbench are designed to assist in this objective. If defect-free products are not produced, they should be reworked until the defects are removed or, with management's concurrence, the defects can be noted and the defective products passed to the next workbench.

Many of the productivity and quality problems within the test function are attributable to the incorrect or incomplete definition of tester’s workbenches. For example, workbenches may not be established, or too few may be established. A test function may only have one workbench for software test planning, when in fact, it should have several; such as, a budgeting workbench, a scheduling workbench, a risk assessment workbench, and a tool selection workbench. In addition, they may have an incompletely defined test data workbench that leads to poorly defined test data for whatever tests are made at the workbench, which has not been fully defined.

The worker performs defined procedures on the input products in order to produce the output products. The procedures are step-by-step instructions that the worker follows in performing his/her tasks. Note that if tools are to be used, they are incorporated into the procedures.

The standards are the measures that the worker uses to validate whether or not the products have been produced according to specifications. If they meet specifications, they are quality products or defect-free products. If they fail to meet specifications or standards, they are defective and subject to rework.

It is the execution of the workbench that defines product quality and is the basis for productivity improvement. Without process engineering, the worker has little direction or guidance to know the best way to produce a product and to determine whether or not a quality product has been produced.
2.2.3.1 Workbenches are Incorporated into a Process

To understand the testing process, it is necessary to understand the workbench concept. In IT, workbenches are more frequently referred to as phases, steps, or tasks. The workbench is a way of illustrating and documenting how a specific activity is to be performed.

The workbench and the software testing process, which is comprised of many workbenches, are illustrated in Figure 2-2.

![Figure 2-2 Software Testing Process](image)

The workbench concept can be used to illustrate one of the steps involved in testing software systems. The tester’s unit test workbench consists of these steps:

1. Input products (unit test specifications) are given to the tester.
2. Work is performed (e.g., debugging); a procedure is followed; and a product or interim deliverable is produced, such as a defect list.
3. Work is checked to ensure the unit meets specifications and standards, and that the procedure was followed.
4. If check finds no problems, the product is released to the next workbench (e.g., integration testing).
   
If check finds problems, the product is sent back for rework.

2.2.4 Responsibility for Building Work Processes

It is important that organizations clearly establish who is responsible for developing work processes (i.e., policies, procedures, and standards). Responsibility must be assigned and fulfilled. Having them developed by the wrong group can impede the effectiveness of the processes.
2.2.4.1 Responsibility for Policy

IT management is responsible for issuing IT policy. Policies define the intent of management, define direction, and, by definition, are general rules or principles. It is the standards that will add the specificity needed for implementation of policies. For example, the test team needs direction in determining how many defects are acceptable in the product under test. If there is no policy on defects, each worker decides what level of defects is acceptable.

The key concepts that need to be understood on policies are:

- Policies are developed by senior management. (Note that in some instances subordinates develop the policy, but senior management approves it.)
- Policies set direction but do not define specific products or procedures.
- Policies define the areas in which standards and procedures will be developed. If there are no policies, there should, by definition, be no standards or procedures in that area.

2.2.4.2 Responsibility for Standards and Procedures

The workers who use the procedures and are required to comply with the standards should be responsible for the development of those standards and procedures. Management sets the direction and the workers define that direction. This division permits each to do what they are best qualified to do. Failure to involve workers in the development of standards and procedures robs the company of the knowledge and contribution of the workers. In effect, it means that the people best qualified to do a task (i.e., development of standards and procedures) are not involved in that task. It does not mean that every worker develops his own procedures and standards, but that the workers have that responsibility and selected workers will perform the tasks.

2.2.4.3 Key Concepts of Process Engineering

The key concepts of a process engineering program are:

- Management provides an organizational structure for the workers to develop their own standards and procedures
- The program is driven by management policies
- Absolute compliance to standards and procedures is required
- A mechanism is provided for the continual maintenance of standards and procedures to make them more effective

Please note that the software testers should be the owners of test processes—and thus involved in the selection, development, and improvement of test processes.
2.2.5 Continuous Improvement

Process improvement is best considered as a continuous process, where the organization moves continually around an improvement cycle. Within this cycle, improvement is accomplished in a series of steps or specific actions. In sections 1.4.1 and 1.4.2 of Skill Category 1, the CMMI-Dev Process Improvement Model and TMMi Test Process Improvement Model were described. Both models describe a macro view of the improvement process within their respective domains.

### 2.2.5.1 PDCA Model for Continuous Improvement of Processes

One of the best known process improvement models is the Plan-Do-Check-Act model for continuous process improvement. The PDCA model was developed in the 1930s by Dr. Walter Shewhart of Bell Labs. The PDCA model is also known as the Deming circle/cycle/wheel, the Shewhart cycle, and the control circle/cycle. A brief description of the four components of the PDCA concept are provided below and are illustrated in Figure 2-3.

![PDCA Concept](image)

#### Figure 2-3 PDCA Concept

### 2.2.5.1.1 (P) Plan - Devise Your Plan

Define your objective and determine the conditions and methods required to achieve your objective. Clearly describe the goals and policies needed to achieve the objective at this stage. Express a specific objective numerically. Determine the procedures and conditions for the means and methods you will use to achieve the objective.

### 2.2.5.1.2 (D) Do - Execute the Plan

Create the conditions and perform the necessary teaching and training to execute the plan. Make sure everyone thoroughly understands the objectives and the plan. Teach workers the procedures and skills they need to fulfill the plan and thoroughly understand the job. Then, perform the work according to these procedures.
2.2.5.1.3  (C) Check - Check the Results

Check to determine whether work is progressing according to the plan and whether the expected results are obtained. Check for performance of the set procedures, changes in conditions, or abnormalities that may appear. As often as possible, compare the results of the work with the objectives.

2.2.5.1.4  (A) Act - Take the Necessary Action

If your checkup reveals that the work is not being performed according to plan or that results are not as anticipated, devise measures for appropriate action.

If a check detects an abnormality—that is, if the actual value differs from the target value—search for the cause of the abnormality and eliminate the cause. This will prevent the recurrence of the defect. Usually you will need to retrain workers and revise procedures to eliminate the cause of a defect.

2.2.5.2  Continuous Ascending Spiral

Continuous process improvement is not achieved by a one time pass around the PDCA cycle. It is by repeating the PDCA circle continuously that process improvement happens. This concept is best illustrated by an ascending spiral as shown in Figure 2-4.

*Figure 2-4  PDCA Ascending Spiral*
2.2.6 Analysis and Improvement of the Test Process

Industry studies have shown that testers make more defects (process defects) in performing test activities than developers do in performing development activities. For example, if the developers make three defects per function point of logic; testers would make more than three defects in testing that function point of logic.

There are two reasons for this high defect rate. The first is that test processes are less mature than most development processes. In this case, mature means the variability in performing the activity. Generally the more variability in a process the higher the defect rates. The second reason is that testers do not have a quality control process over their test process.

Figure 2-5 illustrates the two test processes. One process performs testing and the other parallel process checks the effectiveness of testing.

![Figure 2-5 The Two Testing Processes](image)

Let’s look at an example. If testers were to develop test transactions to test transactions a, b, and c, there would be a quality control process that checks to assure that those three test transactions were actually prepared, and the transactions contained necessary data to perform the specified testing. Quality control of the testing process can be performed by the individual who does the work or by another individual. For example, testers may be involved in performing a code inspection process on program code prior to unit testing. This same inspection process can be used to inspect the test plan, test data, and the documentation produced from testing. Other testers would perform the test inspection process on the work performed by another tester.

Testers are the quality control process for developers. They are checking to see that the work done by developers meets the specifications. The same concepts need to be applied to the test process. If high-quality work is needed, quality control should be an integral part of the test process. Sometimes quality control can be automated, other times it must be performed manually.
2.2.6.1 Test Process Analysis

Test quality control is performed during the execution of the process. In other words, a test task is performed, and then the quality of that test task is checked. Analysis is performed after the test process is completed. This analysis can be performed by the test group, or it can be performed by another group within the IT organization such as the quality assurance function or process engineering activity.

Test process analysis can only be performed if adequate data is maintained during the test process. A determination needs to be made as to the type of analysis that will be performed, and then that data is collected for analysis purposes. Ideally, the data that should be collected is data that is produced as a by-product of testing. This data tends to be more reliable than data which is collected manually during and after the execution of a test activity.

If data is to be collected during the test process for analysis purposes, the process must include the capabilities to collect that data. The type of data collected might include time spent on a specific activity, rework, defects caused by the tester or the test process, surveys of testers and stakeholders in the test process, inquiries by developers and/or users regarding test activities, test reports and test results, as well as logs of the specific activities performed by testers.

There are many reasons why a test group may want to perform an analysis of the test process. The following analyses are the most common in the test industry. Each is discussed in more detail below.

- The effectiveness and efficiency of test processes
- The test objectives are applicable, reasonable, adequate, feasible, and affordable
- The test program meets the test objectives
- The correct test program is being applied to the project
- The test methodology is used correctly
- The task work products are adequate to meet the test objectives
- Analysis of the results of testing to determine the adequacy of testing
- Adequate, not excessive, testing is performed

2.2.6.1.1 Effectiveness and Efficiency of Test Processes

Estimates show that in many organizations testing accounts for 50% of the total project cost. Many organizations are not aware of the division between development work, testing work, and rework because all of those activities are charged to a project without dividing the charge among the three previously described activities.

Effectiveness means that the testers completed their assigned responsibilities. As previously stated, this should be completion of the activities included in the test plan. Efficiency is the amount of resources and time required to complete test responsibilities.

There is normally a trade-off between efficiency and effectiveness. Since there are usually time and budget constraints the testers should be looking for test processes which maximize the two variables of effectiveness and efficiency.
2.2.6.1.2 Test Objectives Applicable, Reasonable, Adequate, Feasible, and Affordable

The test objectives are the responsibilities assigned to testers to test an individual software system. The test objectives should be incorporated into the test plan. The test plan will then define how those test objectives will be accomplished during the execution of testing.

Test objectives may be assigned to testers which are not applicable, not reasonable, not adequate, not feasible and not affordable. The reasons test objectives may not meet these criteria include:

- Testers do not have the needed competencies to meet the test objectives
- Test tools which can complete the test objectives within reasonable time or resource constraints are not available.
- Test objectives are inappropriate for the testers, such as whether the software is appropriately aligned to the corporate business objectives.
- The test objectives overlook what would reasonably be incorporated into software testing, such as identifying implemented functionality not requested by the user of the software.

2.2.6.1.3 Test Program Meets Test Objectives

Surveys by the Quality Assurance Institute have indicated that only approximately one half of the software testing groups develop test plans. Of those groups that develop test plans, approximately 50% do not follow these plans during the test execution activities.

Failure to plan or to follow the plan limits the ability of the software testers to determine whether or not the written or implied test objectives have been achieved. Very few testers can prepare reports at the end of testing indicating whether or not the stated test objectives have been implemented without a plan to follow.

This problem is frequently compounded by the fact that the software specifications have changed from the time the test objectives were defined. This may mean that the objectives need to be modified or changed based on the changes in specification. However, if the test plan and test objectives are not changed to reflect those different specifications, whether the original objectives are met or not met, it may be immaterial based on the new specifications.

2.2.6.1.4 Correct Test Program Applied

Management specifies the approach and test processes that should be used to test software. They must also specify how those general test processes should be customized based on different testing needs and different software development methodologies used.

Depending upon management support for using and following the appropriate test processes, the testers may or may not actually follow those processes. In some IT organizations more emphasis is placed on meeting implementation schedules than on following the test processes.

If the correct test processes are not followed, improvement of those processes is severely limited. In other words, if specific processes or tasks within processes are not followed and
the results are undesirable, it may be difficult to determine whether the cause is the process or the changes to the process made by a single software test team.

There are two aspects of determining whether the correct test program was applied:

- Was it applied as developed and incorporated into the test environment?
- Was it corrected to adjust for changing test conditions and different software development methodologies?

### 2.2.6.1.5 Test Methodology is Used Correctly

This analysis requires an effective quality control process. Without quality control in place it may be difficult and expensive to identify the root cause of a test failure. For example, if a tester selects an alternate test tool rather than the specified test tool, but no record is maintained of what tool is actually used, then compliance to the test methodology is difficult to determine.

If adequate quality control records are maintained over testing, it is possible to identify where the methodology was not complied with. Non-compliance is not necessarily bad, because the process may not be doable as developed. In other words if the tester followed the process exactly as specified an undesirable result would occur. Testers frequently know this and make changes during test execution so that their test objectives can be met. However, for the processes to be continuously improved test management must know:

- Whether or not the process was complied with
- Whether or not the tasks are adequately defined and can be followed

### 2.2.6.1.6 Task Work Products are Adequate to Meet Test Objectives

During development and execution of testing, testers produce many different test products. Among these are risk analysis reports, the test plan, test data specifications, test matrices, test results, interim test status reports and final test reports.

The following two analyses need to be performed on these work products:

- Are they adequate to ensure that the test objectives can be met?
- Are they products, or parts of test products that are not needed to meet the test objectives?

The result of this analysis is to ensure that the appropriate information is prepared and available for the testers and other stakeholders to use. Reports that contain unwanted or unnecessary information make using the work products more difficult.

### 2.2.6.1.7 Analysis of the Results of Testing to Determine the Adequacy of Testing

Two processes must be in place in order to manage testing and assess the adequacy of testing. These are a monitoring process which monitors progress and a communication process which provides information to both monitor and act on monitoring information. Skill Category 9 provides the monitoring and communication processes.
Many test organizations develop “tester’s dashboards” comprised of key indicators to facilitate this monitoring process. The key indicators on the dashboard are those measurements which are needed to assure that testing is performed in an effective, efficient and timely manner. Some of the frequently used key indicators are:

- Budget status
- Schedule status
- Requirements tested correctly
- Requirements tested but not correct
- Severity of recorded defects
- Age of recorded but not corrected defects
- Percentage of test plan completed
- Stakeholders satisfaction

To determine the adequacy of the test process, adequacy must be defined quantitatively. For example, adequacy might define that checkpoints are completed no later than two days following the checkpoint date. If that standard/goal is met then the test program is deemed to be adequate. Another adequate test standard/goal might be that all high-priority and medium-priority requirements are tested, and at least 80% of the low-priority requirements are tested by implementation date.

### 2.2.6.1.8 Adequate, Not Excessive, Testing is Performed

Implementing defective software is a risk; testing is a control designed to eliminate or minimize that risk. However, the cost of controls should never exceed the maximum potential loss associated with risk. For example if the cost of testing requirement X is $10,000 and the potential loss if requirement X does not work correctly is $500, that testing would not be warranted. Perhaps a very minimal test that would cost $200 that could reduce the risk from $500 to $100 would be economically feasible.

Adequate testing needs to be defined to ensure that adequate, but not excessive, testing is performed. To do this testing must establish guidelines as to what is excessive testing. These guidelines might include:

- A potential test objective is not wanted by the stakeholders; for example, testing the efficiency of the software when operating on the hardware
- The cost of testing exceeds the potential loss of the risks associated with not testing
- The cost of using automated tools exceeds the costs of performing the same test manually

To ensure that testing is not excessive, testers must continually question the value of each test they perform. They must also look at the magnitude of the risks that are associated with the software being tested. Once the magnitude of the risks is known, the testers can focus their limited resources on the high-risk attributes associated with the software.
2.2.6.2 Testing Processes Assessment

An important step in the improvement cycle is the execution of data gathering to establish the initial state, and subsequently, to confirm the improvements. A testing process assessment is an excellent way to determine the status of your current test process.

Testing Process Assessment is the thoughtful analysis of testing process results, and then taking corrective action on the identified weaknesses.

Testing process assessment is a means of capturing information describing the current capability of an organization’s test process. Assessment is an activity that is performed either during an improvement initiative or as part of a capability determination. In either case, the formal entry to the assessment process occurs with the compilation of the input, which defines:

- The purpose – why it is being carried out
- The scope – which processes are being assessed and what constraints (if any) apply
- Any additional information that needs to be gathered
- The responsibility for carrying out the assessment

Process assessment is undertaken to measure an organization’s current processes. An assessment may be conducted as a self-assessment, an assisted self-assessment, or an independent assessment. The assessment may be discrete or continuous. A team with, or without, tool support typically carries out a discrete assessment manually. A continuous assessment may use manual methods or automated tools for a data collection. Whatever form of assessment is used, a qualified assessor, who has the competencies required, oversees the assessment.

An assessment is carried out by assessing selected processes against a compatible model of good engineering practice created from, or mapped to, the defined reference model. The reference model defines processes characterized by statements of purpose, and attributes that apply across all processes. The process attributes are grouped into process capability levels that define an ordinal scale of capability.

The assessment output consists of a set of attribute ratings for each process instance assessed and may also include the capability level achieved. Process assessment is applicable in the following circumstances:

- Understanding the state of processes for improvement
- Determining the suitability of processes for a particular requirement or class of requirements

The framework for process assessment:

- Encourages self-assessment
• Takes into account the context in which the assessed process operates
• Produces a set of process ratings (a process profile) rather than a pass or fail result
• Addresses the adequacy of the management of the assessed processes through generic practices
• Is appropriate across all application categories and sizes of organization

The sophistication and complexity required of a process is dependent upon its context. This influences how a qualified assessor judges a practice when assessing its adequacy, and influences the degree of comparability between process profiles.

Within a process improvement context, assessment provides the means of characterizing the current practice within an organizational unit in terms of capability. Analysis of the results in the light of business needs identifies strengths, weaknesses, and risks inherent in the processes. This, in turn, leads to the ability to determine whether the processes are effective in achieving their goals and to identify significant causes of poor quality or overruns in time or cost. These provide the drivers for prioritizing improvements to processes.

One of the most commonly identified weaknesses in software testing has been the lack of facts (metrics), and without facts there is no reason to take action (improvement). Once appropriate measures are identified, tracked, and analyzed, then a plan for continuous improvement can be implemented.

It is important to understand that the concept of “measurement first and action second”, is most effective when the measures are very specific. Measurement must be able to determine the effect of the actions. For example, the metric approach fulfills this requirement in that it shows very specific relationships. Using this concept, if action is taken by changing one of the metric variables, the result of that action can be quickly measured.

Changing the variable in one metric can usually be measured by the changes to another metric. As an example, if the number of defects detected after the system goes into production is higher than expected or desirable, then appropriate and timely action should be taken. That action might be to increase the number of test steps. Obviously, this will increase testing costs with the objective of improving the quality of the product by reducing defects. If, after analysis, it is demonstrated that increasing the emphasis on test steps has had the desirable effect, then these additional steps should be incorporated into the normal testing process. If the opposite is true, then the action did not produce the desired effect and the time and resources spent were less than effective; the actions should be discontinued and another action attempted. The process continues until the appropriate and effective improvement mechanisms are discovered and included in the normal process.

2.2.6.3 Test Process Improvement Model

A model for test process improvement has eight steps:

1. Examine the Organization’s Needs and Business Goals
A process improvement program starts with the recognition of the organization’s needs and business goals, usually based on the main drivers and stimuli identified. From an analysis of the organization’s needs and existing stimuli for improvement, the objectives of process improvement are identified and described. The final stage of the preliminary definition of the goals for the improvement program is setting the priorities of the process improvement objectives.

Once the analysis of the organization’s needs and business goals has been completed, it is essential to build executive awareness of the necessity for a process improvement program. This requires both managerial and financial commitments.

The objectives of such a process improvement program should be clearly stated and understood, and expressed using measurable process goals. The process improvement program should form part of the organization’s overall strategic business plan.

2. Conduct Assessment

The assessment should be conducted according to a documented process. Assessors must have access to appropriate guidance on how to conduct the assessment and the necessary competence to use the tools.

Each process is assessed by detailed examination. A rating is assigned and validated for each process attribute assessed. In order to provide the basis for repeatability across assessments, the defined set of indicators is used during the assessment to support the assessors’ judgment in rating process attributes.

Objective evidence based on the indicators that support the assessors’ judgment of the ratings are recorded and maintained to provide the basis for verification of the ratings.

3. Initiate Process Improvement

The process improvement program is a project in its own right and planned and managed accordingly. A plan should be produced at the beginning of the program and subsequently used to monitor progress. The plan should include the relevant background, history, and the current status, if possible, expressed in specific and numerical terms. The input derived from the organization’s needs and business goals provide the main requirements for the plan.

The plan should include a preliminary identification of the scope in terms of the boundaries for the program and the processes to be improved. The plan should cover all the process improvement steps, although it may initially give only outline indications of the later stages. It is important to ensure that key roles are clearly identified, adequate resources allocated, appropriate milestones and review points established, and all risks identified and documented in the plan. The plan should also include activities to keep all those affected by the improvement informed of progress.

4. Analyze Assessment Output and Derive Action Plan

Information collected during the assessment, in particular the capability level ratings, the generic practice ratings, and the base practice ratings, is first analyzed, and a plan of action is derived. This consists of the following activities:
• Identify and prioritize improvement areas
• Analyze assessment results
  Analysis of the results provides information about the variability as well as current strengths and weaknesses and indicates opportunities for improvement.
• Analyze the organization’s needs and improvement goals
  The processes and their relationships are analyzed to evaluate which have direct impact on the goals identified. A priority list of processes to be improved is then derived.
• Analyze effectiveness measurements
• Analyze the risks in not achieving improvement goals
  The impact of failing to achieve improvement goals is evaluated in order to understand the urgency and to set the priority of initiatives.
• Analyze risks of improvement action failure
  The potential risks of failure of an improvement action are analyzed to support the definition of priorities and to assure commitment and organizational support.
• List improvement areas
  A prioritized list of improvement areas is provided as a result of analyzing all the factors listed above.
• Define specific improvement goals and set targets
  Targets for improvement should be quantified for each priority area.
• Derive action plan
  A set of actions to improve processes should be developed. Care should be taken to select a set of actions, which support each other in achieving the complete set of goals and targets. It is desirable also to include some improvement actions, which yield clear short-term benefits in order to encourage acceptance of the process improvement program.

5. Implement Improvements

A process improvement action plan is implemented in order to improve the process. Implementation may be simple or complex depending on the contents of the action plan and the characteristics of the organization.

In practice, several process improvement projects will be initiated, each concerned with implementing one or more process improvement actions. Such projects will often cover only initial implementation of improvements. Four main tasks are involved in each process improvement project:

• Operational approach to implementation
Where there are alternative operational approaches to implementation, they should be evaluated and the most suitable selected. It may be possible to implement in small steps through piloting in a selected unit, throughout the whole organization at the same time, or somewhere between these two extremes. Among the factors to consider are costs, time scales, and risks.

- **Detailed implementation planning**
  
  A detailed implementation plan is then developed. The process improvement project may need to carry out a deeper analysis of improvement opportunities than that already carried out. Those implementing the actions and those affected by them should be involved or consulted in developing the plan and in evaluating alternatives in order to draw both on their expertise and enlist their cooperation.

- **Implementing improvement actions**
  
  During this activity, it is critical for successful improvement that due account is taken of human and cultural factors.

- **Monitoring the process improvement project**
  
  The organization’s management should monitor the process improvement project. Records should be kept for use to both confirm the improvements and to improve the process of process improvement.

6. **Confirm Improvements**

Management as well as stakeholders must be involved both to approve the results and to evaluate whether the organization’s needs have been met. If, after improvement actions have been taken, measurements show that process goals and improvement targets have not been achieved, it may be desirable to redefine the project or activity by returning to an appropriate earlier step.

- **Improvement targets**
  
  Current measurements of process effectiveness should be used to confirm achievement of process effectiveness targets. The possibility of having introduced desirable or undesirable side effects should be investigated.

- **Confirm achievement of targets**
  
  A further process assessment should be used to confirm achievement of targets expressed as process capability levels. Where several improvement projects were undertaken, consideration should be given to a reassessment of wider scope to check for potential side effects arising from the parallel improvement actions.

- **Organizational culture**
  
  The effect of the improvements on organizational culture should be reviewed to establish that desired changes have taken place without undesirable side effects.
• Re-evaluate risks

The organization should re-evaluate the risks of using the improved process to confirm that they remain acceptable, and if they are not, determine what further actions are required.

• Re-evaluate cost benefit

The costs and benefits of the improvements may be re-evaluated and compared with earlier estimates made. These results are useful to support planning of subsequent improvement actions.

7. Sustain Improvement Gains

After improvement has been confirmed, the process needs to be sustained at the new level of performance. This requires management to monitor institutionalization of the improved process and to give encouragement when necessary. Responsibilities for monitoring should be defined, as well as how this will be done by using appropriate effectiveness measurements.

If an improved process has been piloted in a restricted area or on a specific project or group of projects, it should be deployed across all areas or projects in the organization where it is applicable. This deployment should be properly planned and the necessary resources assigned to it. The plan should be documented as part of the process improvement project plan or the process improvement program plan as appropriate.

8. Monitor Performance

The performance of the process should be continuously monitored. New process improvement projects should be selected and implemented as part of a continuing process improvement program, since additional improvements are always possible.

• Monitoring performance of the process

The performance of the process should be monitored as it evolves over time. The effectiveness and conformance measures used for this should be chosen to suit the organization’s needs and business goals and should be regularly reviewed for continuing suitability. The risks to the organization and its products from using the process should also be monitored and action taken as risks materialize or become unacceptable.

• Reviewing the process improvement program

Management should review the process improvement program regularly. Further process assessments can be an important component of the continuing improvement program. The extent to which improved processes have been institutionalized should be considered before scheduling further process assessments. It may be more cost-effective to delay assessing a process until improvements have been fully deployed, rather than expend resources assessing a process, which is in transition, when the results can be difficult to interpret.
The bottom line of assessment is making application testing more effective. This is performed by a careful analysis of the results of testing and then taking action to correct identified weaknesses. Facts precede action, and testing in many organizations has suffered from a lack of facts. Once these facts have been determined, action should be taken.

The “measurement first, action second” concept is effective when the measurement process is specific. Measurement must be able to determine the effect of an action. For example, the metric approach fulfills this requirement in that it shows very specific relationships. Using this concept, if action is taken by changing one of the metric variables, the result of that action can be quickly measured.

Changing the variable in one metric can normally be measured by the change in another metric. For example, if the number of defects detected after the system goes operational is higher than desirable, then action should be taken. The action taken might be to increase the number of instructions exercised during testing. Obviously, this increases test cost with the hopeful objective of reducing undetected defects prior to operation. On the other hand, if increasing the number of instructions executed does not reduce the number of defects undetected prior to production, then those resources have not been used effectively and that action should be eliminated and another action tried.

Using the measurement/action approach, the variables can be manipulated until the desired result is achieved. Without the measurement, management can never be sure that intuitive or judgmental actions are effective. The measurement/action approach works and should be followed to improve the test process.

### 2.2.7 Test Process Alignment

In establishing the test environment management must assure that the mission/goals of the test function are aligned to the mission/goals of the organization. For example, if a goal of the organization is to have high customer satisfaction, then the mission/goal of the test function would be to do those activities which lead to high customer satisfaction of the testers’ customers. This may mean that they focus testing more on what the customer needs, as opposed to the defined specifications.

Table 2-2 is an example of how that alignment occurs. This table is a test process alignment map. The objective of the map is to assure that the test processes are aligned with the organizational user testing goals. One axis of the matrix lists the organizational user testing goals, and the other axis lists the test processes. A determination is then made as to whether or not a specific test process contributes to an organizational and/or user testing goal. In this example a check-mark is put in the intersection indicating which test process contributes to which goal.
The purpose of this map is two-fold:

1. To determine that there are adequate processes in place to achieve the goal
2. To determine that there are no goals without processes or processes that do not contribute to defined goals

### 2.2.8 SDLC Methodologies Impact on the Test Process

The Software Development Life Cycle Models as discussed in section 1.6 of Skill Category 1 do not prescribe a specific method for testing. Despite some commentary to the contrary, all the SDLC models embrace the concept of full life cycle testing. When testing is viewed as a life cycle activity, it becomes an integral part of the development process. In other words, as development occurs testing occurs in conjunction with development. For example, when requirements are developed, the testers can perform a requirements review to help evaluate the completeness and correctness of requirements. Note that testers may be supplemented with subject matter experts in some of these tests, such as including users in a requirements review.

Testers testing software developed by a specific software development methodology need to:

1. Understand the methodology
2. Understand the deliverables produced when using that methodology
3. Identify compatible and incompatible test activities associated with the development methodology
4. Customize the software test methodology to effectively test the software based on the specific development methodology used to build the software

<table>
<thead>
<tr>
<th>Software meets requirements</th>
<th>Risk Analysis</th>
<th>Test Planning</th>
<th>Test Data Preparation</th>
<th>Test Reporting</th>
<th>Measurement of Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software meets user needs</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>Software easy to use</td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Five sigma defects in operational software</td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
<td></td>
</tr>
</tbody>
</table>

*Table 2-2 Example of a Test Process Alignment Map*
2.2.9 The Importance of Work Processes

The major purposes for and advantages of having work processes for testers are:

- **Improves communication**

  The standards define the products (or services) to be produced. Not only are the products defined, but also the detailed attributes of each product are defined. This definition attaches names to the products and the attributes of the products. Thus, in a standardized environment when someone says, “a requirements document,” it is clear what that means and what attributes must be attained in order to have requirements defined. In an environment without standards, the communication between workers is impaired, because when one worker says requirements, another worker is probably not certain what that means.

- **Enables knowledge transfer**

  Processes are in fact “expert systems.” They enable the transfer of knowledge from one person to another. Once a procedure has been learned, it can be formalized and all people in the department can perform that procedure with reasonable effort. In an environment in which processes are not well defined, some people may be able to do a job very effectively, and others perform the same job poorly. The formalization of process engineering should raise the productivity of the poor performers in the department and at the same time not hinder the productivity of the effective performers.

- **Improves productivity**

  It is difficult to improve productivity throughout a function without first standardizing how the function is performed. Once it is clear to everyone how the function is performed, they can contribute to improving the process. The key to productivity becomes constant improvement. Since no one person can do everything the best, each shortcut or better method identified by anyone in the organization can be quickly incorporated into the procedures and the benefits gained by everybody.

- **Assists with mastering new technology**

  The processes are designed to help the IT function master new technology. There is a huge cost of learning whenever new technology is introduced. Each individual must invest time and energy into the mastering new technology. Effective processes assist in that technological mastery. It is knowledge transfer for new technology.

- **Reduces defects and cost**

  It costs a lot of money to make defective products. If workers make defects through lack of mastery of technology or using ineffective processes, the organization has to pay for not only making the defect but also searching for and correcting it. Doing it right the first time significantly reduces the cost of doing work.


2.3 Test Environment

Adding to the “role of management” and the “work processes,” the third component of the software testing ecosystem is the Test Environment. The challenge with defining the Test Environment is recognizing that there are a variety of overlapping definitions of the term Test Environment.

In the previous release of the Software Testing Common Body of Knowledge, the Test Environment components were identified as management support for testing, test processes, test tools, and a competent testing team. This might sound familiar as those components are now defined as the software testing ecosystem (this skill category of the STBOK). The review committee working on the STBOK update decided that defining the Test Environment in such broad terms was inconsistent with the more contemporary definition which defines the Test Environment in terms more closely related to the actual testing process. Focusing on a more technical description of a Test Environment still leaves questions about how discrete and separate the test environment will be. Is there a test lab? If there is a test lab, is it brick and mortar or a virtualized test lab?

2.3.1 What is the Test Environment?

The Test Environment can be defined as a collection of hardware and software components configured in such a way as to closely mirror the production environment. The Test Environment must replicate or simulate the actual production environment as closely as possible. The physical setup should include all relevant hardware loaded with server operating system, client operating system, database servers, front end (client side) environment, browsers (if web application), or any other software components required to effectively validate the application under test. Automated testing tools are often considered part of the test environment. (Test tools will be discussed in section 2.4).

2.3.2 Why do We Need a Test Environment?

Why are separate environments needed? Simply stated, you should never make changes to a production environment without first validating that changes, additions, and previously working code has been fully tested (new function and regression tests) and the production entrance criterion has been met. The risk to the application’s users is too significant not to complete this. A test environment increases the tester’s ability to ensure repeatability of the testing process and have better version control of the application under test. Other benefits include the testers’ ability to plan and execute tests without being interfered by or interfering with development activity or the production system.
In the end, the goal of the test environment is to cause the application under test to exhibit true production behavior while being observed and measured outside of its production environment.

2.3.3 Establishing the Test Environment

Establishing a Test Environment requires first understanding what SDLC development framework is used on the project as well as the levels of testing that will be performed in the environment. For instance, the environment for an Agile project will likely be different from a waterfall project. Potential questions to ask about levels of testing include will unit testing be performed in the test environment? If so, are the tools the developers use loaded and available for the test team? Will only functional (system testing) be performed in the environment or will it also be used for acceptance testing?

Understanding the SDLC methodology and what levels of testing will be performed allows for the development of a checklist for building the Test Environment.

2.3.3.1 The Test Environment Checklist

There are many factors that go into the development of a test environment. Listed below are some of the factors to consider:

- What types of tools will be available in the test environment?
  - Test management tools
    - defect tracking
    - test case management
  - Traceability tools
  - Code coverage tools
  - Performance testing tools
  - Regression test tools
  - Specialized tools
    - Mobile application test tools
    - Security test tools

- Hardware configured to match production hardware
  - Network infrastructure configuration
  - Servers including DB servers, application servers, web hosting servers
  - Client side hardware
  - Specialized hardware
- Point-of-Sale devices
- Mobile phones and tablets
- Real-time data acquisition systems

- Software systems
  - Operating environment and versions active on production servers
  - Operating environment and versions active on client devices
  - Operating environment and versions active on specialized hardware
  - Software applications that interface with the application under test
  - Configuration setup of operating environments

- Test data
- Test harnesses
- Object libraries
- Change management protocols
- Test environment backup process
- Test environment security standards
- Software licensing requirements
- Configuration Management
- Virtualization hardware and software
- Lastly, that the responsibility for maintaining environment has been assigned

### 2.3.3.2 Test Labs

The test environment takes on a variety of potential implementations. In many cases, the test environment is a “soft” environment segregated within an organization’s development system allowing for builds to be loaded into the environment and test execution to be launched from the tester’s normal work area. Test labs are another manifestation of the test environment which is more typically viewed as a brick and mortar environment (designated, separated, physical location). Establishing a test lab would include the items on the checklist from section 2.3.3.1.

#### 2.3.3.2.1 Why Setup a Test Lab?

There are many reasons for a dedicated test lab. To decide if a test lab is necessary, the following questions should be asked:

- Can all the necessary components of the system under test be made available to or accessed from a “soft” test environment?
- Are there specialized hardware configurations necessary to execute the tests?
- Will performance or stress testing require dedicated resources to execute?
- Are there security concerns that can be more appropriately addressed in a test lab?
- Does access to the test environment need to be restricted?
2.3.3.2 Test Lab Facility

Building a test lab under the brick and mortar model will require physical space. Considerations that should be made include:

- The size of the space and comfort – Ensure adequate space is set aside with appropriate lighting. The space should not be cramped nor poorly lit. Also, ensure that the necessary office furnishing (desks, chairs, bookshelves, and white boards) are included. Discouraging testers from spending time in the test lab is not the objective.
- Proximity to tester’s work areas – The location of the test lab should be as convenient as possible to those utilizing the space.
- Air conditioning – Most office spaces have A/C set to provide average temperature between 69° and 73° F (21°- 23° C). However, the normal load does not account for a closed environment with a large number of heat producing machines. Ensure the A/C is adequate to cool the space appropriately.
- If a criterion for setting up the test lab relates to security or access control, then appropriate locking mechanisms must be installed.

2.3.3.3 Equipping the Test Lab

With the physical space constructed, the hardware and software necessary to equip the test lab should be installed. If specialized hardware will be required, it must be procured and installed also. The test lab will need to be outfitted as the requirements for the testing dictate.

2.3.3.3 Model Office

The concept of the Model Office is sometimes defined at a high level relating to the overall implementation of a business solution and the efforts of the business and technical stakeholders to successfully plan that transformation. Model Office is also described at a more tactical level, and for the purposes of the STBOK, we will confine our definition as it directly relates to software testing. Model Office is a specialized form of the Test Lab described in 2.3.3.2.

The implementation of and updates to production software impacts the processes and people using the application solution. To test this impact, the software test team, along with business stakeholders and process and system experts, need a replica of the impacted business function. Model office should be an “exact model” defined in both business and technology terms. The key to the implementation of a Model Office is the process and workflow maps diagramming how things happen. This should include the who, what, and when of the business process including manual processing, cycles of business activity, and the related documentation. Like any testing activity, clearly defining the expected outcomes is critical. The development of the Model Office requires a detailed checklist like the one listed for the environment checklist described in 2.3.3.1 above.

Testing performed in the Model Office are generally complete, end-to-end processing of the customer’s request using actual hardware, software, data, and other real attributes. The advantage of testing in a Model Office environment is that all the stakeholders get the
opportunity to see the impact on the organization before the changed system is moved into production.

### 2.3.4 Virtualization

The concept of virtualization (within the IT space) usually refers to running multiple operating systems on a single machine. By its very definition, a virtual “server” is not an actual server but a virtual version of the real thing. A familiar example of virtualization is running a Mac OS on a Windows machine or running Windows on a Mac OS system. Virtualization software allows a computer to run several operating systems at the same time. The obvious advantage of virtualization is the ability to run various operating systems or applications on a single system without the expense of purchasing a native system for each OS.

The reality of setting up a test environment is that the financial costs associated with potentially duplicating all the production systems that would be required to validate all systems under test would be infeasible. Virtualization offers a potential solution. The ability to establish several virtual systems on a single physical machine considerably increases the IT infrastructure flexibility and the efficiency of hardware usage.

There are a number of advantages to using virtualization technologies to create test environments. First, testers can create a number of virtual machines with different configurations allowing for tests to be run on these different configurations which simulate real-world variability of systems. Virtualization allows for the creation of operating environments on equipment that doesn’t normally support that environment allowing test cases to be run from the testers system. Also, virtualization can prevent the damage to a real operating system that a critical defect might inflict. It is also easy to create a backup of the virtual machine so the precise environment that existed when a defect was encountered can be more easily saved and reloaded.

However, there are challenges to using virtualization when setting up a test environment. Not all operating environments or devices can be emulated by a virtual environment. The configuration of a virtual system can be complex and not all systems will support virtualization. There may be equipment incompatibilities or issues with device drivers between the native OS and the virtual OS.

Testing in a virtual environment provides many benefits to the test organization and can provide excellent ROI on test dollars. However, in the end, the final application must be tested in a real operating environment to ensure that the application will work in production.
2.3.5 Control of the Test Environment

Regardless of the approach to establishing the test environment, the responsibility for the ongoing maintenance and control of the test environment rests with the test organization. The policies regarding input/exit criteria and the staging of builds for the test environment must be set by test management and controls put in place to ensure those policies are adhered to.

2.4 Test Tools

The fourth component of the software testing ecosystem is test tools. Once considered optional for the software test organization, test tools have now become a mandatory part of the ecosystem. The ever expanding application feature sets, software complexity, and aggressive deadlines have made comprehensive manual testing economically challenging and the ROI in automation much easier and quicker to realize. Not that implementation of automated tools is easy, but rather the maturity of the automated tool industry has changed the dynamic so that the value of automation has moved it from expensive “shelfware” status to an institutionalized process.

The testing organization should select which testing tools they want used in testing and then incorporate their use into the testing procedures. Tool usage is not optional but mandatory. However, a procedure could include more than one tool and give the tester the option to select the most appropriate one given the testing task to be performed.

As valuable as automated tools are, it is important to remember that a required precursor to implementation of an automated test tool is a mature test process already in place. A great tool will not make a poor process great. More than likely an automated tool deployed in an environment with poor manual processes will only exacerbate the problems.

Equally important is the understanding that the decision to automate and the tool selected should be a result of specific need and careful analysis. There are too many cases of unsuccessful automated tool implementation projects that almost always find a root cause in poor planning and execution of the acquisition process. A detailed tool acquisition process is given in section 2.4.4.

2.4.1 Categories of Test Tools

Test tools can be categorized in a number of ways. Tools can be separated into commercial versus open source (e.g., HP-QTP, Selenium). They can be categorized as specialized or
generalized tools (Silk Test™, Mobile Labs Trust™) or grouped by what test processes they automate (e.g., performance, regressions, defect tracking, etc).

2.4.1.1 Commercial versus Open Source

The software test automation industry today is represented by an array of companies and technologies. Commercial tool companies, some with roots going back to the 1980’s, have provided robust tool sets across many technologies. In contrast to the commercial tool products are the open source test tools. Both commercial and open source tools can provide significant value to the software test organization when intelligent implementation is followed. The most important concept is to understand the needs and from that discern the true cost of ownership when evaluating commercial versus open source options. Note: While the term “commercial” tool is generally accepted as defined within the context of this discussion, “proprietary” would be a more accurate term as there are commercial open source tools on the market as well as proprietary free tools.

2.4.1.1.1 Commercial Test Tools

Commercial automated test tools have been around for many years. Their evolution has followed the evolution of the software development industry. Within the commercial test tools category there are major players represented by a handful of large IT companies and the second tier tools companies which are significant in number and are growing daily.

Shown below are “generalized” statements about both the positive and negative aspects of commercial tools. Each item could reasonably have the words “may be” in front of the statement (e.g., may be easy to use). Characteristics of commercial tools include:

- Positives
  - Maturity of the product
  - Stability of the product
  - Mature implementation processes
  - Ease of use
  - Availability of product support teams
  - Substantial user base
  - Number of testers already familiar with tools
  - Significant third party resources in training and consulting
  - Out-of-the-box implementation with less code customization
  - More efficient code
  - A lower risk recommendation (old saying: “No one ever got fired for buying IBM.”)

- Negatives

2. Silk Test is a registered trademark of the Borland Corporation (a Micro Focus Company).
3. Mobile Labs Trust is a registered trademark of Mobile Labs.
Expensive licensing model
- Require high priced consultants
- Expensive training costs
- Custom modifications require specialized programming skills using knowledge of the proprietary language
- Less flexibility

2.4.1.1.2 Open Source Test Tools

Open Source: “pertaining to or denoting software whose source code is available free of charge to the public to use, copy, modify, sublicense, or distribute.” Open source automated test tools have grown in popularity over the last few years. The high cost of ownership of many commercial tools has helped spur the open source model. While open-source tools have become much more accepted, the quality of open-source software is dependent on the exposure, history, and usage of the tool. Careful investigation and evaluation must be done before considering using an open source tool.

Shown below are “generalized” statements about both the positive and negative aspects of open source tools. Each item could reasonably have the words “may be” in front of the statement (e.g., may be more flexibility).

The positive and negative aspects of open source automated tools include:
- Positives
  - Lower cost of ownership
  - More flexibility
  - Easier to make custom modifications
  - Innovative technology
  - Continuing enhancements through open source community
  - Based on standard programming languages
  - Management see as “faster, cheaper, better”
  - Easier procurement
- Negatives
  - Open source tool could be abandoned
  - No single source of support
  - Test or development organization is not in the business of writing automated tools
  - Fewer testers in marketplace with knowledge of tool
  - Not ready “out-of-the-box”
  - Less readily available training options

2.4.1.1.3 No Free Lunch

The notion that an open source tool is free is a dangerous misconception. While one advantage of open source may be lower cost of ownership, in reality the opportunity costs to the testing and/or development organization to dedicate resources to something for which a COTS (commercial off the shelf) product exists needs to be carefully analyzed. A decision about commercial versus open source should never be first and foremost about the costs but about what is the right tool for the right job. Section 2.4.4 discusses the Tool Acquisition process.

2.4.1.2 Specialized versus Generalized Tools

A debate often occurs regarding the value of specialized tools designed to meet a relatively narrow objective versus a more generalized tool that may have functionality that covers the specific need but at a much higher or less detailed level along with other functionality that may or may not be useful to the acquiring organization. Within the automated test tool industry, the debate is alive and well. Some organizations have developed niche tools that concentrate on a very specific testing need. Generalized testing tools focus on providing a testing experience across a wider swath of the life cycle. In many cases the specialized tools have interfaces to the larger generalized tools which, when coupled together, provide the best of both worlds. Regardless, understanding the test automation need should drive tool acquisition.

2.4.1.3 Categories of Test Tools Grouped by Test Process

There are literally hundreds of tools available for testers. At the most general level, our categorization of tools relates primarily to the test process that is automated by the tool. The most commonly used tools can be grouped into these eight areas:

- **Automated Regression Testing Tools** – Tools that can capture test conditions and results for testing new versions of the software.
- **Defect Management Tools** – Tools that record defects uncovered by testers and then maintain information on those defects until they have been successfully addressed.
- **Performance/Load Testing Tools** – Tools that can “stress” the software. The tools are looking for the ability of the software to process large volumes of data without either losing data, returning data to the users unprocessed, or have significant reductions in performance.
- **Manual Tools** – One of the most effective of all test tools is a simple checklist indicating either items that testers should investigate or to enable testers to ensure they have performed test activities correctly. There are many manual tools such as decision tables, test scripts used to enter test transactions, and checklists for testers to use when performing testing techniques such as reviews and inspections.
- **Traceability Tools** – One of the most frequently used traceability tools is to trace requirements from inception of the project through operations.
- **Code Coverage** – Tools that can indicate the amount of code that has been executed during testing. Some of these tools can also identify non-entrant code.
• **Specialized Tools** – This category includes tools that test GUI, security, and mobile applications.

### 2.4.1.4 Tool Usage Guidelines

Most testing organizations agree that if the following three guidelines are adhered to tool usage will be more effective and efficient.

- **Guideline 1** – Testers should not be permitted to use tools for which they have not received formal training.
- **Guideline 2** – The use of test tools should be incorporated into test processes so that the use of tools is mandatory, not optional.
- **Guideline 3** – Testers should have access to an individual in their organization, or the organization that developed the tool, to answer questions or provide guidance on using the tool.

### 2.4.2 Advantages and Disadvantages of Test Automation

The debate over test automation as a viable part of the software testing ecosystem is over. The ROI has been proved and the majority of test organizations utilize automated tools at some level. Regardless, there still exists advantages and disadvantages of using automated tools and understanding these issues will help the tester use automation more intelligently.

#### 2.4.2.1 Advantages

Advantages of using automated test tools are:

- **Speed** – Test cases generally execute faster when automated. More importantly, different batches of test cases can be executed on multiple computers simultaneously.

- **Reusability** – Assuming automation has been designed and implemented properly, test cases can be run as many times as necessary to test a software release or any future software releases. Over time, this may amount to very significant productivity gains.

- **Increased Coverage** – Testers using automated test tools can easily execute thousands of different complex test cases during every test run providing coverage that is impossible with manual tests.

- **Accuracy** – Manually repeating the same tests over and over inevitably leads to boredom-induced complacency that allows defects that would otherwise be caught to be overlooked. This may lead to testing shortcuts which could also have detrimental effects on the final quality. With automation, test cases are executed with 100% accuracy and repeatability every time.

- **Relentlessness** – Tests can be run, day and night, 24 hours a day potentially delivering the equivalent of the efforts of several full-time testers in the same time period.

- **Simulate Load** – Automated testing can simulate thousands of virtual users interacting with the application under test.
• Efficiency – Automating boring repetitive tasks not only improves employee morale, but also frees up time for staff to pursue other tasks they otherwise could not or would not pursue. Therefore, greater breadth and depth of testing is possible this way.

2.4.2.2 Disadvantages

The disadvantages of using automated test tools are:

• Significant Investment – The investment required to implement test automation is substantial. Test scripts created during the initial automation exercise need to be maintained to keep in synchronization with changes to the relevant application.

• Dependency on Automation Experts – Test automation is largely a technical exercise performed by a skilled automation expert.

• Not as Robust – Automated scripts will only check what has been explicitly included for checking.

• Error Detection – Errors introduced during the automation process are more difficult to detect. Once a test has been fully automated and becomes part of the test regime, human interaction is minimized and errors will come to light only if the automated test itself includes robust error detection routines or the manual tester is actively monitoring every automated test as they execute.

• Cannot Think – Automated tools cannot detect and intercede when unexpected situations arise.

2.4.3 What Should Be Automated

The decision to automate or not automate is based on many factors, most of which have been discussed in this section. Listed below are examples of when automation should be considered and when automation should not be considered.

2.4.3.1 What to Automate

• Regression Tests – Stabilized tests that verify stabilized functionality

• Tests rerun often – Tests that are executed regularly

• Tests that will not expire shortly – Most tests have a finite lifetime during which their automated scripts must recoup the additional cost required for its automation

• Tedious/Boring tests
  o Test with many calculations and number verifications
  o Repetitive tests performing the same operations over and over
  o Tests requiring many performance measurements
  o Just plain boring tests

• Load Tests – When it is necessary to simulate many concurrent users accessing application
2.4.3.2 What Not to Automate

- **Unstable functionality** – Not reliably repeatable
- **Rarely executed tests** – Poor Return-On-Investment
- **Tests that will soon expire** – Poor Return-On-Investment
- **Requiring in-depth business analysis** – Some tests require so much business specific knowledge that it becomes too expensive to include every verification required to make their automated script robust enough to be effective

2.4.4 Tool Acquisition

There are many tools available to support the testing process, from simple checklists to defect tracking tools and automated regression tools. The Test Manager plays a key role in the identification, selection, and acquisition of automated tools. This section’s discussion describes one acquisition process that may be used.

The management of any significant project requires the work be divided into tasks for which completion criteria can be defined. To permit orderly progress of the activities, the introduction of a test tool, and the scheduling of these events must be determined in advance. A general outline for such a schedule is discussed in 2.4.4.1 Sequence of Events to Select Testing Tools. The actual calendar time schedule will depend on many factors, particularly on the time required for procurement of the tool and training. One format used for the tool selection event sequence is consistent with the Critical Path Method (CPM) of project scheduling and can be used to develop the optimum calendar time schedule.

Most of the activities included in the event sequence are obviously necessary, but a few are included specifically to avoid the difficulties encountered when tools are obtained “through the side door” without adequate consideration of the resources required for the effective deployment of the tool and without determination by a responsible manager that the tool will serve a primary need of the organization. A key feature of the proposed approach is that tool acquisition will be initiated only in response to an expressed management goal for software development or for the entire computing function.

Difficulties surrounding the introduction of tools can arise in three areas:

- Organizational obstacles
- Problems arising from the tools
- Obstacles in the computer environment

The individual activities described below, as well as the ordering of the event sequence, are designed to eliminate as many of these difficulties as possible. They are most effective with regard to organizational obstacles and probably least effective with regard to obstacles in the computer environment. The need to involve a responsible management level in the tool introduction has already been mentioned, and this is, indeed, the key provision for avoiding organizational obstacles. “Responsible management” is that level that has the authority to obligate the resources required for the introduction process.
The scope of the resource requirement will become clearer after all introduction activities have been described. Because one criterion for the selection of a tool is the need to commit funds, the management level that can commit funds is hereafter referred to as funding management. In organizations, this may be the project management, functional management, or department management.

The activities associated with the introduction of tools should include the following:

- Identifying the goals to be met by the tool and assigning responsibility for the activities required to meet these goals.
- Approving a detailed tool acquisition plan that defines the resource requirements for procurement and in-house activities.
- Approving the procurement of tools and training, if this is not explicit in the approval of the acquisition plan.
- Determining, after some period of tool use, whether the goals have been met.

The management of the organization that will introduce the tool must overcome additional organizational obstacles. A pitfall that must be avoided is assigning the details of the tool acquisition as a sideline to an individual who carries many other responsibilities. Even in a small software organization (up to 14 testers), it should be possible to make the tool acquisition and implementation process the principal assignment of an experienced individual with adequate professional background. This person is referred to as the Tool Manager. In medium-size organizations (15 to 39 testers), several individuals may be involved in deploying a test tool.

Obstacles arising from the tools themselves are expected to be avoided in the event sequence by a careful, methodical selection of tools. In particular, distinct contributions to the tool selection are specified for test management and the Test Manager.

Test management is assigned responsibility for:

- Identifying tool objectives
- Approving the acquisition plan (it may also require approval by funding management)
- Defining selection criteria
- Making the final selection of the tool or the source

The Test Manager is responsible for the following activities (note that a selection group for the more important test tools may assist the Tool Manager):

- Identifying candidate tools
- Applying the selection criteria (in informal procurement) or preparing the RFP (in formal procurement)
- Preparing a ranked list of tools or sources
- Conducting any detailed evaluations or conference room pilots

Further, the ultimate user of the tool is involved in the recommended event sequence in reviewing either the list of candidate tools or, for formal procurement, the tool requirements.
This distribution of responsibilities reduces the chances of selecting a tool that:

- Does not meet the recognized needs of the organization
- Is difficult to use
- Requires excessive computer resources
- Lacks adequate documentation

The repeated exchange of information required by the process outlined above will also avoid undue emphasis on very short-term objectives that may lead to the selection of a tool based on availability rather than suitability.

### 2.4.4.1 Sequence of Events to Select Testing Tools

The event sequence provides the following help in this area:

1. Employ a methodical process of identifying candidate tools and selecting among these based on established criteria. This will avoid some of the worst pitfalls associated with “borrowing” a tool from an acquaintance or procuring one from the most accessible or persuasive tool vendor.

2. Determine the assignment and training of a Tool Manager who can make minor modifications to both the computer environment and the tool. This is expected to provide relief where there are version-related or release-related incompatibilities with the operating system, or where the memory requirements of the tool exceed the capabilities of the installation. In the latter case, remedies may be provided by removing tool options.

The event sequence described below is conceived as a procedure generally applicable to the introduction of tools. For this reason, a systematic reporting of the experience with the introduction process as well as with the tool is desirable. The evaluation plan and the evaluation report specified in the event sequence support these goals.

### 2.4.4.1.1 Recommended Event Sequence

The event sequence described below is applicable to both small and large IT tool environments. Because of differences in tool requirements, personnel qualifications, and organizational structure, some differences in the content of the individual events will be expected.

The recommended event sequence allows for two procurement methods for bids:

1. Informal procurement (by purchase order)
2. Formal procurement (by a request)

Obviously, the latter is much more time-consuming, but it may lead to the procurement of better or cheaper tools. Acquisition of tools by the purchasing department or for government agencies should follow the formal procurement steps even when there is no procedural
requirement for this. As mentioned above, tool acquisitions, which do not obtain the concurrence of all affected operational elements, frequently do not achieve their objectives.

Some steps are shown which can be combined or eliminated where less formal control is exercised or where plans or modifications required for the introduction of a tool are available from a prior user. The event sequence is intended to cover a wide range of applications. It was constructed with the thought that it is easier for the tool user to eliminate steps than to be confronted with the need for adding some that had not been covered in this section.

### 2.4.4.1.2 Event 1: Goals

The goals should be identified in a format that permits later determination that they have been met (see 2.4.4.1.15 Event 14: Determine if Goals Are Met).

Typical goal statements are:
- Reduce the average test time by one-fifth
- Achieve complete interchangeability of test data sets
- Adhere to an established standard for documentation format

The goals should also identify responsibilities, in particular, the role that project development may have and any coordination with other activities. Where a decentralized management method is employed, the goals may include a not-to-exceed budget and a desired completion date. Once these constraints are specified, funding management may delegate the approval of the acquisition plan to a lower level.

### 2.4.4.1.3 Event 2: Tool Objectives

The goals generated in Event 1 are translated into desired tool features and requirements arising from the development and operating environment. Constraints on tool cost and availability may also be added at this event.

Typical test tool objectives are:
- The tool must run on our existing hardware and operating system platform.
- Only tools that have been in commercial use for at least one year and at no less than N different sites shall be considered.

At this point, the sequence continues with either Event 2: A or Event 2: B.

#### Event 2: A Acquisition Activities for Informal Procurement

**Event 2: A1 – Acquisition Plan**

The acquisition plan communicates the actions of test management both upward and downward. The plan may also be combined with the statement of the tool objectives created in Event 2. The acquisition plan should include:
- Budgets and schedules for subsequent steps in the tool introduction.
- Justification of resource requirements in light of expected benefits.
• Contributions to the introduction expected from other organizations (e.g., the tool itself, modification patches, or training materials).

• Assignment of responsibility for subsequent events within the IT organization, particularly the identification of the Test Manager.

• Minimum tool documentation requirements.

Event 2: A2 – Selection Criteria

The criteria should include ranked or weighted attributes that will support effective utilization of the tool by the user. Typical selection criteria are:

• Accomplishment of specified tool objectives
• Ease of use
• Ease of installation
• Minimum processing time
• Compatibility with other tools
• Low purchase or lease cost
• Documentation, Training, and Support availability

Most of these criteria need to be factored further to permit objective evaluation, but this step may be left up to the individual who does the scoring. Together with the criteria (most of which will normally be capable of a scalar evaluation), constraints, which have been imposed by the preceding events or are generated at this step, should be summarized.

Event 2: A3 – Identify Candidate Tools

This is the first event for which the Test Manager is responsible. The starting point for preparing a listing of candidate tools is a comprehensive tool catalog. A desirable but not mandatory practice is to prepare two lists:

• The first list contains all tools meeting the functional requirements without considering the constraints. For the viable candidates, literature should be requested from the supplier and examined for conformance with the given constraints.

• The second list contains tools that meet both the functional requirements and the constraints. If this list does not have an adequate number of entries, relaxation of some constraints will have to be considered.

Event 2: A4 – User Review of Candidates

The user(s) reviews the list of candidate tools prepared by the Test Manager. Because few users can be expected to be very knowledgeable in the software test tools area, specific questions may need to be raised by the Tool Manager, such as: “Will this tool handle the present file format? Are tool commands consistent with those of the editor? How much training will be required?”
Adequate time should be budgeted for this review and a due date for responses should be indicated. Because the user views this as a far-term task, of lower priority than many immediate obligations, considerable follow-up by the Tool Manager will be required. If tools can be obtained for trial use, or if a demonstration at another facility can be arranged, it will make this step much more significant.

**Event 2: A5 – Score Candidates**

For each of the criteria previously identified, a numerical score is generated on the basis of the following:

- Information obtained from a vendor’s literature
- Demonstration of the tool
- The user’s review
- Observation in a working environment
- Comments of prior users

If weighting factors for the criteria are specified, the score for each criterion is multiplied by the appropriate factor and the sum of the products represents the overall tool score. Where only a ranking of the criteria was provided, the outcome of the scoring may be simply a ranking of each candidate under each criteria heading. Frequently, a single tool is recognized as clearly superior by the end of this process.

**Event 2: A6 – Select Tool**

This decision is reserved for test management in order to provide an objective review of the scoring, and to permit additional factors that were not expressed in the criteria to be taken into consideration. For example, a report might just have been received from another organization to which the selected vendor did not provide adequate service. If the selected tool did not score the highest, the Tool Manager should have an opportunity to review the tool characteristics thoroughly to avoid unexpected installation difficulties.

The tool selection concludes the separate sequence for informal procurement for Event 2: B Acquisition Activities for Formal Procurement. Continue with *Event 3: Procure Tool* on page 2-43.

**Event 2: B Acquisition Activities for Formal Procurement**

**Event 2: B1 – Acquisition Plan**

The plan generated here must include all elements mentioned under Event 2: A1, plus:

- The constraints on the procurement process.
- The detailed requirements for all procurement documents (statement of work, technical and administrative provisions in the request for proposal (RFP), etc.).

The technical requirements document is a formal description of the tool requirements and the constraints under which the tool has to operate. It will utilize much of the material from the acquisition plan but should add enough detail to support a meaningful review by the tool user.

Event 2: B3 – User Review of Requirements

The user reviews the technical requirements for the proposed procurement. As in the case of Event 2: A4 – User Review of Candidates, the user may need to be prompted with pertinent questions, and there should be close management follow-up in order to get a timely response.

Event 2: B4 – RFP Generation

From the Technical Requirements Document and its user comments, the technical portions of the RFP can be generated. Usually these include:

- **Specification**
  This should include applicable documents, a definition of the operating environment, and the quality assurance provisions.

- **Statement of Work**
  This should state any applicable standards for the process by which the tool is generated (e.g., configuration management of the tool) and documentation or test reports to be furnished with the tool. Training and operational support requirements are also identified in the Statement of Work.

- **Proposal Evaluation Criteria and Format Requirements**
  Evaluation criteria are listed in the approximate order of importance. Restrictions on proposal format (major headings, page count, and desired sample outputs) may also be included.

Event 2: B5 – Solicitation of Proposals

Administrative and purchasing personnel carry out this activity. Capability lists of potential sources are maintained by most purchasing organizations. Where the software organization knows of potential bidders, their names should be made known to the procurement office. When responses are received, they are screened for compliance with major legal provisions of the RFP.

Event 2: B6 – Technical Evaluation should be Consistent

Each of the proposals received in response to the RFP is evaluated against the criteria previously established. Failure to meet major technical requirements can lead to outright disqualification of a proposal. Those deemed to be in the “competitive range” are assigned point scores. These point scores are used together with cost and schedule factors that are being separately evaluated by administrative personnel.
**Event 2: B7 – Source Selection**

On the basis of the combined cost, schedule, and technical factors, a source for the tool is selected. If this was not the highest-rated technical proposal, prudent management will require additional reviews by test management and the Tool Manager to determine that it is, indeed, acceptable.

The source selection concludes the separate sequence for formal procurement for 2.4.4.1.3 Event 2: Tool Objectives.

**2.4.4.1.4 Event 3: Procure Tool**

In addition to determining whether the cost of the selected tool is within the approved budget, the procurement process also:

- Considers the adequacy of licensing and other contractual provisions and compliance with the “fine print” associated with all the organization’s procurements.
- Identifies the vendor's responsibility for furnishing the source program, meeting specific test and performance requirements, and tool maintenance.

In informal procurement, a period of trial use may be considered if this has not already taken place under one of the previous events.

If the acquisition plan indicates the need for outside training, the ability of the vendor to supply the training and the cost advantages from combined procurement of tool and training should be investigated. If substantial savings can be realized through simultaneous purchase of tool and training, procurement may be held up until outside training requirements are defined. See 2.4.4.1.7 Event 6: Training Plan.

**2.4.4.1.5 Event 4: Evaluation Plan**

The evaluation plan is based on the goals identified in 2.4.4.1.2 Event 1: Goals and the tool objectives derived from these in 2.4.4.1.3 Event 2: Tool Objectives. It describes how the attainment of these objectives is to be evaluated for the specific tool selected. Typical items to be covered in the plan are:

- Milestones for installation
- Dates
- Performance levels for the initial operational capability and for subsequent enhancements
- Reports and supporting data that address expected improvements in throughput, response time, or turnaround time
- Responsibility for tests, reports, and other actions
- Topics outline of the evaluation report

The procedure for the acceptance test is a part of the evaluation plan, although in major tool procurement it may be a separate document. It lists the detailed steps necessary to test the tool in accordance with the plan to evaluate the interaction of the tool with the computer environment (e.g., adverse effects on throughput), and for generating an acceptance report.
2.4.4.1.6 Event 5: Implementation Plan

The plan will describe the responsibilities and tasks for the implementation of the tool, and the training that will be required. An experienced system programmer, familiar with the current operating system, should do the implementation. Training in the operation and installation of the selected tool in the form of review of documentation, visits to current users of the tool, or training by the vendor must be arranged.

2.4.4.1.7 Event 6: Training Plan

The training plan should first consider the training, if any, provided with the tool, (e.g., documentation, test cases, online diagnostics, Help.) These features may be supplemented by standard training aids supplied by the vendor.

Because of the expense, training sessions at other locations should be considered only when none of the previous categories are available. The number of personnel to receive formal training should be specified in the plan, and adequacy of in-house facilities (number of terminals, computer time, etc.) should be addressed. If training by the tool vendor is desired, this should be identified as early as possible to permit training to be procured with the tool (see 2.4.4.1.4 Event 3: Procure Tool).

User involvement in the preparation of the training plan is highly desirable, and coordination with the user is considered essential. The training plan is normally prepared by the Tool Manager and approved by test management.

Portions of the plan should be furnished to procurement staff if outside personnel or facilities are to be utilized.

2.4.4.1.8 Event 7: Tool Received

The tool is turned over by the procuring organization to the Tool Manager or systems programmer.

2.4.4.1.9 Event 8: Acceptance Test

The Tool Manager and test staff test the tool. This is done as much as possible in an “as received” condition with only those modifications made that are essential for bringing it up on the host computer. A report on the test is issued. After approval by test management, it constitutes the official acceptance or rejection of the test tool.

2.4.4.1.10 Event 9: Orientation

When it’s determined that the tool has been received in a satisfactory condition, test management holds an orientation meeting for all personnel involved in the use of the tool and tool products (reports or listings generated by the tool). The main purpose is to communicate as directly as possible the objectives of the tool use, such as increased throughput or improved legibility of listings.
Highlights of the evaluation plan should also be presented, and any changes in duties associated with the introduction of the tool should be described. Personnel should be reassured that allowance will be made for problems encountered during the introduction and the full benefits of the tool may not make themselves felt for some time.

2.4.4.1.11 Event 10: Modifications

The systems programmer and Tool Manager carry out this step in accordance with the approved plan. It includes modifications of the following:

- **The Tool**
  
  Typical tool modifications involve deletion of unused options, changes in prompts or diagnostics, and other adaptations made for efficient use in the prevailing environment. Documentation of the modifications is an essential part of this event.

- **Documentation**
  
  Vendor literature for the tool is reviewed in detail and is tailored for the planned computer environment and for the tool modifications that have been made. Deleting sections that are not applicable can be just as useful as adding material that is required for the specific programming environment. Unused options shall be clearly marked or removed from the manuals. If there is some resident software for which the tool should not be used (e.g., because of language incompatibility or conflicts in the operating system interface), warning notices should be inserted into the tool manual.

- **Operating system**
  
  In rare cases some modification of the computer operating environment may also be necessary (channel assignments, etc.).

2.4.4.1.12 Event 11: Training

Training is a joint responsibility of the Tool Manager and the tool user. The former is responsible for the content (in accordance with the approved training plan), and the latter should have control over the length and scheduling of sessions.

Training is an excellent opportunity to motivate the user to utilize the tool. The tool user should have the privilege of terminating steps in the training that are not helpful and extending portions that are helpful and need greater depth. Training is not a one-time activity. Retraining or training in the use of additional options after the introductory period is desirable. This also provides an opportunity for users to talk about problems with the tool.

2.4.4.1.13 Event 12: Use in the Operating Environment

The first use of the tool in the operating environment should involve the most qualified test personnel and minimal use of options. The first use should not be on a project with tight schedule constraints. Any difficulties resulting from this use must be resolved before expanded service is initiated. If the first use is successful, use by additional personnel and use of further options may commence.
User comments on training, first use of the tool, and use of extended capabilities are prepared and furnished to the Tool Manager. Desired improvements in the user interface, speed, or format of response and utilization of computer resources are appropriate topics. Formal comments may be solicited shortly after the initial use, after six months, and again after one year.

### 2.4.4.1.14 Event 13: Evaluation Report

The Tool Manager prepares the evaluation report, using the outline generated in 2.4.4.1.5 Event 4: Evaluation Plan. The report should include:

- User comments and observations of the systems programmer
- Whether the general goals and tool objectives were met
- Observations on the installation and use of the tool
- Cooperation received from the vendor in installation or training.
- Any other “lessons learned”
- Tool and host computer modifications
- A section of comments useful to future users of the tool

The report is approved by test management and preferably also by funding management.

### 2.4.4.1.15 Event 14: Determine if Goals Are Met

Funding management receives the evaluation report and determines whether the goals established in 2.4.4.1.5 Event 4: Evaluation Plan have been met. This determination should be in writing and include:

- Attainment of technical objectives
- Adherence to budget and other resource constraints
- Timeliness of the effort
- Cooperation from other departments
- Recommendations for future tool acquisitions

### 2.4.5 Last Thoughts on Tool Acquisition

The right tool, in the right environment, with the right amount of resources dedicated to successful implementation and institutionalization will provide positive ROI for any test organization. It is important to remember that the most important aspect of a testing tool is the process used to acquire that tool. The wrong tool, in the wrong environment, no matter how many resources are dedicated will likely expend financial resources, staff time resources, and in the end provide little if any value to the organization.
2.5 Skilled Team

The fifth component of the software testing ecosystem is a skilled team. Involved management, great processes, and the right environment with good tools will be ineffectual if the skills necessary to “make it happen” are unavailable. There is no substitute for a well-trained team of skilled professionals.

The direct responsibility of both the individual and the organization that employs that individual is to ensure the necessary competencies exist to fulfill the test organization’s objectives. However, the individual has the primary responsibility to ensure that his/her competencies are adequate and current. For example, if a tester today was conducting manual testing of Cobol programs, and that tester had no other skill sets than testing Cobol programs, the probability of long-term employment in testing is minimal. However, if that individual maintains current testing competencies by continually learning new testing tools and techniques, that individual is prepared for new assignments, new job opportunities, and promotions.

2.5.1 Types of Skills

Too often skills are defined purely in academic terms; for example, does the tester know how to create an automated script? In reality the skills necessary to be an effective software tester include much more than technical knowledge to do a task. The capabilities needed to succeed as a test professional include technical skills and also the practical skills including good problem solving capabilities along with “soft” skills. The combination of practical, “soft,” and technical skills define the competent tester needed in today’s challenging IT industry.

2.5.1.1 Practical Skills

In Skill Category 1, the software testing style known as “Exploratory Testing” was described. In that section it stated, “The Exploratory Testing style is quite simple in concept; the tester learns things that together with experience and creativity generates new good tests to run.” The ability to be a creative thinker, to “think outside the box,” and formulate new, better ways to test the application relies not on a “textbook” solution but a cognitive exercise on the part of the tester to solve a problem. This type of skill, while dependent in many ways on the collective lifetime experience of the individual, can be taught and improved over time.

Aligned with the discussion of practical skills is the notion of Heuristics, a term popularized in the software testing context by James Bach and Dr. Cem Kaner. Heuristic refers to experience-based techniques for problem solving, learning, and discovery.

Skills defined in this context are:

- Critical thinking skills – The objective analysis and evaluation of a problem
- Problem solving skills – The process of finding solutions to complex issues
• Inquiring mindset – The habit, curiosity, and courage of asking open-minded questions

2.5.1.2 “Soft” Skills

Soft skills are defined as the personal attributes which enable an individual to interact effectively and harmoniously with other people. Skill Category 3 will describe in depth the communication skills needed within the context of managing the test project. Surveys of software test managers regarding what skills were considered most important for a software tester reveal that the most important skill for a software tester is communication skills.

The ability to communicate, both in writing and verbally, with the various stakeholders in the SDLC is critical. For example, the ability to describe a defect to the development team or to effectively discuss requirements issues in a requirements walkthrough is crucial. Every tester must possess excellent communication skills in order to communicate the issues in an effective and efficient manner.

Soft skills are not confined just to communication skills. Soft skills may be used to describe a person’s Emotional Quotient (EQ). An EQ is defined as the ability to sense, understand, and effectively apply the power and acumen of emotions to facilitate high levels of collaboration and productivity. Software testing does not happen in a vacuum. The software tester will be engaged with many different individuals and groups and the effectiveness of that engagement greatly impacts the quality of the application under test.

2.5.1.3 Technical or Hard Skills

The term “technical skills” is a rather nebulous term. Do we define technical skills to mean “programming capability”? Are technical skills required for white-box testing? Does understanding SQL queries at a detailed level mean a tester has technical skills or does not understanding SQL queries at a detailed level cast aspersions on a testers technical skills? The STBOK defines technical skills as those skills relating directly to the process of software testing. For example, writing a test case would require technical skills. Operating an automated tool would require technical skills. The presence or absence of a particular technical skill would be defined by the needs of the test project. If a project required knowledge of SQL queries, then knowing SQL queries would be a technical skill that needs to already exist or be acquired.

The majority of the content contained in the Software Testing Body of Knowledge relates to technical skills. How to write a test plan, how to perform pairwise testing, or how to create a decision table are skills that are described in the different skill categories.

2.5.2 Business Domain Knowledge

A final category relating to the capability or competency of a professional software tester is domain knowledge of the application under test. The need for a working knowledge of what the application does cannot be minimized.
In section 1.2.3.1 of Skill Category 1, the concept of two software quality gaps was discussed. The first gap, referred to as the producer gap, describes the gap between what was specified and what was delivered. The second gap, known as the customer gap, describes the gap between the customer’s expectations and the product delivered. That section went on to state that a significant role of the software tester is to help close these two gaps.

The practical and soft skills described in section 2.5.1.1 and 2.5.1.2 would be used by the software tester to help close both the producer and customer gap. The technical skills discussed in section 2.5.1.3 would primarily be used to close the producer gap. The domain knowledge of the software tester would be used to close the customer gap.

Domain knowledge provides a number of benefits to the software test organization. They include:

- Testing the delivery of a quality customer experience. Domain knowledge gives the tester the ability to help “fine tune” the user interface, get the look and feel right, test the effectiveness of reports, and ensure that operational bottlenecks are resolved.
- An understanding of what is important to the end user. Often times the IT team has their own perception of what is important in an application. It often tends to be the big issues. However, in many cases critical attributes of a system from the users stand point may be as simple as the number of clicks needed to access a client record.
- Section 1.2.5 of Skill Category 1 described the Software Quality Factors and Criteria. These factors are not business requirements (which typically define what a system does), but rather the quality factors and criteria describe what a system “is” (what makes it a good software system). Among these criteria are such attributes as simplicity, consistency, and operability. These factors would be most effectively evaluated by a tester who has the knowledge about what makes an application successful within the operational environment.

### 2.5.3 Test Competency

Regardless of whether the reference is to practical skills, soft skills, technical skills, or domain knowledge, there are two dimensions of competency: knowing what to do and doing it. The first is the skill sets possessed by the individual, and the latter refers to the performance of the individual using those skills in real-world application.

#### 2.5.3.1 Individual Capability Road Map

Developing the capabilities of individuals within the software testing organization does not just happen. A dedicated effort, which includes the efforts of both the test organization and the individual, must be made. Working together, a road map for tester capability development can be created which includes both development of skills and the capability to effectively utilize those skills in daily work.
2.5.3.2  Measuring the Competency of Software Testers

Figure 2-6 is typical of how a software testing organization may measure an individual tester’s competency. This type of chart would be developed by the Human Resource department to be used in performance appraisals. Based on the competency assessment in that performance appraisal, raises, and promotions are determined.

*Figure 2-6  Measuring the Competency of Software Test*
Managing the Test Project

Software testing is a project with almost all the same attributes as a software development project. Software testing involves project planning, project staffing, scheduling and budgeting, communicating, assigning and monitoring work, and ensuring that changes to the project plan are incorporated into the test plan.

<table>
<thead>
<tr>
<th>Skill Category</th>
<th>3</th>
</tr>
</thead>
</table>

3.1 Test Administration

Test administration is managing the affairs of software testing. It assures that what is needed to test effectively for a software project will be available for the testing assignment at the correct time. This section addresses:

- Test Planning – assessing the software application risks, and then developing a plan to determine if the software minimizes those risks
- Estimation – understanding the size and effort of the project
  - Budgeting – the resources to accomplish the test objectives
  - Scheduling – dividing the test project into accountable pieces and establishing start and completion dates
- Resourcing – obtaining the testers to achieve the plan
- Customization of the test process – determining whether or not the standard test process is adequate for a specific test project, and if not, customizing the test process for the project

Logically the test plan would be developed prior to the test schedule and budget. However, testers may be assigned a budget and then build a test plan and schedule that can be accomplished within the allocated budget. The discussion in this skill category will include planning, scheduling and budgeting as independent topics, although they are all related.

The six key tasks for test project administration are the planning, estimation which includes budgeting and scheduling, staffing, and customization of the test process if needed. The plan defines the steps of testing, estimation provides clarity on the magnitude of the tasks ahead, the schedule determines the date testing is to be started and completed, the budget determines the amount of resources that can be used for testing and test process customization assures the test process will accomplish the test objectives.

Because testing is part of a system development project, its plan, budget, and schedule cannot be developed independently of the overall software development plan, budget, and schedule. The build component and the test component of a software development project need to be integrated. In addition, these plans, schedules, and budgets may be dependent on available resources from other organizational units, such as users.

Each of the six items listed should be developed by a process. These are the processes for developing test planning, budgeting, scheduling, resourcing, and test process customization. The results from those processes should be updated throughout the execution of the software testing tasks. As conditions change so must the plan, budget, schedule, and test process. These are interrelated variables and changing one has a direct impact on the other three.

### 3.1.1 Test Planning

Test planning is a major component of software testing. It is covered in detail in Skill Category 5. That category provides the process for developing the test plan and provides a standard for defining the components of a test plan.

### 3.1.2 Estimation

As part of the test planning process, the test organization must develop estimates for budget and scheduling of software testing processes.

At the heart of any estimate is the need to understand the size of the “object” being estimated. It would be unrealistic to ask a building contractor to estimate the cost of building a house without providing the information necessary to understand the size of the structure. Likewise, to estimate how much time it will take to test an application before the application exists can be daunting. As Yogi Berra, the famous New York Yankees catcher, once said, “It’s tough to make predictions, especially about the future.” Unfortunately, predictions must be made for
both budget and schedule and a first step in this process is understanding the probable size of
the application.
There is no one correct way to develop an estimate of application size. Some IT organizations use
judgment and experience to create estimates; others use automated tools. The following
discussions represent some of the estimation processes but not necessarily the only ones. The tester
needs to be familiar with the general concept of estimation and then use those processes as
necessary.

To a great degree, test management bases an estimate on expert judgment. By judgment we mean
the expertise of the test manager or person responsible for the estimation. Internal factors within the
organization and external factors (such as the economy and client requests) always affect the
project. This is where risk analysis and estimation meet. Estimation involves risk at all levels.

Factors that influence estimation include but are not limited to:

- Development life cycle model used
- Requirements
- Past data
- Organization culture
- Selection of suitable estimation technique
- Personal experience
- Resources available
- Tools available

Remember, by definition an estimate means something that can change, and it will. It is a
basis on which decisions can be made. For this reason the test manager must continually
monitor the estimates and revise them as necessary. The importance of monitoring will vary
depending on the SDLC model and the life cycle phase of that the selected model.

3.1.2.1 Top-Down Estimation

The assumptions in the Top-Down Estimation approach are that
software has its own complexity and difficulty in design and
implementation. Project management uses the Top-Down technique
since it generates an overall estimate based on the initial
knowledge. It is used at the initial stages of the project and is based
on similar projects. Past data plays an important role in this form of
estimation.

3.1.2.1.1 Analogous and Parametric Techniques

Within the higher level approach of Top-Down Estimation is the
process of analogous estimating. The analogy model is a non-
algorithmic costing model that estimates the size, effort, or cost of a
project by relating it to another similar completed project.
Analogous estimating takes the actual time and/or cost of a historical project as a basis for the current project. The time cost of the historical project is applied to the current project taking into account the scope of the current project, its size, and other known variables.

Parametric Modeling uses a mathematical model based on known parameters to predict cost/schedule of a test project. The parameters in the model can vary based on the type of project. For example, a parameter can be “years of experience of the test analyst” or “years of experience with a programming language of the programmer.” Source Lines of Code, Function Point Analysis and COCOMOII are parametric techniques.

### 3.1.2.1.2 Source Lines of Code (SLOC)

When describing software size, the most common methodology has been counting the source lines of code (thousand sloc = ksloc) written for the application. However, SLOC is a resultant measure. It is a measure that is ultimately obtained after the application has been written or, at the very least, estimated with significant knowledge of the application’s structure. As a method of size estimation for maintenance or enhancements to an existing application, the usefulness of SLOC is much better. It does not, however, provide direct insights into application size when the SLOC count does not exist. One method in estimating SLOC is to break the application or potential application down into modules or logical groups. Regardless of the method, once a logical breakdown of the potential application is made the programmers can then give estimates on the SLOC based on their experience with other similar modules or applications.

### 3.1.2.1.3 Function Point Analysis (FPA)

Another sizing method is Function Point Analysis. In FPA, the program’s functionality is measured by the number of ways it must interact with the users. Function points (FP) are counted first by estimating the number of external inputs, external interface files, external outputs, external queries, and logical internal tables. These five raw types are defined as:

- **External inputs** are largely data entry screens. If a screen contains a tabbed notebook or similar design, each tab counts as a separate external input.
- **External interface files** are file-based inputs or outputs. Each record format within the file, or, in the case of XML, each data object type would count as a separate interface file even if residing in the same physical file.
- **External outputs** are reports.
- **External queries** are message or external function-based communication into or out of the application.
- **Logical internal tables** are the number of tables in the database, assuming the database was third normal form or better.
Function Point Conversion Factors

<table>
<thead>
<tr>
<th>Raw Type</th>
<th>Function Point Conversion Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>External inputs</td>
<td>4</td>
</tr>
<tr>
<td>External interface files</td>
<td>7</td>
</tr>
<tr>
<td>External outputs</td>
<td>5</td>
</tr>
<tr>
<td>External queries</td>
<td>4</td>
</tr>
<tr>
<td>Logical internal tables</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 3-1 Function Point Conversion Factors

To convert from the raw value types counted to actual function points, the raw type counts are multiplied by the conversion factors shown in Table 3-1 above.

There are several ISO standards relating to Function Point Analysis. They include:

- FiSMA: ISO/IEC 29881:2008 Information technology – software and systems engineering – FiSMA 1.1 functional size measurement method
- NESMA: ISO/IEC 24570:2005 Software engineering – NESMA function size measurement method version 2.1 – Definitions and counting guidelines for the application of Function Point Analysis

Object Oriented Function Point (OOFP) and Weighted Micro Function Points are further variants on the FPA process.

The accuracy of function point counting depends on a number of factors including the functional domain of the applications, the process maturity of the developing organization, and the extent of use of the FPA.

Converting FP to SLOC – Backfiring

Many organizations utilize SLOC as the historical measurement; therefore, to use other historical project data to predict new project items like budget and schedule would require function points to be converted to SLOC for use. Backfiring is a term used to define the process of converting from function points to an equivalent number of SLOC. Caper Jones was a pioneer in this area studying the relationship between programming language, function points, and lines of code. He published standard efficiency values used to make the conversion from function points to language dependent SLOC for different programming languages.
3.1.2.1.4  Use Case Points (UCP)

A well accepted and institutionalized process in many organizations is the use of Use Cases to describe the application product. A derivative of the Use Cases method is the estimation technique known as Use Case Points. Use Case Points are similar to Function Points and are used to estimate the size of a project.

To determine the number of use case points in a project, the following information is collected:

• the number and complexity of the use cases in the system
• the number and complexity of the actors on the system
• various non-functional requirements such as portability and performance (see section 1.2.6)
• the environment in which the project will be developed

The process weighs the complexity of the use cases and actors and then adjusts their combined weight to reflect the influence of the nonfunctional and environmental factors.

3.1.2.1.5  Cost Models within the Top-Down Approach

The discussion relative to estimating the size of a software project was purposed to provide the best data to estimate the cost (to establish a budget) of a project. The best recognized software development cost model is the Constructive Cost Model II. COCOMO®II is an enhancement over the original COCOMO® model. COCOMO®II extends the capability of the model to include a wider collection of techniques and technologies. It provides support for object-oriented software, business software, software created via spiral or evolutionary development models and software using COTS application utilities.

COCOMO® and COCOMO®II combine estimated SLOC, five Scale Drivers, and fifteen Cost Factors to determine cost. The Scale Drivers determine the exponent used in the Effort Equation.

The 5 Scale Drivers are:

• Precedentedness
• Development Flexibility
• Architecture/Risk Resolution
• Team Cohesion
• Process Maturity

The 15 Cost Factors include subjective assessments of product, hardware, personnel and project attributes.

• Product attributes
  • Required software reliability
  • Size of application database
• Complexity of the product
  • Hardware attributes
    • Run-time performance constraints
    • Memory constraints
    • Volatility of the virtual machine environment
    • Required turnabout time
  • Personnel attributes
    • Analyst capability
    • Software engineering capability
    • Applications experience
    • Virtual machine experience
    • Programming language experience
  • Project attributes
    • Use of software tools
    • Application of software engineering methods
    • Required development schedule

### 3.1.2.2 Bottom-Up Estimation

The Top-Down approaches that have been described in the previous sections are most often used during the project initiation or early life cycle phases. They tend to be more “rough” estimates. By contrast, Bottom-Up estimating is the most accurate approach but can be very time-consuming and expensive. In this technique, the cost of each single activity is determined with the greatest level of detail at the bottom level and then rolls up to calculate the total project cost.

#### 3.1.2.2.1 Work Breakdown Structure

A Work Breakdown Structure (WBS) groups project components into deliverable and accountable pieces. A project level WBS displays and defines the product to be developed by hardware, software, and other elements and relates them to each other and to the product. The aim is to have measurable components for which people (team members, subject matter experts and specialists) can be held accountable. It is created during the proposal stage.

The WBS defines the total project. It is a product-oriented family tree composed of hardware elements, software elements, and service elements. The WBS relates project elements or work scope definitions to each other and to the end product. It is not an organization chart of company personnel.

There is a misconception that a WBS should be immediately broken down to the smallest level. In actuality, it is only during subsequent detail planning that the WBS is broken down
into measurable components which will then be further broken down into smaller and achievable components assigned to team members. As is the case in any of the estimation approaches, expert judgment is an important component.

A work breakdown structure can also be applied more specifically to testing within the SDLC. Figure 3-1 shows a simple WBS.

![Example of a Work Breakdown Structure](image)

**3.1.2.3 Estimation Methods in Agile Framework**

The nature of the agile approach to software development creates new challenges for developing viable estimates of size and effort. The flow of an agile project changes the dynamics of estimating since it requires estimates within the iterative cycle of agile. It is important to understand that within an agile project everyone estimates their own tasks; there is no estimating expert who estimates everything for everyone. Estimating should be done at the beginning of each iteration and estimates reevaluated daily. The feedback loop to the agile team and customer is critical. Story Points, Planning Poker, and Triangulation are methods used in estimation within the agile environment.

**3.1.2.3.1 Story Points**

Story Points, like Use Case Points described in section 3.1.2.1.4, are not a measure of the time needed to complete a feature but a measurement of a feature’s size relative to other features. Story Points are an analogous method in that the objective is to compare the sizes of features to other stories and reference stories. The reference story (analogy), is an example of a story which relates well to the story under consideration. A new story, when compared to the reference stories, can then be sized. Understanding size and the team’s velocity help plan the stories in a given sprint and ultimately predict releases (aka, estimates of effort or time).
3.1.2.3.2 Planning Poker (a form of Wideband Delphi, see section 3.1.2.4)

Planning Poker is most often thought of in the context of the Agile Development Framework. Planning Poker is a consensus-based technique designed to remove the cognitive bias of anchoring. Anchoring is when the first estimate number mentioned sets a precedent for subsequent estimates.

In Planning Poker, members of the group make estimates by playing numbered cards face-down to the table, instead of saying them aloud. The cards are revealed, and the estimates are then discussed.

![Planning Poker Cards](image)

Figure 3-2 Planning Poker Decks are commercially available

The steps in Planning Poker are (Agile used for this example):

- To start an estimating poker session, the product owner or customer reads an agile user story or describes a feature to the estimators who should include everyone on the team.
- The estimators can ask questions as needed.
- Each estimator holds a deck of Planning Poker cards with values like 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, and 89 (the Fibonacci sequence), or other similar numbering system.
- The values represent the number of story points, ideal days, or other unit in which the team estimates.
- Each team member chooses a card based on what they think is the estimate for the story discussed and places it facedown.
- The cards are revealed and the estimators discuss their estimates until a consensus is reached.

3.1.2.3.3 Triangulation

Story Triangulation is a form of estimation by analogy. After the first few estimates have been made, they are verified by relating them to each other. There is no single baseline or universal reference. The estimate for each new story will be compared against an assortment of those
stories that have already been estimated; this is the triangulation. For example, to decide if a story should be estimated at five story points it can be compared with a three story point story and an eight story point story. If the story seems a bit larger than the three point story and a bit smaller than the eight point story then five story points is a good estimate.

3.1.2.4 **Wideband Delphi**

The Delphi technique, which is named after the ancient oracle of Apollo in Greece, is a method for the controlled exchange of information within a group. It provides a formal, structured procedure for the exchange of opinion, which means that it can be used for estimating. Some of the features of the technique are:

- It allows feedback of information to group members
- It allows individuals some opportunity to assess group views
- It allows some opportunity for individuals to make their opinions known to the group
- It provides relative anonymity for individuals

The steps in a Wideband Delphi process are:

1. **Kickoff Meeting** – At least three people discuss the source documents and project, as well as the units of estimation.

2. **Estimation** – Each person creates three estimates (using their preferred method): most likely, optimistic, and pessimistic case.

3. **Meeting** – Each estimator gives their estimates to the facilitator who displays them with averages (owners of the estimates may not be revealed if it is desired to reduce the influence of personality or seniority). Each estimator discusses the insights, problems, and assumptions.

4. **Repeat steps 2 and 3 at least once to get iterative estimation refinement** – Let the feedback drive adaptation and improvement.

Calculate the final numbers using the averages from the final cycle.

\[
\text{Estimate} = \frac{(4 \times \text{most likely}) + \text{optimistic} + \text{pessimistic}}{6}
\]

\[
\text{Likely Deviation} = \frac{\text{Pessimistic} - \text{Optimistic}}{6}
\]

The Wideband Delphi technique can be used in conjunction with other estimation methods.
3.1.2.5 Test Specific Estimation Approaches

All the processes described in this section directly relate to the process of estimating test effort and indirectly budget and schedule. There are some additional methods used to estimate the effort of test processes specifically. Common practices for estimating test effort include:

- Analogous percentage method
- Test point analysis
- Development ratio
- Test conditions analysis

3.1.2.5.1 Analogous Percentage Method

A common method for estimating test effort is to calculate the test estimate as a percentage of previous test efforts using a predicted size factor (SF) (e.g., SLOC or FPA). This approach requires that good measurements for SF and that the actual test effort on previous projects be recorded and retained. Using the SF, the new project size estimate is used to calculate the new test effort estimate. In other words, the test effort required would be a direct percentage of the test effort from a similar sized previously completed project based on the project’s SF.

For example, if a previous project with a SF of 1000 function points required 100 person hours of testing effort (TE), the percentage of testing effort is calculated as:

\[
\text{Test Factor (TF)} = \frac{\text{TE}}{\text{SF}} = \frac{100}{1000} = 0.10
\]

If the current project has been estimated SF of 1500 function points, the testing effort could be calculated as:

\[
\text{Estimated Testing Effort (ETE)} = \text{SF} \times \text{TF} = 1500 \times 0.10 = 150 \text{ person hours}
\]

3.1.2.5.2 Test Point Analysis

Another method for estimating test effort is known as Test Point Analysis (TPA). TPA calculates test effort based on size (derived from FPA), strategy (as defined by system components and quality characteristics to be tested and the coverage of testing), and productivity (the amount of time needed to perform a given volume of testing work). The elements of size and strategy determine the volume of testing work to be undertaken and are expressed in test points. Multiplying the number of test points by the productivity results in the estimated test effort in hours.

Size as a function of Function Points is weighted by the following factors:

- User-significance is an expression of the importance that the user attaches to a given function relative to the other system functions. A rating of 3 (low) to 12 (high) is assigned.
• Usage intensity is defined as the frequency with which a certain function is processed by the users and the size of the user group that uses the function. As with user-importance, the usage-intensity is being determined at a user-function level. A rating of 3 to 12 is assigned here also.

• Complexity relates to the number of test conditions. A higher number of test conditions translates to a greater volume of testing.

• Interfacing relates to the number of interconnections to a function. This can be determined by the number of data sets maintained by a function and the number of other functions, which make use of those data sets.

• Uniformity is the extent to which the information system contains similarly structured functions.

Dynamic quality characteristics are an additional factor in the calculation of Test Points. The two components to consider are:

• Explicit characteristics include the important characteristics: functionality, security, suitability, performance, and portability.

• Implicit characteristics are not standard and vary from project to project. They might include user friendliness, efficiency, performance, and maintainability.

Test Points are then calculated as a function of the size, weighted by the additional factors, and the dynamic quality characteristics. Now productivity, which has two important aspects—environment and productivity factors—must be considered. Environmental factors to consider include tools, test environments, and availability of testware to name a few. Productivity is dependent on knowledge, for example, how many senior people are on the team and other like factors.

3.1.2.5.3 Development Ratio Method

The development ratio method derives the test effort, in this case represented by the number of testers, as a function of number of software developers planned for the development effort adjusted for the type and complexity of the software application. The number of developers would include those involved in analysis, design, development, and unit testing. The estimate is dynamic as it may change as the life cycle progresses.

The impact of the type of the application is represented in Table 3-2.

<table>
<thead>
<tr>
<th>Product Type</th>
<th>Number of Developers</th>
<th>Ratio of Developers to Testers</th>
<th>Number of Testers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commercial Product (Large Market)</td>
<td>30</td>
<td>3:2</td>
<td>20</td>
</tr>
<tr>
<td>Commercial Product (Small Market)</td>
<td>30</td>
<td>3:1</td>
<td>10</td>
</tr>
</tbody>
</table>
Analysis of past projects can be used to develop a similar table to Table 3-2 which can then be used for subsequent estimation tasks.

### 3.1.2.5.4 Test Condition Analysis (TCA)

Estimating test effort based on the number of test conditions planned for the project is another accepted test estimation process. Like most of the other methodologies described, TCA requires a historical database of actual project data and is most accurate when combined with other estimation methods. Software size methods like SLOC and FPA can be used in conjunction with TCA. For TCA to produce accurate estimates it is essential that the organization has highly structured test processes with tasks and time tracked and measured. Identifying test conditions will be discussed in considerable detail in Skill Category 7.

Measurements for hours expended on test conditions should include:

- Defining test conditions
- Designing test cases
- Developing test procedures/scripts
- Executing the tests
- Reporting defects & status

The following steps can be used in TCA:

1. Estimate number of test conditions for new project based on requirements and historical project size values
2. Calculate historical average number of test conditions and hours spent, from two or more similar projects
3. Determine historical number of hours per condition

<table>
<thead>
<tr>
<th>Product Type</th>
<th>Number of Developers</th>
<th>Ratio of Developers to Testers</th>
<th>Number of Testers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development &amp; Heavy COTS Integration</td>
<td>30</td>
<td>4:1</td>
<td>7</td>
</tr>
<tr>
<td>Government (Internal) Application Development</td>
<td>30</td>
<td>5:1</td>
<td>6</td>
</tr>
<tr>
<td>Corporate (Internal) Application Development</td>
<td>30</td>
<td>4:1</td>
<td>7</td>
</tr>
</tbody>
</table>
a. \# person hours spent/ \# conditions

4. Calculate test effort for new project

5. A. Determine test effort based on standard work week (effort = 1 resource, contiguous time)
   Ex. Hours required / standard work week hours
       or
   B. Determine number of testers required based on project schedule
   Ex. Hours required / \# of project test hour duration
       or
   C. Determine test duration based on number of testers available
   Ex. Hours required / \# testers available

### 3.1.3 Developing a Budget for Testing

A budget is a plan for specifying how resources, especially time or money, will be allocated or spent during a particular period.

Section 3.1.2 described a number of the methodologies for estimating the size and effort required for a software development project and by logical extension the efforts required to test the application. An estimate is, however, an ‘educated guess.’ The methods previously described almost universally required some level of expert judgment. The ability, based on experience, to compare historical figures with current project variables to make time and effort predictions is Expert Judgment.

A budget, by contrast, is not an estimate, it is a plan that utilizes the results of those estimating processes to allocate test and monetary resources for the test project. The testing budget, regardless of the effort and precision used to create it, still has, as its foundation, the earlier estimates. Each component that went into creating the budget requires as much precision as is realistic; otherwise, it is wasted effort and useless to help manage the test project.

A budget or cost baseline, once established, should not be changed unless approved by the project stakeholders as it is used to track variances against the planned cost throughout the application life cycle.

### 3.1.4 Scheduling

A schedule is a calendar-based breakdown of tasks and deliverables. It helps the project manager and project leader manage the project within the time frame and keep track of current, as well as future problems. A Work Breakdown Structure (as described in section
3.1.2.2.1) helps to define the activities at a broader level, such as who will do the activities, but planning is not complete until a resource and time is attached to each activity. In simple terms, scheduling answers these questions:

- What tasks will be done?
- Who will do them?
- When will they do them?

Scheduling is dependent on staff availability. Sometimes there is a need to add or redistribute staff to complete the test project earlier as per business need or to prevent the schedule from slipping. A few situations that can change the availability of staff members are:

- Someone falls sick
- Someone takes a holiday
- Someone slips behind schedule
- One staff member is used for two projects

The process of adjusting a schedule based on staff constraints is called resource leveling. Resource leveling involves accounting for the availability of staff when scheduling tasks. Resource leveling will help in efficient use of the staff. Resource leveling is also used for optimization. While resource leveling optimizes available staff, it does not ensure that all the staff needed to accomplish the project objectives will be available at the time they are required.

Once a schedule is made, it is possible that during certain phases some staff will be idle whereas at a peak time, those same staff members will be paid overtime to complete the task. This could happen as a result of delays by the development team deploying planned application builds into the test environment. It is best to plan for such occurrences as they will invariably happen.

Status reports are a major input to the schedule. Scheduling revolves around monitoring the work progress versus work scheduled. A few advantages of scheduling are:

- Once a schedule is made, it gives a clear idea to the team and the management of the roles and responsibility for each task
- It enables tracking
- It allows the project manager the opportunity to take corrective action

### 3.1.5 Resourcing

Ideally, staffing would be done by identifying the needed skills and then acquiring members of the test project who possess those skills. It is not necessary for every member for the test team to possess all the skills, but in total the team should have all the needed skills. In some IT organizations, management assigns the testers and no determination is made as to whether the team possesses all the needed skills. In that case, it is important for the test manager to document the needed skills and the skills available from the team members. Gaps in needed
skills may be supplemented by such individuals assigned to the test project on a short-term basis or by training the assigned resources.

The recommended test project staffing matrix is illustrated in Table 3-3. This matrix shows that the test project has identified the needed skills. In this case they need the planning, test data generation skills, and skills in using tools X and Y. The matrix shows there are four potential candidates for assignment to that project. Assume that only two are needed for testing, the test manager would then attempt to get the two that in total had all the four needed skills.

If the test team does not possess the necessary skills, it is the responsibility of the test manager to teach those individuals the needed skills. This training can be on-the-job training, formal classroom training, or e-learning training.

<table>
<thead>
<tr>
<th>Staff</th>
<th>Planning</th>
<th>Test Data Generation</th>
<th>Tool X</th>
<th>Tool Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

*Table 3-3 Test Project Staffing Matrix*

### 3.1.5.1 Test Team Approaches

The following four different approaches can be used to build a test team:

1. Developers become the Test Team
2. Independent IT Test Team
3. Non-IT Test Team
4. Combination Test Team

#### 3.1.5.1.1 Developers Become the Test Team

In this approach the members of the project development team become the members of the test team. In most instances, the systems development project leader is the test team project leader. It is not necessary to have all of the development team members participate on the test team, although there is no reason why they would not participate. It is important that each member of the test team be primarily responsible for testing a different member’s work. The objective of the team is to establish a test process that is independent of the people who developed the particular part of the project being tested.
The advantage of the developers test team approach is that it minimizes the cost of the test team. The project is already responsible for testing, so using project members on the test team is merely an alternate method for conducting the tests. Testing using the developers test team approach not only trains the project people in good test methods, but also cross-trains them in other parts of the project. The developers test team approach uses those people in testing who are most knowledgeable about the project.

The disadvantage of the developers test team approach is the need for ensuring that the project team allocates appropriate time for testing. In addition, the project team may lack team members who believe that the project solution is incorrect and thus find it difficult to challenge the project assumptions.

### 3.1.5.1.2 Independent Test Team

Testing performed by IT personnel independently of the project does not relieve the project personnel of responsibility for the correctness of the application system. Independent testing is designed to provide a different perspective to testing in order to provide extra assurance of the application. Independent testing normally occurs after the project team has performed the testing they deem necessary (i.e., unit testing). Frequently, the system development team verifies that the system structure is correct and the independent test team verifies that the system satisfies user requirements.

The advantage of an independent test team is the independent perspective they bring to the project. The team is comprised of professionals who have specialized in the area of testing.

The disadvantage of independent test team is the additional cost that may be required to establish and administer a testing function. Also, the development team may place too much reliance on the test team and thus fail to perform adequate testing themselves, resulting in overburdening the professional testers.

### 3.1.5.1.3 Non-IT Test Team

Groups external to the information services department can perform testing. The three most common groups that test application systems are users, consultants, and outsource test organization. These groups represent the organizational needs and test on behalf of the organization.

The advantage of a non-IT test team is that they provide an independent view and at the same time can offer an independent assessment. Loyalty to report unfavorable results only to the information services department does not restrict the non-IT group. The non-IT group has greater ability to act and to cause action to occur once problems are detected than does a group within an information technology department.

The disadvantage of non-IT testing is the cost of the test. Generally, these groups are not familiar with the application and must first learn the application and then learn how to test within the organization.
3.1.5.1.4 Combination Test Team

Any or all of the above groups can participate on a test team. A combination team can be drawn together to meet specific testing needs. For example, if the project had significant financial implications, an auditor could be added to the test team; if it had communication concerns a communication consultant could be added.

The advantage of drawing on multiple skills for the test team is to enable a multi-disciplined approach to testing. In other words, the skills and backgrounds of individuals from different disciplines can be used in the test process. For some of the test participants, particularly users, it can be an educational experience to make them aware of both the system and the potential pitfalls in the system. In addition, a combination test team has greater clout in approving, disapproving, or modifying the application system based upon the test.

The disadvantage of the combination test team is the cost associated with assembling and administering the test team. It also may pose some scheduling problems determining when the tests will occur. Finally, the diverse backgrounds of the test team may make the determination of a mutually acceptable test approach difficult.

3.1.6 Customization of the Test Process

Skill Category 2 discussed assessing the overall test process with software development project goals and implementation processes. This assessment is to ensure that the test process will accomplish the test objectives, or that the test process will need some customization to accomplish the test objectives. Some of the characteristics of software development that may cause customization of the test process are:

- Release cycle schedules
- Software development methodology
- User schedules
- Project status reporting
- Interfacing with other projects
- Interfacing with enterprise-wide databases

Test process customization can occur in many ways, but ideally the customization process is incorporated into the test processes, primarily the test planning process. If not incorporated into the test process, customization should occur as a task for the test manager to perform.

The customization process may include any of the following:

- Adding new test tasks
- Deleting test tasks currently in the test process
- Including or omitting the use of test tools
- Supplementing skills of assigned testers to ensure the tasks in the test process can be executed correctly
3.2 Test Supervision

Test supervision relates to the direction of involved parties, and oversight of work tasks to ensure that the test plan is completed in an effective and efficient manner. Supervision is a combination of the supervisor possessing the skill sets needed to supervise, and the tasks that contribute to successful supervision.

There are literally thousands of books written on how to supervise work. There is no one best way on how to supervise a subordinate. However, most of the recommended approaches to supervision include the following:

- Communication skills
  The ability of the supervisor to effectively communicate the needed direction information, and resolution of potential impediments to completing the testing tasks effectively and efficiently.

- Negotiation and complaint resolution skills
  Some specific skills needed to make a supervisor effective, like resolving complaints, using good judgment, and knowing how to provide constructive criticism.

- Project relationships
  Developing effective working relationships with the test project stakeholders.

- Motivation, Mentoring, and Recognition
  Encouraging individuals to do their best, supporting individuals in the performance of their work tasks, and rewarding individuals for effectively completing those tasks.

3.2.1 Communication Skills

Most studies show that the single most important skill possessed by a supervisor is the ability to communicate. Those supervisors who are effective communicators are almost always effective supervisors.

There are four categories of communication skills that are important for supervisors:

- Written and oral communication
- Listening Skills
- Interviewing Skills
- Analyzing Skills

3.2.1.1 Written and Oral Communication

An important aspect of supervision is the ability to communicate with other parties. An effective way to quickly improve the ability to communicate is to view every communication...
opportunity as making a proposal to another person. The proposal can be made orally, or it can be in a written document.

### 3.2.1.2 Listening Skills

Throughout school, students are taught the importance of speaking, reading, writing, and arithmetic, but rarely is much emphasis placed on listening. The shift in society from industrial production to information management emphasizes the need for good listening skills. This is particularly true in the practice of software testing – oral communication (which includes listening) is rated as the number-one skill for the tester.

Some facts about listening include:

- Many Fortune 500 companies complain about their workers' listening skills.
- Listening is the first language skill that we develop as children; however, it is rarely taught as a skill. Thus, in learning to listen, we may pick up bad habits.
- Listening is the most frequently used form of communication.
- Listening is the major vehicle for learning in the classroom.
- Salespeople often lose sales because they believe talking is more important than listening.

It is also important to understand why people do not listen. People do not listen for one or more of the following reasons:

- They are distracted by mobile devices (texting, email).
- They are impatient and have other stimuli to respond to, such as random thoughts going through their mind.
- They are too busy rehearsing what they will say next, in response to someone else.
- External stimuli, for example, an airplane flying overhead, diverts their attention.
- They lack the motivation and responsibility required of a good listener.
- The speaker’s topic is not of interest to them.

The listener must be aware of these detriments to good listening, so they can recognize them and devote extra attention to listening.

### 3.2.1.2.1 The 3-Step Listening Process

The listening process involves three separate steps:

1. Hearing the speaker
2. Attending to the speaker
3. Understanding the speaker.

The practice of listening requires these three steps to occur concurrently. Mastering each of these steps will help improve listening abilities.

**Step 1: Hearing the Speaker**

Hearing the speaker requires an understanding of the five channels of communication incorporated into speech. Much of listening occurs beyond merely hearing the words. There are five channels through which a speaker delivers information to his/her audience:

- Information Channel - The speaker’s subject.
- Verbal Channel - The words used by the speaker.
- Vocal Channel - The tone of voice associated with the various words.
- Body Channel - The body movements and gestures associated with the information being conveyed.
- Graphic Channel - The pictures, charts, etc. that the speaker uses to emphasize or illustrate the material being discussed.

Speakers normally use the information, verbal, vocal, and body channels in speaking. In some instances, they also use the graphic channel. Listening requires that there is a meeting of the minds on the information channel. Speakers sometimes skip around to different subjects, making it easy to lose the subject being covered on the information channel. In “Step 2: Attending to the Speaker,” we discuss the importance of feedback to confirm the subject being covered on the information channel.

The verbal channel includes the choice of words used to present information; the vocal and body channels modify or emphasize the importance of those words. For example, the words in the verbal channel may be, “John says he can do it.” However, the tone of the vocal channel might indicate that John cannot do it, or the use of a thumbs-down body channel signal will also indicate that John cannot do it.

Hearing the speaker involves an awareness of all five channels. To master the hearing step, you must pay attention to all five channels. If you miss one or more of the channels, you will not hear what the person is saying. For example, if you are only paying partial attention to the speaker when the words, “John can do it” are stated, you may hear that John can do it, while the speaker indicated that John could not do it.

**Step 2: Attending to the Speaker**

Attending to the speaker is sometimes referred to as being an active listener. Devote your full attention to the speaker to confirm that what you heard is what the speaker intended you to hear. You must first understand yourself and your situation. You must evaluate your motivation for wanting to listen to this speaker. If the subject is important to you, but the speaker is boring, it will require significantly more effort on your part to be a good listener.

The most important part of attending to the speaker is establishing an active listening ability. Active listening involves a lot of response and dynamics. Some people view the listening
process as a passive skill where you sit back and let the other person talk. This is fine for hearing the speaker, but not for confirming what the speaker has said. Feedback is very important to the listening process, particularly in this step. Feedback can be a nonverbal response, such as nodding your head, or a verbal response such as a question or a statement of confirmation.

It is very important to send the right type of feedback to the speaker. The wrong type of feedback not only doesn’t confirm what the speaker said, but can also reduce or terminate the listening process. It is very irritating to a speaker who is providing information to have the listener stray from the subject. For example, the speaker might be describing a quality problem, and the listener changes the subject and asks where the speaker is going to have lunch that day.

Some suggestions to help in attending to the speaker are:

• Free your mind of all other thoughts and concentrate exclusively on the speaker's communication.
• Maintain eye contact with the speaker for approximately 80 percent of the time.
• Provide continuous feedback to the speaker.
• Periodically restate what you heard the speaker say, and ask the speaker to confirm the intent of the information spoken.
• Move periodically to the understanding step to ensure that the information passed has been adequately understood.

**Step 3 - Understanding the Speaker**

There are five types of listening. While people can listen several different ways concurrently, normally listening is limited to one of the five types. The type chosen will have an impact on the ability to understand what the speaker is saying. When one has deciphered the information channel (i.e., what the subject is) and related the importance of that subject to the audience, listening must be adjusted to ensure that the message is understood.

The five types of listening and their impact on understanding are:

• **Type 1: Discriminative Listening**
  Directed at selecting specific pieces of information and not the entire communication. For example, one may be listening to determine if an individual did a specific step in the performance of a task. To get this, listen more to the nonverbal expressions rather than the verbal channel.

• **Type 2: Comprehensive Listening**
  Designed to get a complete message with minimal distortion. This type of listening requires a lot of feedback and summarization to fully understand what the speaker is communicating. This type of listening is normally done in fact gathering.

• **Type 3: Therapeutic Listening**
  The listener is sympathetic to the speaker's point of view. During this type of listening, the listener will show a lot of empathy for the speaker's situation. It is very helpful to
use this type of listening when you want to gain the speaker's confidence and understand the reasons why a particular act was performed or event occurred, as opposed to comprehensive listening where you want to find out what has happened.

• Type 4: Critical Listening
  The listener is performing an analysis of what the speaker said. This is most important when it is felt that the speaker is not in complete control of the situation, or does not know the complete facts of a situation. Thus, the audience uses this type of understanding to piece together what the speaker is saying with what has been learned from other speakers or other investigation.

• Type 5: Appreciative or Enjoyment Listening
  One automatically switches to this type of listening when it is perceived as a funny situation or an explanatory example will be given of a situation. This listening type helps understand real-world situations.

One must establish which type of understanding is wanted and then listen from that perspective.

### 3.2.1.3 Interviewing Skills

A software tester will use interviewing skills for many different purposes. The obvious one is interviewing an individual for the job of a software tester, or to be assigned to a specific software project. However, interviewing skills are also used for gathering data for test purposes. The tester may interview a user/customer to better understand how their job is performed, the tester may need to interview project development personnel to understand the structure and function of the software systems, and a tester may need to interview a subject matter expert such as an auditor to better understand the attributes of an effective system of internal control.

The primary purpose of interviewing is fact-finding. A secondary purpose is to convey information to the individual being interviewed. Interviewing involves oral communication, it involves listening skills, and it involves fact-finding. Oral communication and listening skills have been discussed previously in this section.

Fact-finding is a process of identifying facts which is a statement of a condition. In other words, a fact is some attribute of a condition that is agreed by involved parties to be correct. A fact could be the result of a test.

A finding is identifying a difference between what is and what should be. To obtain a finding one must know what the condition of an event should be. It is for this reason one talks about testable requirements which pre-define what the processing result should be.

If a processing result should be that individuals are paid time and a half over 40 hours of work, and a test result showed that individuals were not paid time and a half over 40 hours, that
would be a fact. The finding would be the difference, that is, that people should have been paid time and a half but they were not paid time and a half.

When documenting a finding it should include:

- A fact – tells why the difference between what is and what should be.
- Cause – tells the reasons for the deviation. Identification of cause is necessary as a basis for corrective action.
- Significance – how important the difference is in the context of the testing assignment.

### 3.2.1.4 Analyzing Skills

The first question asked after receiving a finding is: “What should I do about it?” The answer to that question is a recommendation. A recommendation suggests the action that should be taken to resolve a finding.

Findings and recommendations are two important parts of communication. The finding states what has happened, and the recommendation states what to do about it. Time spent carefully constructing recommendations is normally rewarded by increased acceptance of the recommendations.

Developing recommendations requires analysis. Unfortunately the effects of poor analysis are not as apparent as those of poor grammar or spelling. Poor analysis, however, is more destructive to the review process. Analysis relies on facts and inferences. The recipient of the recommendation has questions about how and what occurred. These are best answered by the facts, but the question why, the most important question, must be answered by inference or by conclusions based on facts. The individual developing a recommendation is aware that his/her conclusion is a judgment based on a preponderance of evidence and seldom an absolute determination.

Analysis is an essential part of the software tester’s job. A simple recitation of facts, no matter how solid the facts are, creates questions in the mind of the recipient. When recommendations are made, the recipient asks other questions, such as, “How adequate are the criteria backing the recommendation?” “Will the recommendation cause more problems, or cost more, than the current method?” “How sound is the analysis?” “How effective is the recommendation?” Sharing and exploring both facts and analysis helps to establish the value of the recommendation for the recipient.

Recommendations are based upon findings using the analysis process. Analysis permits the findings and supporting background information to be subjected to a challenging process in order to develop recommendations. The value of the recommendation is normally related to the thoroughness of the analysis process.

The analysis process is illustrated in Figure 3-3. The figure shows that the problems plus the analysis produce recommendations. The problems or findings are normally a combination of facts, opinions, and circumstances. Analysis, on the other hand, is a scientific process used to produce a result or recommendation.
Figure 3-3  The Analysis Process

There are four general methods used for analysis which are:

- **Arrangements**
  
The facts, opinions, and circumstances are arranged to enable relations and patterns to be shown between the facts. The relationship can be used to demonstrate cause and effect as well as correlations. For example, if the facts were arranged to show that there was a direct correlation between extent of training and number of errors, a recommendation could be built on that correlation.

  A simple method to arrange and rearrange facts is to code them using a coding method. Normally, any one fact will have several different codes. For example, an error condition might be coded as follows:

  - Input data entry error
  - Computer system error
  - Accounts receivable error

  The facts and opinions can then be arranged and rearranged in a variety of sequences to show patterns and relationships between the information.

- **Analogy**
  
  Using the analogy method, one situation is compared with or contrasted to another. This makes heavy use of the reviewer’s judgment and experience. The reviewer, drawing upon his/her experience, utilizes the similarity between situations in an effort to capitalize on previous situations and recommendations which are applicable to the current situation.

- **Logic**
The reviewer can use inductive or deductive logic to develop a recommendation. Using inductive logic, the argument moves from facts to a generalization. The generalization then becomes the situation that needs to be addressed by the recommendation. Using deductive logic, the main idea is stated and then supported by the facts. Using this approach, the recommendation is obvious and only needs to be justified by the facts in the situation.

- Principles

The reviewer can rely upon good business practices and principles. These principles dictate the best method to accomplish tasks. When it can be determined from the analysis process that a good business practice or principle has been violated, and thus caused a problem, the recommendation is the reinstatement of the practice or principle. For example, if the problem is diagnosed as high maintenance costs and the analysis process shows that the principle of “formal systems documentation” has not been followed, the recommendation would be to document the system using a formal documentation.

3.2.2 Negotiation and Complaint Resolution Skills

As is the case with many activities, negotiating a resolution to a complaint is a skill that is learned through practice. In many situations it is prudent to document the steps taken toward resolution and the actions that are appropriate to every level of the negotiation.

3.2.2.1 Negotiation

Conflict can be defined as a breakdown in the decision-making process. A decision acceptable to all cannot be made among the alternate positions available. Understanding the root cause of the conflict is the first step in resolving the conflict. Negotiation is the means by which the conflict will be resolved. The sources of conflict are listed and defined in Table 3-4.

<table>
<thead>
<tr>
<th>Sources of Conflict</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Priorities</td>
<td>Views of project participants differ over sequence and importance of activities and tasks.</td>
</tr>
<tr>
<td>Administrative Procedures</td>
<td>Managerial and administration-oriented conflicts over how the project will be managed.</td>
</tr>
<tr>
<td>Technical Opinions and Performance Trade-Offs</td>
<td>Disagreements over technical issues, performance specifications, and technical trade-offs.</td>
</tr>
</tbody>
</table>
Table 3-4  Conflict in Project Environment

In determining the root cause of the conflict, a supervisor needs to use all of the communication skills. These will help determine the root cause of the conflict. Once the root cause has been identified, a decision-making process to eliminate the problem can commence. However, when a conflict is interpersonal, it may be more difficult to find the root cause, and thus more difficult to resolve the conflict. Conflict resolution is a subset of conflict management. Conflicts are usually solved in one of these ways:

- **Forcing**
  Conflict is resolved when one party is successful in achieving its own interests at the expense of the other party’s interest through the use of high relative power. This is a win-lose approach.

- **Withdrawal**
  Conflict is resolved when one party attempts to satisfy the concerns of others by neglecting its own interests or goals. This is a lose-win approach.

- **Smoothing**
  An unassertive approach – Both parties neglect the concerns involved by sidestepping the issue, postponing the conflict, or choosing not to deal with it.

- **Compromise**
  An intermediate approach – Partial satisfaction is sought for both parties through a “middle ground” position that reflects mutual sacrifice. Compromise evokes thoughts of giving up something, therefore earning the name “lose-lose.”

- **Problem-solving**
  Cooperative mode – Attempts to satisfy the interests of both parties. In terms of process, this is generally accomplished through identification of “interests” and freeing the process from initial positions. Once interests are identified, the process moves into a phase of generating creative alternatives designed to satisfy identified interests (criteria).

The conflict resolution methods of withdrawal and smoothing may temporarily address the conflict but fail to resolve the root cause. If the conflict resolution approach is withdrawing, one party loses and one party wins. If the conflict resolution is smoothing, both parties lose.
The conflict resolution methods of forcing, compromising and problem-solving resolve the conflict and reach a decision. However, in forcing the decision, one party wins and another party loses. Compromising only to provide resolution may mean that both parties lose. The win-win conflict resolution process is problem-solving. Negotiations in this conflict resolution approach will assure that the most important interests of both parties are achieved so that each party feels they won.

### 3.2.2.2 Resolving Complaints

Research shows that complaints must be resolved within four minutes. Within that time, you should be communicating the solution of the problem to the customer. In his book, Contact: The First Four Minutes, Dr. Leonard Zunin, a human relations consultant, states that unless you have satisfied your customer within four minutes, they will give up on you. They will sense that you have not accepted the urgency of their problem and that you are not the one to solve their problem.

If you make your customer's problem your problem – in other words, you sincerely want to resolve his or her complaint – then you need to execute the following process.

#### 3.2.2.2.1 The 4-Step Complaint-Resolution Process

**Step 1: Get On Your Customer’s Wavelength**

You cannot begin to resolve your customer’s complaint until you show your customer your concern for his/her problem. You need to:

- Get on the same physical wavelength. Establish a position for mutual discussion. If your customer is standing, you stand. If you want your customer to sit, ask the customer to sit first, and then you sit.
- Show interest in your customer’s problem. Give your customer your undivided attention. Comments to a secretary or receptionist, such as, “Do not interrupt us,” will show sincere interest.
- Physically display interest. Assume a body position, gestures, and tone of voice that show concern to your customer.
- React positively to you customer’s concern. Show empathy for your customer’s complaint. For example, if the customer indicates you have caused great inconvenience to their staff, apologize.

**Step 2: Get the Facts**

You cannot deal with your customer's problem until you know what it is. Do not deal with the symptoms; deal with the problem. An angry person is more likely to tell you symptoms than the real problem. To find out the facts, you need to do the following:

- Ask probing questions. A few examples are:
  - “Give me an example of the problem.”
  - “Do you have any samples of defective products?”
  - “Explain more about what you mean.”
  - “Where did you get that piece of information?”
• Take detailed notes. Write down names, amounts in question, dates or times at which events happened, and the specific product or service where problems occurred.

• Observe feelings and attitudes. In many cases the problem may be more emotional than factual. However, you will need to deal with the emotions. Find out how a person feels about what has happened; find out what his/her colleagues or boss feels about the problem.

• Listen carefully to what is being said. Try to listen through the words to find out what the real problem is.

Step 3: Establish and Initiate an Action Program

Even if you believe that the complaint is not reasonable, you still need to take action. The action will serve two purposes: it will determine the validity of the facts and it will pacify the complainer. When taking action, you need to do the following:

• If an error has been made, admit it. Admit the error if you are responsible for the error. Do not minimize the seriousness of the error if it is serious. Not only admit it, but also apologize for it.

• Negotiate a satisfactory resolution with your customer. Only the customer knows what is satisfactory. Even if the solution is to conduct an investigation, it is an appropriate action if it is satisfactory to your customer.

• State the solution and get agreement from your customer. After you have agreed on what action to take, repeat it back to your customer and confirm agreement.

• Take action. Whatever you agreed to do, do it, and do it immediately. Just as it is important to begin communicating a solution within four minutes, it is equally important to resolve the action quickly.

Step 4: Follow Up with Your Customer

When the action you agreed upon has been taken, follow up with your customer to determine satisfaction. Just because you believe the problem has been solved, it is not logical to assume that your customer also agrees. The problem may be a difference of opinion about the solution. Words sometimes do not convey exactly what we mean. If your customer is happy with the resolution, the complaint has been finished. If your customer is not happy, more work has to be done, and you should go back to Step 2: Get the Facts, and try again.

3.2.2.3 Judgment

Judgment is a decision made by an individual. In sports the individual making the judgment is a referee, in law it is a judge. However, using judgment can apply to any activity. Judgment is normally a decision based on three criteria which are:

• Fact
  Indisputable evidence regarding a specific situation.

• Standards
What the facts in the situation should be. Having the facts and the standards, the individual making the judgment knows what has happened, and what should have happened.

• Experience

An individual’s involvement in similar situations, and using that experience to select the best decision which will have the minimum negative impact on the activity. However, if the standard is an objective standard, for example an athlete steps out of bounds then judgment reinforces the standard. However if the standard is subjective such as judging artistic impression in ice skating, then experience will play a much greater role in arriving at a decision.

3.2.2.4 Providing Constructive Criticism

In giving constructive criticism, the following tactics should be incorporated:

• Do it Privately

Criticism should be given on a one-on-one basis. Only the individual being criticized should be aware that criticism is occurring. It is best done in a private location. It is more effective if it is done in a neutral location, for example, in a conference room or while taking someone to lunch, rather than in the boss' office.

• Have the Facts

General statements of undesired performance are not very helpful. For example, statements such as "That proposal is not clear, fix it" or "Your program does not make best use of the language or technology" leave people feeling confused and helpless. Before criticizing someone’s performance, have specific items that are causing the deficiency or undesirable performance documented.

• Be Prepared to Help the Individual Improve His/Her Performance

It is not good enough to ask the individual to “fix it.” One must be prepared to help fix it. Be prepared to train the subordinate in the area of deficiency. For example, in a proposal, indicate that a return-on-investment calculation was not made; or if a program failed to use the language properly, state specifically how it should and should not be used. You should not leave an individual feeling that they have performed poorly or unsure as to how to correct that performance.

• Be Specific on Expectations

Be sure your subordinate knows exactly what is expected from him or her now and in the future. The expectations should be as clear as possible so there can be no confusion. Again, in a proposal, indicate that a return-on-investment calculation is expected to be included in all proposals. Most people will try to do what they are expected to do—if they know what those expectations are.

• Follow a Specific Process in Giving Criticism
Managing the Test Project

The specific process that is recommended is:

- State the positive first. Before criticizing indicate what is good about their performance. Again, be as specific as possible in the things in the statements.
- Indicate the deficiencies with products or services produced by the individual. Never criticize the individual, only the work performed by the individual. For example, never indicate that an individual is disorganized; indicate that a report is disorganized. People can accept criticism of their products and services; they have great difficulty when you attack their personal work ethic.
- Get agreement that there is a problem. The individual being criticized must agree there is a problem before proper corrective action can be taken. Avoid accepting agreement just because you are the boss; probe the need for improvement with the subordinate until you actually feel there is agreement that improvement can be achieved. For example, if you believe a report or program is disorganized, get agreement from the individual that the report or program is in fact disorganized.
- Ask the subordinate for advice on how to improve their performance. Always try to get the employee to propose what needs to be done. If the employee’s suggestion is consistent with what you have decided is a realistic method of improvement, you have finished the process.
- If the subordinate is unable to solve the problem, suggest the course of action that you had already determined before performing the actual criticism.
- Make a specific “contract” regarding what will happen after the session. Be very specific in what you expect, and when and where you expect it. If the employee is uncertain how to do it, the “contract” should include your participation, as a vehicle to ensure what will happen.
- One last recommendation for criticism:
  Avoid making threats about what will happen if the performance does not change. This will not cause any positive behavior change to occur and normally produces negative behavior. Leave the individual with the assumption that he or she has the capability for improvement, and that you know he or she will improve.

3.2.3 Project Relationships

The software tester is providing a service to those having a vested interest in the success of a software project. What is important about the relationships with the stakeholders are:

- The project relationships are defined
- The roles of each party and the relationships are defined
- The importance of the relationship to the success of the project is defined
- The influence that a party can have on software testing is defined

An approach used by many organizations to document relationships is a project relationship chart illustrated in Figure 3-4.
Figure 3-4  Project Relationship Chart

This chart is an example of a relationship chart and is constructed as follows:

1. Define the stakeholders
   All those individuals who have a stake in the success of the project, and thus must have a relationship to the project need to be defined. In the example above, we defined the tester, the user, the software developer, IT management and acceptance testers.

2. Indicate the influence one stakeholder has on another stakeholder
   This chart uses a scale of “1”-to-“3” for influence:
   - 1 meaning a stakeholder has a significant influence over another stakeholder
   - 2 meaning a major influence
   - 3 meaning a minor influence
   Note that the lines show the direction of influence and the significance of the influence.

The purpose of this project relationship chart is to ensure that the tester has clearly identified which relationships are important to the success of software testing, and that the relationships will be developed and maintained during the software testing project.
3.2.4 Motivation, Mentoring, and Recognition

An important aspect of a supervisor’s job is to motivate, mentor, and recognize the testers in his or her organization.

3.2.4.1 Motivation

Motivation has sometimes been defined as getting individuals to do work tasks they do not want to do or to perform those work tasks in a more efficient or effective manner.

Experience has shown that the motivation approach that works best is positive motivation. In other words don’t attempt to motivate by fear or threats such as “no raises” or “termination.”

Different people are motivated by different things. IBM at one time had a policy of supervisors asking their subordinates how they would like to be rewarded for a successful project. It is important to recognize that what motivates people is highly individualized.

The four most common motivators are:

1. Personal challenge – A job task which will challenge the individual’s competency and capabilities.
2. Respect – Treating the individual as a professional.
3. Rewards – Some tangible thing that an individual will receive if they meet specific goals/objectives.
4. Recognition – Publicizing the value contributed by the individual among peers and management.

3.2.4.2 Mentoring

Mentoring is helping or supporting an individual in a non-supervisory capacity. Mentors can be peers, subordinates, or superiors. What is important is that the mentor does not have a managerial relationship to the mentored individual when performing the task of mentoring.

Mentoring can occur in any of the following three areas:

- Career counseling – Discussing career opportunities and assisting individuals in accomplishing their career objectives.
- Work tasks – Helping individuals achieve work tasks by either imparting the necessary skills or working with an individual in completing a job task.
- Professional advancement – Helping an individual achieve professional goals such as becoming a certified software tester (CSTE).

The only benefit a mentor receives for becoming a mentor is the satisfaction of helping another person succeed.
3.2.4.3 Recognition

Employees are recognized at the end of each pay period by being given a paycheck. However, the motivation of employees can be increased by other recognition methods. People like to be recognized for the contribution they make to a project.

The only key concept in this part of supervision is that recognition is important. However, recognition should not have a significant monetary value because obtaining that financial recognition may cause individuals to circumvent controls and good practices.

Some of the recognitions that have been used successfully within software testing are:
- Recognition by an IT manager at a formal IT meeting
- Group luncheons/group celebrations
- Tokens of appreciation such as a coupon for dinner out or a movie
- Time off if completing a task involved an excess of work hours
- Lunch with the boss

3.3 Test Leadership

All test managers are part manager and part leader. Most software test managers will spend most of their time managing and only a part of the time leading. However as testing moves into new areas such as testing to determine whether user success criteria have been achieved, the software test manager becomes more of a leader than a manager.

In discussing leadership, we will address these areas:
- Chairing meetings
- Team building
- Quality Management Organizational Structure
- Code of ethics

3.3.1 Chairing Meetings

Many IT staff members spend almost one half of their day in meetings. Meetings can be both productive and non-productive depending on how they are organized, run, and how meeting decisions implemented. The software project manager in chairing meetings must be more of a leader than a manager.

The following guidelines on conducting meetings are common to most of the books and manuals on how to run an effective meeting. These guidelines are:
- Specific objectives to accomplish at the meeting must be defined
- Those having a stake in the potential decisions need to be represented at the meeting
• An agenda for the meeting, plus any background data, must be distributed to the attendees prior to the meeting allowing enough time for the individuals to prepare for the meeting discussions
• Rules for running the meeting need to be established such as Robert’s Rules of Order
• The individual chairing the meeting must ensure that all present have an equal opportunity to express their opinions
• A consensus process should be used to develop conclusions and actions to be taken, as a result of the meeting
• Specific responsibilities should be assigned to complete the actions
• Minutes of the meeting should be disseminated to the attendees within a reasonable period of time after the meeting concludes

### 3.3.2 Team Building

Much has been written about organization’s loyalty to its employees and employee loyalty to organizations. R.M. Kanter stated that, “New loyalty is not to the boss or to the company, but to projects that actualize the mission and offer challenge, growth, and credit for results.” What this tells the project leader is that team building needs to focus on the challenge, growth and credit an individual can achieve from working on a specific project.

There are a myriad of books on team building. The objective of this discussion is not to duplicate what is available, but to focus on components of team building that are directed more at software teams, than traditional implementation teams. These components are: team development, team member interaction, team ethics, and team rewards.

#### 3.3.2.1 Team Development

There are seven guidelines that are helpful in developing compatibility and motivation of a software project team:

1. **Communicate the vision, objectives, and goals of the project.**

   A software professional wants to know what the project is trying to accomplish. The vision indicates why the project is undertaken, the goals and objectives indicate what the project is to achieve. For example, the vision of a bank commercial loan software project might be to increase profitability. This specific objective might be to provide the loan officer the information needed to make a good loan decision.

2. **Define roles and responsibilities of team members.**

   Software projects, unlike non-software projects, have roles which are heavily people dependent and project scope dependent. It’s important for professional staff to have those roles and responsibilities clearly defined. The staffing matrix described in an earlier part of this skill category would define those roles and responsibilities.
3. Empower team members to manage their responsibilities.

Empowerment is a major motivator for professional people. Many of the agile concepts relate to empowerment. In other words, enable people to perform the tasks in the most efficient and effective manner. This helps eliminate barriers that increase costs and the project schedule.

4. Hold team members accountable for their assigned responsibilities in the team process.

Team members need to have their work tasks well defined and then be held accountable for completing those work tasks. Managerial practices indicate that this process works best when individuals accept responsibility for performing tasks. Thus, having the Project Manager work individually with team members to assign tasks they agree to perform, and then hold those individuals accountable for completing those tasks is an effective managerial practice.

5. Ensure that all the required skills are present on the team.

Projects cannot be completed successfully if the team members lack the skills to complete the project. It is not necessary for every team member to have all the needed skills, but the team in total needs the skills. The staffing matrix helps assure that the appropriate skills exist within the project team.

6. Provide the necessary technical and team training.

If the team lacks technical and team skills, the project manager should provide that training. Technical skills include the skills necessary to design and build the software. Team skills cover consensus building and conflict resolution.

7. Award successes and celebrate achievements.

Establishing goals and objectives provides the basis for rewards and celebrations. While it’s appropriate to reward and celebrate individual achievements, the team building necessitates team goals and team celebrations. These can be centered around milestones accomplished, as well as scoring high on customer satisfaction surveys.

3.3.2.2 Team Member Interaction

The following guidelines have proven effective in building an effective and cohesive team:

- Know communication and work preference styles of staff and ensure that the team complements those communication and work preference styles.
- Set clear, measurable work requirement standards.
- Delegate authority to staff members that empowers them to perform the tasks in the manner they deem most effective and efficient.
- Exact responsibility and accountability for team members for completing their work tasks in an effective efficient manner with high quality work products.
- Give immediate and objective feedback to team members on the performance of their individual and team tasks.
• Communicate, communicate, communicate! Be certain that all team members know about any event that may impact team performance.

### 3.3.2.3 Team Ethics

The accounting and other corporate scandals during the past few years have undermined the public’s confidence in corporations to act in an ethical manner, and to report accurate accounting data. In the U.S. these scandals resulted in the passage of the Sarbanes-Oxley Act which made unethical and improper accounting a criminal act and subjected the corporate management to jail sentences. Corporate ethics at all levels of an organization are important.

The following six attributes are associated with ethical team behavior:

1. Customer relations that are truthful and fair to all parties
   
   Ethical customer relations means treating customer/user personnel with integrity; not promising unachievable results; and informing customer/users of problems that could have a negative impact on delivery or software performance; and striving to fully understand the user’s true processing needs.

2. Protecting company property
   
   The team should not undertake any action that would have a negative impact on the protection of company property or subject that property to loss.

3. Compliance with company policies
   
   The team should be knowledgeable in company policies, considerate of those policies when making team decisions and taking only those actions that meet both the letter and intent of company policies.

4. Integrity of information
   
   The team should strive to ensure that information they produce is reliable and valid, and that the information is conveyed to the appropriate stakeholders on a timely basis.

5. Attendance
   
   Except for valid reasons, the team should be in attendance during normal work hours, be prompt for meetings, and during work hours devote their effort to performing work tasks.

6. Redefine standards of quality
   
   The team should be knowledgeable in the quality standards for team deliverables, work in a manner that is conducive to meeting those quality standards, and if they cannot be met, inform the appropriate stakeholders of the lack of quality of team deliverables.
3.3.2.4 Team Rewards

Over the years, organizations have focused their reward system on individual performance. However, focusing only on individual rewards may undermine team cooperation and performance. If teams are the means by which work is performed, and software project development teams are the means for building software, then team rewards need to be incorporated into the organization’s reward system.

There is no generally accepted approach for rewarding teams. The following reward systems have proven effective in organizations.

- Team celebrations
  At the conclusion of meeting team milestones and objectives, the team celebrates as a group at events such as luncheons, sporting activities, and other off-site events.

- Team financial rewards
  Teams are given a cash reward for meeting an objective or milestone and then the team splits the rewards amongst themselves in any manner which the team determines appropriate.

- Team recognition
  Management recognizes the work of the team and recognizes that performance in such rewards as special parking spaces, lunch with the boss, time off with pay, and announcements of the team’s accomplishments.

3.3.3 Quality Management Organizational Structure

Until approximately 30 years ago almost all organizational structures were hierarchical. Direction came from the top down. The quality revolution significantly impacted the typical hierarchical structure. The structure was flattened, employees were empowered to make more decisions and new approaches to management were introduced.

The ability to become a leader is partially dependent upon whether the organization has a traditional hierarchical management approach, or the new quality management philosophy. The new quality management philosophy encourages leadership; the traditional hierarchical approach to management encourages managers.

Most managers practice traditional management. They have been taught to control their organization and employees, using an “I’ll tell you what to do, and you’ll do it” mentality. Many managers look at the short-term because their commitment to the organization is short range.

The key differences in philosophy between traditional management and quality management environments are illustrated in Table 3-5.
Managing the Test Project

Table 3-5  Traditional versus Quality Management Philosophy

<table>
<thead>
<tr>
<th>Traditional Management Philosophy</th>
<th>Quality Management Philosophy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controls each result</td>
<td>Use the process</td>
</tr>
<tr>
<td>Who made the error?</td>
<td>What allowed the error?</td>
</tr>
<tr>
<td>Correct the error</td>
<td>Reduce variation and prevent the error</td>
</tr>
<tr>
<td>Employees are the problem</td>
<td>Refine the process</td>
</tr>
<tr>
<td>Management accountable to their manager</td>
<td>Management accountable to the customer</td>
</tr>
<tr>
<td>Competition between organizations</td>
<td>Teamwork</td>
</tr>
<tr>
<td>Motivation from fear of failure</td>
<td>Motivation from within (self)</td>
</tr>
<tr>
<td>Management of outputs (results)—focusing on detection of defects</td>
<td>Management of process inputs—methods or sources of variation that focus on preventing defects</td>
</tr>
<tr>
<td>Fire fighting</td>
<td>Continuous process improvement</td>
</tr>
<tr>
<td>Accomplishment from meeting quotas, the monthly or quarterly bottom line</td>
<td>Accomplishment from long-term impact of improving processes</td>
</tr>
</tbody>
</table>

Table 3-5  Traditional versus Quality Management Philosophy

The culture change required to build a quality management environment is significant. Management must change its philosophy, practices, and assumptions about work and people. The biggest mistake usually made when implementing a quality management environment is underestimating the cultural changes that must occur and the time required for accomplishing these changes. It is usually felt that only a few control charts are needed, and little effort is made to change the culture of the organization.

The programs needed to change from a traditional to quality management culture must be customized for an organization and its current culture. Table 3-6 illustrates cultural changes that must be made.

<table>
<thead>
<tr>
<th>Category</th>
<th>Traditional Culture</th>
<th>Quality Management Culture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mission</td>
<td>Maximum return on investment (ROI), management by objectives (MBO)</td>
<td>Ethical behavior and customer satisfaction, climate for continuous improvement, ROI as a measure of performance</td>
</tr>
<tr>
<td>Customer Requirements</td>
<td>Incomplete or ambiguous understanding of customer requirements</td>
<td>Uses a systematic approach to seek out, understand and satisfy both internal and external customer requirements</td>
</tr>
<tr>
<td>Suppliers</td>
<td>Undirected relationship</td>
<td>Partnership</td>
</tr>
<tr>
<td>Objectives</td>
<td>Orientation to short-term objectives and actions with limited long-term perspective</td>
<td>Deliberate balance of long-term goals with successive short-term objectives</td>
</tr>
</tbody>
</table>
Table 3-6  Organizational Changes from Traditional Culture to a Quality Management Culture

<table>
<thead>
<tr>
<th>Category</th>
<th>Traditional Culture</th>
<th>Quality Management Culture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improvement</td>
<td>Acceptance of process variability and subsequent corrective action as the norm</td>
<td>Understanding and continually improving the process</td>
</tr>
<tr>
<td>Problem Solving</td>
<td>Unstructured individualistic problem-solving and decision-making</td>
<td>Predominantly participative and interdisciplinary problem-solving and decision-making based on substantive data</td>
</tr>
<tr>
<td>Jobs and People</td>
<td>Functional, narrow scope, management controlled</td>
<td>Management and employee involvement, work teams, integrated functions</td>
</tr>
<tr>
<td>Management Style</td>
<td>Management style with uncertain objectives that instills fear of failure</td>
<td>Open style with clear and consistent objectives, encouraging group-derived continuous improvement</td>
</tr>
<tr>
<td>Role of Manager</td>
<td>Plan, organize, assign, control and enforce</td>
<td>Communicate, consult, delegate, coach, mentor, remove barriers, and establish trust</td>
</tr>
<tr>
<td>Rewards &amp; Recognition</td>
<td>Pay by job, few team incentives</td>
<td>Individual and group recognition and rewards, negotiated criteria</td>
</tr>
<tr>
<td>Measurement</td>
<td>Orientation toward data gathering for problem identification</td>
<td>Data used to understand and continuously improve processes</td>
</tr>
</tbody>
</table>

3.3.4 Code of Ethics

Members of professional organizations have a responsibility to accept the standards of conduct that the organization represents. Those certified by a professional organization must maintain high standards of conduct in order to effectively discharge their responsibility.

3.3.4.1 Responsibility

Acceptance of any certification designation is a voluntary action. By acceptance, those certified assume an obligation of self-discipline beyond the requirements of laws and regulations.

The standards of conduct set forth in the code of ethics provide basic principles in the practice of IT quality assurance. Those certified should realize that their individual judgment is required in the application of these principles.
3.4 Managing Change

Once a test plan has been developed, it must be implemented. A test plan should be viewed as a contract between the testers and the stakeholders in the software system. If the test plan is executed as stated the test should be viewed as successful by the testers.

A test plan should also be viewed as a document that needs to be continually updated as circumstances change. When users make changes to the software system, the test plan needs to be changed. When circumstances surrounding the execution of the test plan change, the test plan must change. For example, if the developers are late in getting the software to the testers, there may be time constraint problems. Thus the scope of testing may need to be changed because of the late delivery of the software components to the testers.

There are three test management activities that should be incorporated to ensure currentness and effectiveness of the test plan:

- Software Configuration Management
- Software Change Management
- Software Version Control

Each of these will be discussed individually.

3.4.1 Software Configuration Management

Software Configuration Management (CM) is a key component of the infrastructure for any software development organization. The ability to maintain control over the changes made to all project artifacts is critical to the success of a project. The more complex an application is, the more important it is to control change to both the application and its supporting artifacts.

Most organizations understand the importance of managing source code and the changes made to it, but all project artifacts need to be managed; from requirements and models, to test cases and test data. For example, if a requirement is changed or added during the project after the requirements have been baselined, the tests designed and built to validate the requirement must also be updated. The appropriate version of the tests must be executed for the updated code, or incorrect results will be obtained.

For large projects, the Configuration Manager is often a full-time member of the project team. This person provides and supports the overall CM infrastructure and environment for the project team. Smaller efforts might not be able to support a full-time Configuration Manager, and might share a resource with other projects, or assign the role to someone on the project team in addition to their other responsibilities.
The CM function ensures that the CM environment facilitates product baselines, reviews, changes, and defect tracking. Responsibilities also include writing the CM plan and reporting change request-based progress statistics. The CM function supports product development activities so that developers and integrators have appropriate workspaces to build and test their work, and all artifacts are available as required.

The list below illustrates the types of project artifacts that must be managed and controlled in the CM environment:

- Source code
- Requirements
- Analysis models
- Design models
- Test cases and procedures
- Automated test scripts
- User documentation, including manuals and online Help
- Hardware and software configuration settings
- Other artifacts as needed

Multiple tools are available on the market to help teams manage project artifacts. Mainframe tools for source control like Change Man are typically used in conjunction with tools that support analysis and design artifacts created in a PC-based environment. A single tool, such as PVCS or Clear Case, can usually support client/server, Intranet, and Internet applications. If the application is implemented in an environment with multiple operating systems, e.g., UNIX and Windows NT, then the tool selected must support both environments.

### 3.4.2 Software Change Management

Managing software change is a process. The process is the primary responsibility of the software development staff. They must assure that the change requests are documented, that they are tracked through approval or rejection, and then incorporated into the development process. Many software development project teams group changes to be implemented into a new version of the system. This grouping normally reduces the costs associated with incorporating changes into the software requirements during development.

The testers need to know two aspects of change:

1. The characteristics of the change so that modification to the test plan and test data can be made to assure the right functionality and structure are tested.

2. The version in which that change will be implemented.

Without effective communication between the development team and the test team regarding changes, test effort may be wasted testing the wrong functionality and structure.
3.4.3 Software Version Control

Once dynamic testing begins, the project team must ensure that the appropriate versions of the software components are being tested. The time and effort devoted to testing are wasted if either the incorrect components, or the wrong version of the correct components, have been migrated to the test environment. The Configuration Manager must develop both migration and back-out procedures to support this process. These processes can be either manual or automated. Often, the CM tool selected will support the creation of automated build procedures that reduce the level of risk associated with migration.

3.4.3.1 Example

For discussion purposes, let’s assume that you are working on a large, multi-platform iterative development project. The application architecture consists of an Intranet Web Server, an Application Server, a Data Server, and a legacy application on the mainframe. The architecture alone has introduced a high level of risk into the project because of the multiple points of failure, and the multiple platforms and operating systems have added an additional amount of risk. To speed up development of the application, the project team has selected to employ an iterative development approach that will result in four iterative “builds” of the application. Each build is given a new release number, and integrates additional functionality with that delivered in the previous releases.

The development environment in place to support this project consists of the following:

- Four development servers, one for each of the development iterations.
- An Integration Test environment, where integration work is completed for each of the builds.
- A Test environment, where the test team conducts testing on each build.
- A Production environment, where users access and use the production version of the application.
- A Production Support environment, where the support team investigates reported problems and tests emergency fixes prior to moving them into production.

The first challenge for the CM function is to keep the environments configured appropriately. For example, the Production Support environment must be set up exactly as the Production environment, and must contain the same release version of the application components that are running in Production. Emergency fixes must be applied to the version of the component running in production – not necessarily the latest version of the component, which might be vastly different from the production version if it is being enhanced for one of the future releases. If there is a problem with the new release migrated to production, the back-out procedures must be utilized to revert the environment back to the previous release of the software.

The Test environment configurations must also mirror the Production environment. This is critical if the test is to be successful at predicting the applications performance in the Production environment. Once the environment is configured correctly, the release that is
ready for test must be migrated into the environment. Any defects that are identified can be corrected in the next “build” or release. If a release is either built incorrectly, or contains defects that prevent the continuation of testing, the back-out procedures can be utilized here as well.

Once a stable CM infrastructure is in place, the benefits of iterative development are easier to achieve. Without it, the development may be faster, but may also be more costly due to the management overhead associated with the build process.
It is often stated that a primary goal of a software tester is to reduce the risk associated with the deployment of a software application system. The very process of test planning is based on an understanding of the types and magnitudes of risk throughout the software application life cycle. The objective of this skill category is to explain the concept of risk which includes project, process, and product risk. The tester must understand these risks in order to evaluate whether the controls are in place and working in the development processes and within the application under test. Also, by determining the magnitude of a risk the appropriate level of resources can be economically allocated to reduce that risk.

<table>
<thead>
<tr>
<th>Skill Category</th>
<th>4</th>
</tr>
</thead>
</table>

Risk in the Software Development Life Cycle

<table>
<thead>
<tr>
<th>Risk Concepts and Vocabulary</th>
<th>4-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Risk Management</td>
<td>4-5</td>
</tr>
<tr>
<td>Risk Assessment</td>
<td>4-7</td>
</tr>
<tr>
<td>Risk Control</td>
<td>4-9</td>
</tr>
<tr>
<td>Risk Management Plan</td>
<td>4-11</td>
</tr>
<tr>
<td>Risks Associated with Software Development</td>
<td>4-12</td>
</tr>
<tr>
<td>Premature Deployment Risk</td>
<td>4-16</td>
</tr>
<tr>
<td>Risks Associated with Software Testing</td>
<td>4-17</td>
</tr>
<tr>
<td>Risk Scoring Methods</td>
<td>4-20</td>
</tr>
</tbody>
</table>

The first eight sections of this skill category discuss Risk within the broader SDLC and the ninth section describes risk specifically as it relates to software testing.
4.1 Risk Concepts and Vocabulary

As mentioned in the introduction, risks to the overall project can be categorized in three groups: software project risk, software process risk, and software product risk. When thinking about risk and its impact on applications under development and subsequently customer satisfaction, it is useful to consider each of the three risk categories.

4.1.1 Risk Categories

The three categories of software risk, project, process, and product, are depicted in Figure 4-1.

![Software Risk Categories](image)

Figure 4-1  Software Risk Categories

4.1.1.1 Software Project Risk

This first category, software project risk, includes operational, organizational, and contractual software development parameters. Project risk is primarily a management responsibility and includes resource constraints, external interfaces, supplier relationships, and contract restrictions. Other examples are unresponsive vendors and lack of organizational support. Perceived lack of control over project external dependencies makes project risk difficult to manage. Funding is the most significant project risk reported in risk assessments.

4.1.1.2 Software Process Risk

This second category, software process risk, includes both management and technical work procedures. In management procedures, you may find process risk in activities such as planning, resourcing, tracking, quality assurance, and configuration management. In technical procedures, you may find it in engineering activities such as requirements analysis, design,
code, and test. Planning is the management process risk most often reported in risk assessments. The technical process risk most often reported is the development process.

4.1.1.3 Software Product Risk

This third category, software product risk, contains intermediate and final work product characteristics. Product risk is primarily a technical responsibility. Product risks can be found in the requirements phase, analysis and design phase, code complexity, and test specifications. Because software requirements are often perceived as flexible, product risk is difficult to manage. Requirements are the most significant product risks reported in product risk assessments.

4.1.2 Risk Vocabulary

Understanding the definitions of the following terms will aid in comprehending the material in this skill category.

Risk is the potential loss to an organization, for example, the risk resulting from the misuse of computer resources. This may involve unauthorized disclosure, unauthorized modification, and/or loss of information resources, as well as the authorized but incorrect use of computer systems. Risk can be measured by performing risk analysis.

Risk Event is a future occurrence that may affect the project for better or worse. The positive aspect is that these events will help you identify opportunities for improvement while the negative aspect will be the realization of threats and losses.

Risk Exposure is the measure that determines the probability of likelihood of the event times the loss that could occur.

Risk Management is the process required to identify, quantify, respond to, and control project, process, and product risk.

Risk Appetite defines the amount of loss management is willing to accept for a given risk.

Active Risk is risk that is deliberately taken on. For example, the choice to develop a new product that may not be successful in the marketplace.

Passive Risk is that which is inherent in inaction. For example, the choice not to update an existing product to compete with others in the marketplace.

Risk Acceptance is the amount of risk exposure that is acceptable to the project and the company and can be either active or passive.

Risk Assessment is an examination of a project to identify areas of potential risk. The assessment can be broken down into analysis, identification, and prioritization.
Risk Analysis is an analysis of an organization’s information resources, its existing controls, and its organization and computer system or application system vulnerabilities. It combines the loss potential for each resource or combination of resources with an estimated rate of occurrence to establish a potential level of damage in dollars or other assets.

Risk Identification is a method used to find risks before they become problems. The risk identification process transforms issues and concerns about a project into tangible risks, which can be described and measured.

Threat is something capable of exploiting a vulnerability in the security of a computer system or application. Threats include both hazards (any source of potential damage or harm) and events that can trigger vulnerabilities.

Vulnerability is a design, implementation, or operations flaw that may be exploited by a threat. The flaw causes the computer system or application to operate in a fashion different from its published specifications and results in destruction or misuse of equipment or data.

Damaging Event is the materialization of a risk to an organization’s assets.

Inherent Risk is the risk to an organization in the absence of any actions management might take to alter either the risk’s likelihood or impact.

Residual Risk is the risk that remains after management responds to the identified risks.

Risk Mitigation is the action taken to reduce threats and/or vulnerabilities.

Control is anything that tends to cause the reduction of risk. Control can accomplish this by reducing harmful effects or by reducing the frequency of occurrence.

A risk is turned into a loss by threat. A threat is the trigger that causes the risk to become a loss. For example, if fire is a risk, then a can of gasoline in the house or young children playing with matches are threats that can cause the fire to occur. While it is difficult to deal with risks, one can deal very specifically with threats.

Threats are reduced or eliminated by controls. Thus, control can be identified as anything that tends to cause the reduction of risk. In our fire situation, if we removed the can of gasoline from the home or stopped the young children from playing with matches, it would have eliminated the threat and thus, reduced the probability that the risk of fire would be realized.

If our controls are inadequate to reduce the risk, we have vulnerability.

The process of evaluating risks, threats, controls, and vulnerabilities is frequently called risk analysis. This is a task that the tester performs when he/she approaches the test planning from a risk perspective.

Risks, which are always present in a application environment, are triggered by a variety of threats. Some of these threats are physical—such as fire, water damage, earthquake, and hurricane. Other threats are people-oriented—such as errors, omissions, intentional acts to disrupt system integrity, and fraud. These risks cannot be eliminated, but controls can reduce
the probability of the risk turning into a damaging event. A damaging event is the materialization of a risk to an organization’s assets.

### 4.2 Risk Management

Risk is usually one of the last areas addressed (if addressed at all) in the software development process. Many projects are undertaken without a solid foundation for making informed decisions by consciously assessing what can go wrong and the resultant impact when things do. Consequently, no forethought is given to contingencies when the inevitable does happen (e.g., scope of the project changes, time frames for development and/or testing are drastically slashed, new technology is engaged for the first time, etc.) If for no other reason than to avert crisis management, organizations need to adopt a risk management strategy.

While the essence of effective risk management is based on common sense and is employed to some degree by many managers within the SDLC, many formal techniques, methods, and tools can be used to enhance their abilities to deal with risks. Using these techniques and tools will help:

- Minimize management by crisis
- Minimize surprises and problems
- Decrease overall project variances
- Increase the probability of project success

#### 4.2.1 The Risk Management Process

It takes discipline in an organization for a good risk management program to be successful. Risk management is dynamic and ongoing throughout the software development process. It requires active participation from the entire team starting with management and down to the working level.

Risk management involves several important components, each of which is illustrated in Figure 4-2.
4.2.1.1 Risk Assessment

Risk Assessment can best be described as an examination of a project and then identifying areas of potential risk. The assessment can be broken down into the three following subcomponents:

- Risk Identification
- Risk Analysis
- Risk Prioritization

Risk Assessment will be described in detail in section 4.3.

4.2.1.2 Risk Control

Risk control is the process of managing risks to achieve the desired outcomes. Risk control can be broken down into the following subcomponents:

- Risk Planning
- Risk Resolution
- Risk Monitoring

Risk Control will be described in detail in section 4.4.

4.2.1.3 Hindrances to Risk Management

Those who practice risk management agree that it must be performed regularly throughout the life cycle of a software system. Risks are dynamic, meaning that they change over time. It is important to be aware of the following concerns.

Risk viewed as extra activity – If risk management is viewed as an extra activity layered on top of already assigned work, then risk management will be relegated to an “only if we have time” process and the value of risk management will be lost.
**Risk viewed as someone else’s activity** – If risk management is viewed as someone else’s problem with a not “my” responsibility attitude, then the basic tenet that managing risk is the responsibility of everyone on the project team has not been communicated correctly.

When risk management is implemented as a mainstream function in the development process, the value to the project is substantial. Risk is neither more nor less important than work; it is an inline process. There is not a “risk season” or a separate team to perform risk management. Risk management is not tied to a planning phase that is completed early in the project life cycle. It must be included throughout the project.

Risk management based on intuitions and individual initiative alone is seldom effective and rarely consistent. When risk management methods are used, they are often simplistic and users have little confidence in the results of their risk analysis results. The following factors may contribute to the low usage of risk management methods in practice:

- Risk is an abstract and fuzzy concept and users lack the necessary tools to define risk more accurately for deeper analysis.
- Many current risk management methods are based on quantification of risks for analysis and users are rarely able to provide accurate enough estimates for probability and loss for the analysis results to be reliable.
- Risks have different implications to different stakeholders.
- Each risk may affect a project in more than one way.

Many current risk management methods are perceived as complex or too costly to use. A risk management method should be easy to use and require a limited amount of time to produce results; otherwise, it will not be used.

### 4.3 Risk Assessment

There is no single widely accepted definition of risk assessment. It is not a discipline or a methodology but rather an overall approach to problem solving.
Risk assessment means:

- Identification of risks
- Analysis of identified risks
- Prioritization of identified risks

Assessing the risk to a business requires knowledge of the business functions. Users MUST be involved in the risk assessment process. All business tasks/functions must be identified, a prioritization scheme developed, and then identified risks ranked using structural, technical, and size risk assessments.

In other words:

- What can happen? What is the risk?
- What category of risk is it?
- How big is it?
- What exactly is being exposed to the risk?
- How likely is it to happen?
- What’s the worst that will occur if it does?
- Can it be considered an acceptable risk?
- What alternatives are there?
- Will the alternatives invoke additional risks?
- How can the losses be mitigated?

### 4.3.1 Risk Assessment - Risk Identification Process

Risk identification is the first element within the risk assessment component. Before risks can be managed, they must be identified. Risk identification aims to find the major risks before they adversely affect a project (project, process, or product). It is important to develop techniques to discover risks by exploiting and communicating the risk knowledge of the project team.

### 4.3.2 Risk Assessment - Risk Analysis Process

Risk analysis will likely involve a collection of numerical data and the use of mathematical and statistical analyses. Questions such as “how risky is it?” or “how likely is it to happen?” or “what will happen if we don’t do ABC?” are usually central concerns but can only be answered probabilistically, at best. Sometimes the questions are framed as cost/benefit analyses.
Often policy costs can be readily quantified but resulting benefits are frequently difficult to estimate in monetary terms. Sometimes, however, qualitative information or relative ranking of the risks associated with various alternatives provides a better basis for decision-making and is more comprehensible to decision-makers and the public. Risk analysis usually requires making judgments of this sort. Risk analysis carries a responsibility to reveal underlying assumptions of the analysis, data quality, and data gaps, and the degree of uncertainty in its findings and recommendations.

### 4.3.3 The Risk Assessment Method

The risk assessment is performed using an assessment team trained in basic risk concepts, the risk assessment method, and the risk management process. The risk assessment objectives include training in techniques for risk identification that are used throughout the project and providing a baseline of assessed risks for continues risk management.

### 4.4 Risk Control

Why are so many problems encountered while developing software systems? Unfortunately, the culture today has become unaccustomed to thinking ahead or anticipating the worst-case scenario.

![Figure 4-4 Risk Control](image)

**Figure 4-4 Risk Control**

Risk control means:

- Planning for managing risk
- Risk resolution strategies
- Monitoring and Tracking
A useful analogy in Risk Control that helps to illustrate the importance of prevention and early detection in producing high-quality software is fire prevention.

- **Scenario 1:** Do nothing. Fires occur occasionally, and there is nothing we can do about it. This will lead to a disaster; there will be losses in both lives and property. **There is a low probability but a high loss. No Risk Control**

- **Scenario 2:** Rely on fire fighters. Firefighters will be contacted, but what will happen during the time from when someone calls them to the time they arrive at the scene of the fire? As in scenario 1, there could be losses in both lives and property. **Probability stays the same but loss is reduced.**

- **Scenario 3:** Rely on fire extinguishers as a first defense. When a fire is small, it could be put out with a fire extinguisher and prevented from growing. The property will be saved from further losses. There is one problem with this; it has to be known when the fire starts. If everyone is asleep, no one will report the fire in time: this is now scenario 2. **Probability stays the same but the potential loss is reduced.**

- **Scenario 4:** Use smoke detectors for early fire detection. If people are alerted about the fire in its initial stages, it could be put out as in scenario 3. There is still one problem: someone has to hear the alarm. The best situation is not to have a fire in the first place. **Probability stays the same but the potential loss is reduced further.**

- **Scenario 5:** Use prevention. By preventing fire there is nothing to worry about. Prevention such as: learning that matches are not toys, heeding advice from fire experts, and so on. **Probability is reduced and loss is reduced. Risk controls are in place that reduce both the probability and the loss.**

### Develop Risk Resolution Alternatives

The most creative aspect of risk control is determining the possibilities to resolve risk. In order to develop risk resolution alternatives, there must be one or more goals and a list of assessed risks. Goals as well as risks should be prioritized. Knowing priorities helps in planning the allocation of resources. When goals or risks share the same priority, resources should be provided equally.

### 4.4.1 Risk Control - Planning

In most organizational levels, planning usually comes before the accomplishment of work. Although management is responsible for high-level project planning, everyone makes plans for accomplishing their assigned work. Planning for risk management includes the activity of developing some type of management policy and procedures.

A risk management plan is documented to respond to the goals established for the risk management initiative. Technical staff and integrated product teams may develop more detailed action plans to resolve technical risk. These plans are needed to delegate responsibility and authority for managing risk to the lowest possible levels within the organization.
4.4.2 Risk Control - Risk Resolution Strategies

Risk resolution strategies are usually cited as risk handling options subdivided as acceptance, avoidance, protection, reduction, transfer, and research to name a few. Generally, the assessment of management strategies is an approximation since the necessary decisions must occur early in a program when things are still unclear. However, if experienced personnel are given the facts, one can expect very good decisions since there is seldom any real mystery about the practicality of options available. In most cases, the practicality of any option is usually just an issue of schedule and funding.

4.4.3 Risk Control - Monitoring & Tracking

After the planning is accomplished and risk control is well underway, the risk manager should be responsible for presenting the status of all risks at all reviews. Risk reviews should be a part of both technical and programmatic reviews.

Monitoring risk may seem like a passive activity, but that is far from the truth. Risk tracking activities include measuring risk and observing project indicators for valuable information regarding when it is time to execute the risk action plan.

An indicator implies a value without specifying the quantity directly (e.g., the number of function points is a size measure that is an indicator of software complexity). Groups of indicators provide visibility into project status (e.g., planned versus actual cost). A leading indicator is one that has a predictive capability (e.g., requirements growth may be a leading indicator for software size).

Indicators help us to know when to take action to avoid the consequences of risk. The key is to know when to take action for effective risk control.

4.5 Risk Management Plan

Having described Risk Management which included Risk Assessment and Risk Control in previous sections, this section describes the process of actually creating the plan for Risk Management.

A risk management plan incorporates the documented goals, strategy, and methods for performing risk management on a project. The purpose of developing such a plan is to determine the approach for performing cost-effective risk management on the project.
The plan should be detailed enough to convey the required information to project participants. For a small project, the risk management plan might be only 5 pages long. For a large project, the plan might grow to 20 pages. The plan may be a separate document or part of a larger plan (e.g., project management plan, system test plan, or software development plan).

4.6 Risks Associated with Software Development

Each software system has a unique set of risks. Some of those risks are associated with the software functions, and other risks are associated with the process that develops the software. The risks associated with development should be assessed for each software system during development.

The risks associated with software development and its related use are listed below and then described individually.

- Improper use of technology
- Repetition of errors
- Cascading of errors
- Inability to translate user needs into technical requirements
- Incorrect entry of data
- Concentration of data
- Inability to react quickly
- Inability to substantiate processing
- Concentration of responsibilities
- Erroneous or falsified input data
- Misuse by authorized end users
- Uncontrolled system access
- Ineffective security and privacy practices for the application
- Procedural errors during operations
- Program errors
- Operating system flaws
- Communications system failure

4.6.1 Improper Use of Technology

Computer technology provides systems analysts and programmers with a variety of processing capabilities. This technology must be matched to the needs of the user to optimize the implementation of those needs. A mismatch of technology and needs can result in an unnecessary expenditure of organizational resources.
4.6.2 Repetition of Errors

In a manual-processing environment, errors are made individually. Thus, a person might process one item correctly, make an error on the next, process the next twenty correctly, and then make another error. In automated systems, the rules are applied consistently. Thus, if the rules are correct, processing is always correct, but if the rules are erroneous, processing will always be erroneous.

4.6.3 Cascading of Errors

The cascading of errors is the domino effect of errors throughout an application system. An error in one part of the program or application triggers a second (yet unrelated) error in another part of the application system. This second error may trigger a third error, and so on.

4.6.4 Inability to Translate User Needs into Business and Technical Requirements

One of the major failures of system development has been a communication failure between users and project personnel. In many organizations, users cannot adequately express their needs in terms that facilitate the application development process. Likewise, the technical staff is often unable to appreciate the concerns and requirements of its users.

While the risk has been mitigated in many organizations by the implementation of iterative development methodologies, like Agile, the risk continues to be significant industry wide.

4.6.5 Incorrect Entry of Data

In software applications, there is a mechanical step required to convert input data into machine-readable format. In the process of conducting this task, errors can occur. Data that was properly prepared and authorized may be entered into applications incorrectly. Some of the devices used by customers are unknown to the software developers. The data originator manually inputs information and during this process errors are made.

4.6.6 Concentration of Data

Computer applications concentrate data in a format that is easy to access. In manual systems, data is voluminous and stored in many places. It is difficult for an unauthorized individual to spend much time browsing undetected through file cabinets or other manual storage areas.

With digital media, the potential that an unauthorized individual might access the data is substantially increased. The data can be copied quickly without leaving any visible trail or
destroying the original data. Thus, the owners of the data may not be aware that the data has been compromised.

Digital storage technology increases the risk of data manipulation and compromise. The more data stored in a single place, the greater the value of that information to an unauthorized individual.

The concentration of data increases the problems of greater reliance on a single piece of data. If information entered is erroneous, the more applications that rely on that piece of data, the greater the impact of the error. In addition, the more applications that use the concentrated data, the greater the impact when the data is unavailable because of problems with hardware or software.

4.6.7 Inability to React Quickly

Much of the value of computer applications is the ability to satisfy user needs on a timely basis. Some of these needs are predetermined and reports are prepared on a regular basis to meet these needs. Other needs occur periodically requiring special actions to satisfy them. If the computer application is unable to satisfy these special needs on a timely basis, redundant systems may be built for that purpose.

The reliance on an application to access stored data can hold necessary information hostage until an application is developed to provide the necessary reporting.

4.6.8 Inability to Substantiate Processing

Computer applications should contain the capability to substantiate processing. Application systems need to substantiate processing for the purposes of correcting errors and proving the correctness of processing. When errors occur, computer personnel need to pinpoint the cause of those errors so they can be corrected. Application users and control-oriented personnel, such as auditors, frequently want to verify the correctness of processing.

4.6.9 Concentration of Responsibilities

Software applications tend to concentrate the responsibilities of many people into the hands of a few. Responsibilities that are conceptually segregated for control purposes may be concentrated into a single application system. In addition, a single application system may concentrate responsibilities from many departments within an organization.

Likewise, responsibility and/or access may be concentrated in IT personnel. For example, the database administrator may absorb data control responsibilities from many areas in the organization. A single project leader may have the processing responsibility for many areas in the organization. Methods of separation of duties must be substituted for the segregation of
duties among people. Risks associated with the concentration of responsibilities includes controls may be thwarted when too much responsibility is concentrated in one person (e.g., embezzling from company) and risk of too much knowledge loss if a person leaves the company.

### 4.6.10 Erroneous or Falsified Input Data

Erroneous or falsified input data is the simplest and most common cause of undesirable performance by an applications system. Vulnerabilities occur wherever data is collected, processed, or input into application systems.

### 4.6.11 Misuse by Authorized End Users

End users are the people served by the IT system. The system is designed for their use, but they can also misuse it for undesirable purposes. It is often very difficult to determine whether their use of the system is in accordance with the legitimate performance of their job.

### 4.6.12 Uncontrolled System Access

Organizations expose themselves to unnecessary risk if they fail to establish controls over who can use the IT systems and who can access the information contained in the system.

### 4.6.13 Ineffective Security and Privacy Practices for the Application

Inadequate manual checks and controls to ensure correct processing by the IT system or negligence by those responsible for carrying out these checks can result in many vulnerabilities.

### 4.6.14 Procedural Errors during Operations

Both errors and intentional acts committed by the operations staff may result in improper operational procedures and lapsed controls, as well as losses in storage media and output.
4.6.15 Program Errors

Application programs should be developed in an environment that requires and supports complete, correct, and consistent program design, good programming practices, adequate testing, review, documentation, and proper maintenance procedures. Although programs developed in such an environment will still contain undetected errors, programs not developed in this manner will probably contain many errors. Additionally, programmers can deliberately modify programs to produce undesirable side effects, or they can misuse the programs they are in charge of.

4.6.16 Operating System Flaws

Design and implementation errors, system generation and maintenance problems, and deliberate penetrations resulting in modifications to the operating system can produce undesirable effects in the application system. Flaws in the operating system are often difficult to prevent and detect.

4.6.17 Communications System Failure

Information being routed from one location to another over communication lines is vulnerable to accidental failures and to intentional interception and modification by unauthorized parties.

4.7 Premature Deployment Risk

Premature deployment is defined as releasing the software into production under the following conditions:

- The requirements were implemented incorrectly
- The test plan has not been completed
- Defects uncovered in testing have not been corrected

The customer/user of software should expect operational problems if the software is released without completing the project or testing. These risks include the risks defined as a software risk. The decision that the customer/user must make is, “Is the risk associated with premature deployment into production less than the risk of not releasing the software?”

When a decision is made by project management that an application will be released to production prematurely, the test plan must be updated to reflect the risk associated with this decision. If it is decided that certain tests will not be run because of time constraints then the modules affected should be indicated as “out of scope” for testing. There are many reasons
why an application may be deployed before all planned project tasks, including testing, have been completed. However, all risks must be documented so all stakeholders are aware of this change.

### 4.8 Risks Associated with Software Testing

Testing is inherently a risk-based activity. Most companies would not pay to add testing to the cost of the project if there was not a cost associated with the risk of failure. Exhaustive testing is impractical for most applications under development. Exceptions include applications that support very high risk processes, such as air traffic control, nuclear power generation station operations, defense systems, and so on. The project team must design a test strategy that utilizes a balance of testing techniques to cover a representative sample of the system in order to minimize risk while still delivering the application to production in a timely manner.

It is the test manager’s responsibility to determine how to apply the test methodology to achieve the greatest level of confidence in the application under development. Risk is a major driver in the test planning activity. As described earlier in this skill category, risk exposure is a result of two variables:

- The probability that an undesirable event will occur
- The consequences or impact associated when the undesirable event happens

The test manager must determine the appropriate amount of testing to perform based upon the risks associated with the application. These risks can arise from the newness and reliability of the technology being used, the nature of the application, or from the priority of the business functions under test. The amount of testing that should be performed is directly related to the amount of risk involved.

Understanding the risk-based nature of testing is also the key to dealing with the chronic problem of inadequate test resources. Risk must be used as the basis for allocating the test time that is available and for helping to make the selection of what to test and how to allocate resources.

#### 4.8.1 Software Testing Risks

The test manager is also responsible for identification of potential risks that might impact testing. The risks associated with software testing are listed below and then described individually.

- Not Enough Training/Lack of Test Competency
- Us versus Them Mentality
- Lack or Misuse of Test Tools
- Lack of Management Understanding and Support for Testing
• Lack of Customer and User Involvement
• Not Enough Schedule or Budget for Testing
• Over Reliance on Independent Testers
• Rapid Change
• Testers are in a Lose-Lose Situation
• Having to Say “No”
• Poor Ecosystem
• New Technology
• New Development Models

4.8.1.1 Not Enough Training/Lack of Test Competency

The majority of IT personnel have not been formally trained in testing, and only about half of full-time independent testing personnel have been trained in testing techniques. This causes a great deal of misunderstanding and misapplication of testing techniques.

4.8.1.2 Us versus Them Mentality

This common problem arises when developers and testers are on opposite sides of the testing issue. Political infighting takes up energy, sidetracks the project, and accomplishes little except to negatively impact relationships.

4.8.1.3 Lack or Misuse of Test Tools

IT management may have the attitude that test tools are a luxury. Manual testing can be an overwhelming task. When test tools are acquired, it is often through a process inadequate to fully prepare the organization for their proper use.

4.8.1.4 Lack of Management Understanding and Support of Testing

Support for testing must come from the top, otherwise staff will not take the job seriously and testers’ morale will suffer. Management support goes beyond financial provisions; management must also make the tough calls to deliver the software on time with defects or take a little longer and do the job right.

4.8.1.5 Lack of Customer and User Involvement

Users and customers may be shut out of the testing process, or perhaps they don’t want to be involved. Users and customers play one of the most critical roles in testing: making sure the software works from a business perspective.
4.8.1.6  **Not Enough Schedule or Budget for Testing**

This is a common complaint. The challenge is to prioritize the plan to test the right things in the given time.

4.8.1.7  **Over Reliance on Independent Testers**

Sometimes called the “throw it over the wall” syndrome, developers know that independent testers will check their work, so they focus on coding and let the testers do the testing. Unfortunately, this results in higher defect levels and longer testing times.

4.8.1.8  **Rapid Change**

In some development methodologies, such as Agile, the software is created and modified faster than the testers can test it. This highlights the need for automation but also for version and release management.

4.8.1.9  **Testers are in a Lose-Lose Situation**

On the one hand, if the testers report too many defects, they are blamed for delaying the project. Conversely, if the testers do not find the critical defects, they are blamed for poor quality.

4.8.1.10  **Having to Say “No”**

Having to be the bearer of bad news is the single toughest dilemma for testers. Nobody on the project likes to hear that the project still has significant issues.

4.8.1.11  **Testing Ecosystem**

The work ecosystem is not conducive to effective and efficient testing.

4.8.1.12  **New Technology**

Technology advances create new testing challenges. Mobile and cloud applications introduce even more risk to the test process. Security, performance, availability, complex component integration, and a team new to the technology are just a few of these new risk factors.

4.8.1.13  **New Development Models**

Along with new delivery platforms come new more iterative development methodologies like Agile. An integral part of this iterative approach is that testing activities are much more integrated across the development cycle.
4.8.2 Customize Processes based on Risk

Based on the risk assessment specific to the testing processes, changes to the test process may be needed. For example, if the testers lack training in the test process, that training should be provided prior to testing. If the training cannot occur, then extra supervision would be required. If inadequate test resources are provided and the scheduled test time is not sufficient, the number of tests may need to be reduced.

Risk analysis will determine the risks associated with the software being tested and performing the test process. Risk analysis should determine the magnitude of the risks and prioritize them in importance. Test planning will then consider those risks in developing the Test Plan.

4.9 Risk Scoring Methods

Sections 4.2 to 4.5 described in great detail the processes for Risk Management which included the risk assessment processes and the risk control processes. There was a stronger focus on the project and process risks than on potential product or application risks. Sections 4.6 to 4.8 described the types of risks commonly occurring in the SDLC and specifically within testing.

This section describes two scoring methodologies for analyzing risk for the purpose of prioritizing how best to utilize test resources. The methods discussed here would utilize information gleaned from the analysis processes described in section 4.3. The risk scores would then be utilized as part of the risk control processes as described in section 4.4.

4.9.1 Risk Score

Developing a risk score like most other measurements or metrics is more useful when a history of scores has been recorded on past projects. The ability to compare and draw inference from those comparisons is extremely important. This scoring approach identifies high-risk areas of the system which, when the scoring is completed prior to system testing, can improve testing in two ways:

- High-risk applications can be identified and resources allocated appropriately.
- Testing can focus on critical components and/or quality dimensions that are most important in a particular application system.
4.9.2 Two Risk Levels

Risk is defined as the probability that an unfavorable event will occur that results in a consequence (loss). To clearly define this “probability and loss” a two level risk assessment model as shown below can be used.

- Level I – Criticality/mission impact (consequences or impact of failure, “loss”)
- Level II – Size, system environment, reliability, and technology integration (likelihood of failure, “probability”)

4.9.2.1 Five Dimensions of Risk

Level I and Level II risks can be broken down into five dimensions of Risk. Listed below are these five dimensions and a few characteristics within each dimension.

- Criticality/mission impact
  - Criticality of system to the organization
  - Degree of dependence on the system
- System size, scale, and complexity
  - Size of user area affected
  - Number and complexity of interfaces with other projects or system
- System environment and stability
  - Management involvement and commitment
  - Newness of process to users
  - Problems associated with current system development or performance
- Reliability and integrity
  - Links to information system (i.e., data hardware, communications, facilities)
  - Availability and adequacy of audit trails
- Technology Integration
  - Makeup of project teams in relation to technology used
  - User knowledge of system technology

4.9.3 Application Level Scoring Method

The simple scoring method is used to assess the risk of an application system relative to other application systems that have been scored on the same methodology. For that reason, it is important to build a risk score repository. Scores can be calculated using the format shown in.
4.9.3.1 Application Scoring Procedure

The five steps in this procedure are listed below with data posted to the Form.

4.9.3.1.1 **Assign an importance weight** to each of the five dimensions in the table. For example, when looking at the five dimensions it might be decided that the criticality/mission impact is of the highest importance, so a weight of 38 is assigned. The other four would be assigned weights also. The weighting scale can be set by the organization as long as it is done consistently each time the scoring method is used. The total of the weights can add up to 100.
although that isn’t necessary. However, having a standard total like 100 is a good way to increase consistency each time the method is used.

4.9.3.1.2 Assign risk level for each of the five dimensions. This is a continuum from 1 to 5, where 5 is for high risk and 1 is for low risk.

4.9.3.1.3 Calculate dimension risk score. This is the weight times the risk level.

4.9.3.1.4 Calculate application risk score. The Level I risk score is the score of the criticality dimension (consequence). The Level II risk score is the sum of dimensions 2–5 (likelihood).

4.9.3.1.5 Rank application scores. The highest-scoring systems should receive the most validation.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Weight</th>
<th>Risk Level</th>
<th>Weighted Score</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Criticality/Mission Impact</td>
<td>38</td>
<td>5</td>
<td>190</td>
<td>Customer information tracking must be correct.</td>
</tr>
<tr>
<td>(Consequences)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Size, Scale, and Complexity</td>
<td>10</td>
<td>3</td>
<td>30</td>
<td>System is small compared to other systems at the company.</td>
</tr>
<tr>
<td>3. Environment and Stability</td>
<td>14</td>
<td>4</td>
<td>56</td>
<td>The system will be purchased and will be new to users.</td>
</tr>
<tr>
<td>4. Reliability and Integrity</td>
<td>21</td>
<td>5</td>
<td>105</td>
<td>The system must be available when needed.</td>
</tr>
<tr>
<td>5. Technology Integration</td>
<td>17</td>
<td>5</td>
<td>85</td>
<td>The system uses a new database technology.</td>
</tr>
<tr>
<td>Total of dimensions 2–5 are the</td>
<td></td>
<td></td>
<td>576</td>
<td></td>
</tr>
<tr>
<td>likelihood (probability)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.9.4 Component Level Scoring Method

The component scoring method is used to assess the risk of a particular component of an application system relative to other components in that application system. Application scores will be calculated using the Criticality/Mission Impact (Consequences) and Complexity, Size and Frequency of Use (likelihood). Scores can be calculated using the format shown in Table 4-3.

The steps are:

- Calculate the Probability score
- Plot the scores on the Risk Analysis chart
- Use the results to focus the test effort
  - first, focus on the components in quadrant IV
  - next, focus on the components in quadrant III
  - then, test the components in quadrant II
  - lastly, test the components in quadrant I

<table>
<thead>
<tr>
<th>Risk Factor/ Module</th>
<th>Weight of Risk</th>
<th>Complexity</th>
<th>Size</th>
<th>Frequency of Use</th>
<th>Probability Score</th>
<th>Impact Score</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>(WC+WS+WF)/3</td>
<td>1 to 10</td>
</tr>
</tbody>
</table>

Table 4-3 Component Level Scoring Method

4.9.4.1 System Scoring Procedure

The five steps in this procedure are listed below with data posted to .

4.9.4.1.1 Assign an impact value from the list shown in . Record it in the right most column of under “Impact Score.”

<table>
<thead>
<tr>
<th>Rating</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No impact</td>
</tr>
</tbody>
</table>
4.9.4.1.2 Calculate the Probability Score based on a weighted scoring of the three factors: complexity, size, and frequency of use. In the example, there is a rating for the module to select items for a shopping cart, process a payment by credit card, and handle shipping queries. The method to weight risk factors are:

Weighted Score for three factors:

Complexity (C) - Use a weight factor of 3
Size (S) - Use a weight factor of 2
Frequency of Use (F) - Use a weight factor of 1

Where High (5); Medium (3); Low (1)

Weighted Complexity (WC) = Weight of Risk x C
Weighted Size (WS) = Weight of Risk x S
Weighted Frequency of Use (WF) = Weight of Risk x F

Use the following formula for score:

\[
\frac{(C \times 3) + (S \times 2) + (F \times 1)}{3} = \text{Probability Score}
\]

Complete as shown in Table 4-4.
Table 4-4 Example of Component Level Scoring Method

4.9.4.1.3 Using the Probability Score and Impact Score from, plot the scores on the grid as shown in Figure 4-5.

<table>
<thead>
<tr>
<th>Risk Factor/Module</th>
<th>Complexity</th>
<th>Size</th>
<th>Frequency of Use</th>
<th>Probability Score (WC+WS+WF)/3</th>
<th>Impact Score 1 to 10</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Shopping Cart</strong></td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>(3+6+3)/3 = 4</td>
<td>7</td>
</tr>
<tr>
<td>Module: Selected Items</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Payment Process</strong></td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>(15+6+3)/3 = 8</td>
<td>9</td>
</tr>
<tr>
<td>Module: Credit Card Payment</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Order Queries</strong></td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>(9+6+1)/3 = 5.1</td>
<td>4</td>
</tr>
<tr>
<td>Module: Shipping Query</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4-5 Test Priority Grid
4.9.4.1.4 Based on the quadrants of the risk grid, resources should be applied to the components that fall into the quadrants in this order: quadrant IV, III, II, and I last. See Figure 4-6.

Figure 4-6 Test Priority Grid
Test Planning

Testers need specific skills to plan tests and to select appropriate techniques and methods to validate a software application against its approved requirements and design. In Skill Category 3, “Managing the Test Project,” test planning was shown to be part of the test administration processes. In Skill Category 4, “Risk in the SDLC,” the assessment and control of risk was discussed in detail. Specific to testing, risk assessment and control is an integral part of the overall planning process. Having assessed the risks associated with the software application under test, a plan to minimize those risks can be created. Testers must understand the development methods and operating environment to effectively plan for testing.

<table>
<thead>
<tr>
<th>The Test Plan</th>
<th>5-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisites to Test Planning</td>
<td>5-2</td>
</tr>
<tr>
<td>Hierarchy of Test Plans</td>
<td>5-5</td>
</tr>
<tr>
<td>Create the Test Plan</td>
<td>5-7</td>
</tr>
<tr>
<td>Executing the Plan</td>
<td>5-19</td>
</tr>
</tbody>
</table>

5.1 The Test Plan

Testing – like any other project – should be driven by a plan. The test plan acts as the anchor for the execution, tracking, and reporting of the entire testing project. A test plan covers what will be tested, how the testing is going to be performed, what resources are needed for testing, the timelines (schedule) by which the testing activities will be performed, and risks that may be faced in the testing process as well as risks to the application under test. The test plan will identify the relevant measures and metrics that will be collected and identify the tools that will be used in the testing process. The test plan is NOT a repository of every testing standard nor is it a list of the test cases.
5.1.1 Advantages to Utilizing a Test Plan

- Helps improve test coverage
- Helps avoid repetition
- Helps improve test efficiencies
- Reduces the number of tests without increasing the odds that defects will be missed
- Helps prevent oversights
- Improves communication
- Enables feedback on the plan (QC the test plan)
- Enables better communication regarding what will be done during the test process
- Provides education about relevant test details
- Improves accountability

The test plan serves two specific purposes: a contract and a roadmap.

5.1.2 The Test Plan as a Contract and a Roadmap

The test plan acts as a contract between the test organization and the project stakeholders spelling out in detail what the testing organization will do during the various testing phases and tasks. If not specifically stated as such, any activities not indicated in the test plan would be by definition, “out of scope.”

Second, the test plan acts as a roadmap for the test team. The plan describes the approach the team will take, what will be tested, how testing will be performed, by whom, and when to stop testing. The roadmap reflects the tactical activities to be accomplished and how they will be accomplished. By clearly identifying the activities, the plan creates accountability on the part of the test team to execute the plan and to ensure that objectives are met.

5.2 Prerequisites to Test Planning

If test planning is viewed as a process or a workbench, there are entrance criteria to the test planning process. The following entrance criteria are prerequisites to test planning:

- Objectives of Testing
- Acceptance Criteria
- Assumptions
- Constraints
- Understanding the Characteristics of the Software being Developed

Terminology

Objectives of Testing
Acceptance Criteria
Assumptions
Constraints
5.2.1 Objectives of Testing

The objectives of testing define what the purpose of the test project is. This might seem obvious, but it is crucial that the “why are we doing this?” is clear to all the stakeholders. Some objectives of the test project would likely be:

- Test to assure that the software development project objectives are met
- Test to assure that the functional and structural objectives of the software under test meet requirements
- Test to assure that the needs of the users are met, also stated as “fit for use”
- Achieve the mission of the software testing group

5.2.2 Acceptance Criteria

A key prerequisite for test planning is a clear understanding of what must be accomplished for the test project to be deemed successful. Nebulous acceptance criteria spell failure before the project ever begins.

Acceptance criteria will likely evolve from high level conceptual acceptance criteria to more detailed product level criteria. The details might manifest themselves in the form of Use Cases or, in an Agile project clear, concise User Stories. Regardless, it is critical that before the plan begins we understand where we are going.

5.2.3 Assumptions

In developing any type of plan, certain assumptions exist. For example, if a mobile application under test required a newly developed smartphone platform, an assumption could be that the hardware would be available on a specific date. The test plan would then be constructed based on that assumption. It is important that assumptions be documented for two reasons. First to assure that they are effectively incorporated into the test plan, and second, so that they can be monitored should the event included in the assumption not occur. For example, hardware that was supposed to be available on a certain date will not be available until three months later. This could significantly change the sequence and type of testing that occurs.

5.2.4 Team Issues

Issues relating to the individuals on the team tend to be both political and personal. The team issues could include who should run the project, who can make decisions, and which organizational group has authority to decide requirements. Unrealistic optimists and pessimistic naysayers alike create issues for the project and those dynamics must be understood. It is important to mitigate the people issues prior to starting the project.
Some organizations divide individuals into four categories when attempting to identify issues. These categories are:

- Those who will make the software system happen
- Those who will hope the software system happens
- Those who will let the software system happen
- Those who will attempt to make the software system not happen

If the stakeholders are divided among these four categories, issues are frequently apparent. For example, if two different business units want to make the software system happen, a decision would have to be made as to which would have primary responsibility and which would have secondary responsibility. If both want to have primary responsibility, conflict will occur.

### 5.2.5 Constraints

As described earlier, the test plan is both a contract and a roadmap. For both the contract to be met and the roadmap to be useful, it is important that the test plan be realistic. Constraints are those items that will likely force a dose of “reality” on the plan. The obvious constraints are test staff size, test schedule, and budget. Other constraints can include the inability to access user databases for test purposes, limited access to hardware facilities for test purposes, and minimal user involvement in development of the test plan and testing activities.

Because constraints restrict the ability of testers to test effectively and efficiently, the constraints must be documented and integrated into the test plan. It is also important that the end users of the software understand the constraints placed on testers and how those constraints may impact the role and responsibility of software testers in testing the application system.

### 5.2.6 Understanding the Characteristics of the Application

The test team should investigate the project characteristics in order to effectively plan how best to test the application. Some aspects of this would be done as part of the risk assessment and control processes. During this investigation the testers should at least do the following:

1. Define what it means to meet the project objectives. These are the objectives to be accomplished by the project team.

2. Understand the core business areas and processes. All information systems are not created equal. Systems that support mission-critical business processes are clearly more important than systems for mission-support functions (usually administrative), although these, too, are necessary. Focusing on core business areas and processes is essential to the task of assessing the impact of the problem and for establishing the priorities for the program.
3. Assess the severity of potential failures. This must be done for each core business area and its associated processes.

4. Identify the components for the system
   - Links to core business areas or processes
   - Platform languages and database management systems
   - Operating system software and utilities
   - Telecommunications
   - Internal and external interfaces
   - Owners
   - Availability and adequacy of technical documentation

5. Assure requirements are testable. Effective testing cannot occur if requirements cannot be tested to determine if they are implemented correctly.

6. Address implementation schedule issues.
   - Implementation checkpoints
   - Schedule of implementation meetings

7. Identify interface and data exchange issues including the development of a model showing the internal and external dependency links amongst core business areas, processes, and information systems.

8. Evaluate contingency plans for the application. These should be realistic contingency plans, including the development and activation of manual or contract procedures, to ensure the continuity of core business processes.

### 5.3 Hierarchy of Test Plans

Skill Category 1, section 1.8, “Testing Throughout the Software Development Life Cycle”, described the various levels of testing that should be completed for any software testing project. The V-diagram, as depicted in Figure 5-1, is used to visually describe these levels.
Figure 5-1 The V-diagram

One the left side of the “V” are primarily the verification or static tests and on the right side of the “V” are the validation or dynamic tests. These levels of testing include:

- Verification or static tests
  - Requirements reviews
  - Design reviews
  - Code walkthroughs
  - Code inspections
- Validation or dynamic tests
  - Unit testing
  - Integration testing
  - System testing
  - User acceptance testing

These testing levels are part of the Software Quality Assurance V&V processes. Many organizations have discrete plans as depicted in Figure 5-2, while others may incorporate all testing activities, both static and dynamic into one test plan. The complexity of this hierarchy may be driven by the development model, the size and complexity of the application under test, or the results of the risk assessment and control processes described in Skill Category 4.
5.4 Create the Test Plan

Some insight into the importance of test planning:

“The act of designing tests is one of the most effective error prevention mechanisms known…

The thought process that must take place to create useful tests can discover and eliminate problems at every stage of development.”

Boris Beizer

There is no one right way to plan tests. The test planning process and the subsequent test plan must reflect the type of project, the type of development model, and other related influencers. However, there are recognized international standards for the format of test plans which can serve as a good starting point in the development of a test plan standard for an organization. As noted in Skill Category 1, section 1.4, ISO/IEC/IEEE 29119-3 (replaces IEEE 829) defines templates for test documentation covering the entire software testing life cycle. Within the section on Test Management Process Documentation is the Test Plan standard. The material found in this section will reflect some
of the IEEE standards along with other good practices. This section will also include “how-to” information in order to help understand the components of the software test plan.

The test plan describes how testing will be accomplished. Its creation is essential to effective testing. If the plan is developed carefully, test execution, analysis, and reporting will flow smoothly. The time spent in developing the plan is well worth the effort.

The test plan should be an evolving document. As the development effort changes in scope, the test plan must change accordingly. It is important to keep the test plan current and to follow it. It is the execution of the test plan that management must rely on to ensure that testing is effective. Also, from this plan the testers ascertain the status of the test effort and base opinions on the results of the test effort.

Test planning should begin as early in the development process as possible. For example, in a waterfall development project planning would begin at the same time requirements definition starts. It would be detailed in parallel with application requirements and during the analysis stage of the project the test plan defines and communicates test requirements and the amount of testing needed so that accurate test estimates can be made and incorporated into the project plan. Regardless of the development methodology, planning must take place early in the life cycle and be maintained throughout.

The test plan should define the process necessary to ensure that the tests are repeatable, controllable, and ensure adequate test coverage when executed.

Recoverable - Once the necessary tests are documented, any test team member should be able to execute the tests. If the test must be executed multiple times, the plan ensures that all of the critical elements are tested correctly. Parts or the entire plan can be executed for any necessary regression testing.

Controllable - Knowing what test data is required, when testing should be run, and what the expected results are all documented to control the testing process.

Coverage - Based on the risks and priorities associated with the elements of the application system, the test plan is designed to ensure that adequate test coverage is built into the test. The plan can be reviewed by the appropriate parties to ensure that all are in agreement regarding the direction of the test effort.

5.4.1 Build the Test Plan

Prior to developing the test plan, the test team has to be organized. This initial test team is responsible for developing the test plan and then defining the administrative resources needed to complete the plan. Thus, part of the plan will be executed as the plan is being developed; that part is the creation of the test plan, which itself consumes resources.

The development of an effective test plan involves the following tasks that are described below.

• Set test objectives
• Develop the text matrix
• State Test Plan General Information
  o Define test administration

5.4.1.1 Set Test Objectives

Test objectives need to be defined and agreed upon by the test team. These objectives must be measurable and the means for measuring defined. In addition, the objectives must be prioritized.

Test objectives should restate the project objectives from the project plan. In fact, the test plan objectives should determine whether those project plan objectives have been achieved. If the project plan does not have clearly stated objectives, then the testers must develop their own by:

• Setting objectives to minimize the project risks
• Brainstorming to identify project objectives
• Relating objectives to the testing policy, if established

The testers must have the objectives confirmed as the project objectives by the project team.

When defining test objectives, ten or fewer test objectives are a general guideline; too many distract the tester’s focus. To define test objectives testers need to:

• Write the test objectives in a measurable statement, to focus testers on accomplishing the objective.
• Assign a priority to the objectives, such as:
  o High – The most important objectives to be accomplished during testing.
  o Average – Objectives to be accomplished only after the high-priority test objectives have been accomplished.
  o Low – The least important test objectives.

Note: Establish priorities so that approximately one-third are high, one-third are average, and one-third are low.

• Define the acceptance criteria for each objective. This should state, quantitatively, how the testers would determine whether the objective has been accomplished. The more specific the criteria, the easier it will be for the testers to determine whether it has been accomplished.

At the conclusion of testing, the results of testing can be consolidated upward to determine whether or not the test objective has been accomplished.

5.4.1.2 Develop the Test Matrix

Two of the essential items in the test plan are the functions to be tested (scope) and how testing will be performed. Both will be clearly articulated in the formal plan. Creating the test matrix is the key process in establishing these items. The test matrix lists which software
functions must be tested and the types of tests that will test those functions. The matrix shows “how” the software will be tested using checkmarks to indicate which tests are applicable to which functions. The test matrix is also a test “proof.” It proves that each testable function has at least one test, and that each test is designed to test a specific function.

An example of a test matrix is illustrated in Table 5-1. It shows four functions in a payroll system, with four tests to validate them. Since payroll is a batch system where data is entered all at one time, test data is also batched using various dates. The parallel test is run when posting to the general ledger, and all changes are verified through a code inspection.

<table>
<thead>
<tr>
<th>Software Function</th>
<th>Desk Check</th>
<th>Parallel Test</th>
<th>Code Inspection</th>
<th>Validate Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Payroll Deduction Calculation</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gross Pay</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tax Deduction</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Posting to the General Ledger</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
</tbody>
</table>

Table 5-1 Payroll System Test Matrix Example

5.4.1.3 Test Plan General Information

The general information is designed to provide background and reference data on testing. In many organizations this background information will be necessary to acquaint testers with the project. Incorporated into the general information are the administrative components of the test plan which identify the schedule, milestones, and resources needed to execute the test plan. The test administration is sometimes referred to as the “business plan” part of the test plan in that it does not describe what or how to test but rather details the infrastructure of the test project. Included in the general information and test administration are:

- Definitions (vocabulary of terms used in the test plan document)
- References
  - Document Map
    - Project Plan
    - Requirements specifications
    - High Level design document
    - Detail design document
  - Additional Reference Documents
    - Development and Test process standards
    - Methodology guidelines and examples
Test Planning

- Corporate standards and guidelines
  Any documents, policies, procedures, or regulations applicable to the software being tested or the test procedures.

- Key Contributors
- Human Resources Required
  - Test Resource Persons
  - Test Team
- Test Environment
  - Hardware Components
  - Software Components
- Resource Budgeting
  - Training for test analysts
  - Hardware
  - Software
  - Test Tools
  - Other

- Schedule
  Test milestones are designed to indicate the start and completion date of each test.

- Pretest Background
  Summary of any previous test experiences that might prove helpful with testing.

5.4.2 Write the Test Plan

The test plan may be as formal or informal a document as the organization’s culture dictates.

Testing is complete when the test team has fully executed the test plan.

5.4.2.1 Guidelines to Writing the Test Plan

Test planning can be one of the most challenging aspects of testing. The following guidelines can help make the job a little easier.

- Start early
  Even though all of the details may not yet be available, a great deal of the planning effort can be completed by starting on the general and working toward the specific. Starting early affords the opportunity to identify resource needs and plan for them before other areas of the project subsume them.

- Keep the Test Plan flexible
Test projects are very dynamic. The test plan itself should be changeable but subject to change control. Change tolerance is a key success factor in the development models commonly used.

- Review the Test Plan frequently
  Other people’s observations and input greatly facilitate achieving a comprehensive test plan. The test plan should be subject to quality control just like any other project deliverable.

- Keep the Test Plan concise and readable
  The test plan does not need to be large and complicated. In fact, the more concise and readable it is, the more useful it will be. Remember, the test plan is intended to be a communication document. The details should be kept in a separate reference document.

- Spend the time to do a complete Test Plan
  The better the test plan, the easier it will be to execute the tests.

---

5.4.2.2 Test Plan Standard

There is no one universally accepted standard for test planning. However, there is great consistency between the different organizations that have defined a test plan standard. This section will begin with a discussion of what is normally contained in a test plan and then provide an example of a test plan standard that is consistent with the test plan standards provided by major standard-setting bodies such as the International Standards Organization (ISO), Institute of Electrical and Electronics Engineers (IEEE), International Electrotechnical Commission (IEC), and National Institute of Standards in Technology (NIST).

Test Plans and their formats vary from company to company, but the best examples contain most of the elements discussed here. Several test plan outlines will be provided to demonstrate the various components of different plans.

5.4.2.2.1 Test Plan Example 1

- Test Scope
- Test Objectives
- Assumptions
- Risk Analysis
- Test Design
- Roles & Responsibilities
- Test Schedule & Resources
- Test Data Management
• Test Environment
• Communication Approach
• Test Tools

5.4.2.2.1.1 Test Scope

This section answers two equally important questions: “What will be covered in the test?” and “What will not be covered in the test?” The answers to either of these questions might include:

• Specific functional or structural requirements
• System interfaces
• Infrastructure components (e.g., network stability)
• Supplemental deliverables, such as application documentation

5.4.2.2.1.2 Test Objectives

A test objective is simply a testing “goal.” It is a statement of what the tester is expected to accomplish or validate during a specific testing activity. Test objectives:

• Guide the development of test cases, procedures, and test data.
• Enable the tester and project managers to gauge testing progress and success.
• Enhance communication both within and outside of the project team by helping to define the scope of the testing effort.

Each objective should include a high-level description of the expected test results in measurable terms and should be prioritized. In cases where test time is cut short, test cases supporting the highest priority objectives would be executed first.

5.4.2.2.1.3 Assumptions

These assumptions document test prerequisites, which if not met, could have a negative impact on the test. The test plan should document the risk that is introduced if these expectations are not met. Examples of assumptions include:

• Skill level of test resources
• Test budget
• State of the application at the start of testing
• Tools available
• Availability of test equipment

5.4.2.2.1.4 Risk Analysis

Although the test manager should work with the project team to identify risks to the project, this section of the plan documents test risks and their possible impact on the test effort. Some teams may incorporate these risks into project risk documentation if available. Risks that could impact testing include:

• Availability of downstream application test resources to perform system integration or regression testing

Version 14.2 5-13
• Implementation of new test automation tools
• Sequence and increments of code delivery
• New technology

5.4.2.2.1.5 Test Design

The test design details the following:
• The types of tests that must be conducted
• The stages of testing that are required (e.g., Unit, Integration, System, Performance, and Usability)
• Outlines the sequence and timing of tests

5.4.2.2.1.6 Roles & Responsibilities

This section of the test plan defines who is responsible for each stage or type of testing. A responsibility matrix is an effective means of documenting these assignments.

5.4.2.2.1.7 Test Schedule & Planned Resources

The test schedule section includes the following:
• Major test activities
• Sequence of tests
• Dependence on other project activities
• Initial estimates for each activity

The test plan should be viewed as a sub-plan of the overall project plan. It should not be maintained separately. Likewise, the test schedule and planned resources should also be incorporated into the overall Project Plan. Test resource planning includes:
• People, tools, and facilities
• An analysis of skill sets so that training requirements can be identified

5.4.2.2.1.8 Test Data Management

This section of the plan defines the data required for testing, as well as the infrastructure requirements to manage test data. It includes:
• Methods for preparing test data
• Backup and rollback procedures
• High-level data requirements, data sources, and methods for preparation (production extract or test data generation)
• Whether data conditioning or conversion will be required
• Data security issues

5.4.2.2.1.9 Test Environment

Environment requirements for each stage and type of testing should be outlined in this section of the plan, for example:
• Unit testing may be conducted in the development environment, while separate environments may be needed for integration and system testing
• Procedures for configuration management, release, and version control should be outlined
• Requirements for hardware and software configurations
• The defect tracking mechanisms to be used

5.4.2.2.10 Communication Approach

In the complex, environment required for software testing in most organizations, various communication mechanisms are required. These mechanisms should include:
• Formal and informal meetings
• Working sessions
• Processes, such as defect tracking
• Tools, such as issue and defect tracking, electronic bulletin boards, notes databases, and Intranet sites
• Techniques, such as escalation procedures or the use of white boards for posting the current state of testing (e.g., test environment down)
• Miscellaneous items such as project contact lists and frequency of defect reporting

5.4.2.2.11 Tools

Any tools that will be needed to support the testing process should be included here. Tools are usually used for:
• Workplan development
• Test planning and management
• Configuration management
• Test script development
• Test data conditioning
• Test execution
• Automated test tools
• Stress/load testing
• Results verification
• Defect tracking

The information outlined here cannot usually all be completed at once but is captured in greater levels of detail as the project progresses through the life cycle.

5.4.2.2 Test Plan Example 2 (based on IEEE 829)

The test plan components shown below follow the IEEE 829 standard closely. While the order may be different, the areas covered by the plans are quite similar. Only the outline list is provided here, for specific descriptions, visit the ISO 29119 site at: www.softwaretestingstandard.org.
5.4.2.2.3 Test Plan Example 3

This example plan uses the case of an application system used in a hospital. The detailed test plan is included as Appendix B.
1.3. KEY CONTRIBUTORS
1.4. HUMAN RESOURCES REQUIRED
   1.4.1. Test Resource Persons
   1.4.2. Test Team
1.5. TEST ENVIRONMENT RESOURCES
   1.5.1. Test Environment
   1.5.2. Hardware Components
   1.5.3. Software Components
1.6. RESOURCE BUDGETING
1.7. SCHEDULE

2. TEST OBJECTIVES AND SCOPE
   2.1. PURPOSE OF THIS DOCUMENT
      2.1.1. Summary of document contents
      2.1.2. Document Outputs
   2.2. OBJECTIVES OF SYSTEM TEST
      2.2.1. Method of achieving the Objectives
   2.3. SCOPE OF SYSTEM TEST
      2.3.1. Inclusions
      2.3.2. Exclusions
      2.3.3. Specific Exclusions
   2.4. DETAILED SYSTEM TEST SCOPE
      2.4.1. Access Management Upgrade
      2.4.2. Ambulatory
      2.4.3. Clinical Document Storage
      2.4.4. Emergency Room Triage Mobile Application
      2.4.5. Infrastructure
      2.4.6. Portal/Mobility Product Inventory Tracker

3. TEST STRATEGY
   3.1. OVERALL STRATEGY
   3.2. PROPOSED SOFTWARE “DROPS” SCHEDULE
   3.3. TESTING PROCESS
   3.4. INSPECTING RESULTS
   3.5. SYSTEM TESTING
      3.5.1. Test Environment Pre-Test Criteria
      3.5.2. Suspension Criteria
      3.5.3. Resumption Criteria
As mentioned previously in this skill category, testing is not complete until the test plan has been fully executed. The question then becomes what happens when factors intervene that make complete execution of the plan impossible. For example, application modules are delayed in delivery from the development team, yet the “go live” production date has not changed. In cases like that or for all cases when substantive changes are made impacting the ability of the test team to complete the test plan as published, the following steps should be taken as a minimum.
1. Make the necessary changes to the test plan to allow the test team to fully execute the modified plan and be in compliance with the project changes. Examples might be:
   a. Reprioritize the test objectives moving some objective out of scope
   b. Reprioritizing test cases moving some tests out of scope
   c. Move some modules out of scope in the plan
   d. Review resources and reallocate as necessary
2. Document, in the modified plan, the changes to the risk assessment and noting the necessary adjustments to risk control mechanisms.
3. All relevant stakeholders must sign off on the modified plan.

5.4.4 Attachments to the Test Plan

In section 5.1 it was stated that “The test plan is NOT a repository of every testing standard nor is it a list of all the test cases.” While certain external testing documents are referenced within the body of the test plan, they are not part of the plan. These other test documents exist as attachments to the Test Plan. Listed here are several such documents.

- Test Case Specification
- Test Data Requirements
- Test Data Readiness Report
- Test Environment Requirements
- Test Environment Readiness Report

5.5 Executing the Plan

The test plan should be executed as designed. If the plan cannot be executed as designed, it should be changed. Testing according to the test plan should commence when the project commences and conclude when the software is no longer in operation. Portions of the test plan can be performed while the test plan is being written. To carry out the test plan, testers require many skills including designing test cases and test scripts, proper and efficient use of test tools, execute tests, recording test results, and managing defects.
The principle of integrated quality control throughout the development life cycle is a recurring theme in Software Testing Body of Knowledge (STBOK). Section 1.8 of Skill Category 1 described the different activities involved with full life cycle testing. Section 1.8.2.1 defines verification testing along with providing several examples. In this skill category, an in depth discussion about these verification techniques is provided.

A review is a quality control technique that relies on individuals other than the author(s) of the deliverable (product) to evaluate that deliverable. The purpose of the review is to find errors before the deliverable; for example, a requirements document is delivered either to the customer or the next step of the development cycle.

Review tasks should be included in the project plan. Reviews should be considered activities within the scope of the project and should be scheduled and resources allocated just like any other project activity.

<table>
<thead>
<tr>
<th>Purpose of Reviews</th>
<th>6-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Review Types</td>
<td>6-4</td>
</tr>
<tr>
<td>Prerequisites to Reviews</td>
<td>6-6</td>
</tr>
<tr>
<td>Conducting Walkthroughs</td>
<td>6-7</td>
</tr>
<tr>
<td>Checkpoint Reviews</td>
<td>6-9</td>
</tr>
<tr>
<td>Conducting Inspections</td>
<td>6-10</td>
</tr>
<tr>
<td>Summary</td>
<td>6-13</td>
</tr>
</tbody>
</table>
6.1 Purpose of Reviews

There are four primary purposes for conducting reviews:

- Emphasize quality throughout the software development life cycle
- Detect defects when and where they are introduced known as phase containment
- Provide an opportunity to involve the end user/customer in the development process
- Permit “midcourse” corrections

6.1.1 Emphasize Quality throughout the SDLC

All software development projects have four factors which must be continuously monitored and adjusted in order to maintain control of the development process. They are “scope, schedule, resources, and quality” (see Figure 6-1). These factors can be viewed as interrelated dials on a control panel. When one dial is adjusted, one or more of the others must be adjusted to compensate for the change.

Reviews, which emphasize the quality of the products produced, can be performed throughout the SDLC ensuring that the “quality” factor is given as much priority as the other three.

![Interrelated Factors in the SDLC](image)

Figure 6-1 Interrelated Factors in the SDLC

6.1.2 Detect Defects When and Where they are Introduced

Industry studies show that in most organizations, more defects originate in the specifications process (i.e., requirements analysis and design) than in the coding process (see Figure 6-2). In
other words, most defects are inserted early in the life cycle. Reviews, unlike traditional quality control techniques (testing), can be conducted during these crucial early stages.

Finding and correcting defects soon after they are introduced not only prevents ‘cascading’ defects later in the life cycle but also provides important clues regarding the root cause of the defect, which explains how and why the defect happened in the first place.

---

**Figure 6-2  Origin of Defects**

---

### 6.1.3 Opportunity to Involve the End User/Customer

Customer involvement during the development process is essential. The traditional role of the customer has been to provide the functional requirements for the system. Reviews give customers an opportunity for involvement throughout the life cycle. Such involvement includes confirming/verifying that deliverables such as the statement of requirements, functional specification, external design, test plans and user manual satisfy their requirements. Regardless of the development framework (i.e., waterfall, iterative), the review process provides the needed structure for the deploying reviews throughout the SDLC.

### 6.1.4 Permit “Midcourse” Corrections

It seems no matter how much time and effort is devoted to the front end of the development life cycle, there are always changes and adjustments along the way. Customers change their minds as business environments dictate new demands. Reviews are consistent with iterative development models which recognize the need for embracing change as a normal occurrence during the development process.

Because reviews are performed throughout the life cycle, they support these approaches in two ways: they add user perspective at various points during development and they stop unwanted/wrong functions from being developed.
6.2 Review Types

Depending on the objectives of the review, the review structure, the level of formality, and the participants involved will vary. Review types include desk checks, walkthroughs, checkpoint reviews, and inspections. There are also sub-categories such as peer-to-peer reviews (that we will discuss in this course) which are suitable for use in the software development process.

6.2.1 Desk Checks

Desk checks are a simple form of review in which the author of a work product distributes it to one or more selected reviewers. The team members read it and provide written feedback on defects found.

6.2.2 Walkthroughs

Informal reviews are usually conducted by the author of the product under review. Walkthroughs do not require advance preparation. Walkthroughs are typically used to confirm understanding, test ideas, and brainstorm.

The walkthrough process consists of three steps. The first step involves selecting the walkthrough participants; the second step is the review meeting or “walk through”; and the third step involves using the results.

6.2.3 Checkpoint Reviews

Reviews held at predefined points in the life cycle which evaluate whether certain quality factors are being adequately addressed in the system are called Checkpoint Reviews.

The Checkpoint Review process consists of two phases. The first phase is a planning phase which occurs at the beginning of each project that has been targeted for checkpoint reviews in which the checkpoints are identified. The second phase includes the steps for conducting a checkpoint review. This phase is iterated for each checkpoint review held during the project. Figure 6-3 illustrates the planning phase and the iterative “checkpoint” phase.
6.2.4 Inspections

Inspections are used to review an individual product and to evaluate correctness based on its input criteria (specifications).

The inspection has six steps. The first three steps involve the inspection team prior to the actual review meeting. The fourth step is the inspection of the product. The fifth step is performed by the producer after the product has been reviewed or “inspected.” The last step assures that the inspection findings have been addressed in the product. Figure 6-4 illustrates the inspection process.
6.3 Prerequisites to Reviews

There are five major prerequisites that must be in place for a review program to be successful. Figure 6-5 illustrates these five prerequisites.

![Figure 6-5 Five Prerequisites for a Successful Review Program](image)

6.3.1 A System Development Methodology

Reviews focus on documentation. Without consistent documentation products, reviews are not practical. The more well-defined the deliverables are, the more review options available. For example, walkthroughs can be used for poorly defined deliverables, but inspections can only be used for well-defined deliverables.

6.3.2 Management Support

Reviews take commitment of time and resources. The following techniques can be used to get management onboard:

- Review the economics of reviews
- Enlist the support of customers. Explain to customers the economic and schedule advantages that occur with reviews and solicit support for the process.
- Try two or three pilots to demonstrate effectiveness
6.3.3 Review Process

A well-defined process for conducting reviews will need to be installed. The details for this process are contained later in this skill category.

6.3.4 Project Team Support

To get other team members onboard:

- Conduct a pilot project to demonstrate the benefits of reviews
- Show management support. If reviews are viewed as important to management, the team members throughout the SDLC will support them.
- Use the results of reviews constructively
- DO NOT evaluate the individual based on the Quality of or the number of defects found during the review of their product
- DO make the following a part of the staff evaluation process:
  - Willingly participates in reviews
  - Adequately prepares for reviews
  - Follows the process, completes assignments on time, shows up for meetings, etc.
- Ensure adequate time has been allocated in the schedule to conduct reviews
- Make using reviews a win-win situation. The project leader and team should be thanked for conducting the review regardless of the outcome of the review.
- Find a reward to give to all participants during any review trial period

6.3.5 Training

People cannot conduct effective reviews unless they are trained in how to do it! Provide initial concepts, overview, and skills training as well as ongoing follow-up coaching.

6.4 Conducting Walkthroughs

The walkthrough process primarily utilizes the experience of the walkthrough team in reviewing ideas or products (deliverables). The process is informal and usually conducted by the author of the product. The informality of the process makes it difficult to achieve consistent results. Walkthroughs are most effective during the creative part of any development phase. Some organizations use a little more formal approach when conducting a walkthrough in that they may actually capture defects.

Some potential uses are:
Informally critique initial requirements, design, and implementation ideas (peer-oriented)
Informally present plans, proposals, or other deliverables for review and approval (management involvement)

The three steps in the walkthrough process are: (1) selecting a walkthrough team, (2) conducting the walkthrough meeting, and (3) the follow-up to the meeting.

### 6.4.1 Select the Walkthrough Team

Successful team selection is dependent on finding the right people for the walkthrough. The best candidates are knowledgeable, skilled, available, and respected. The informality of the process means that an individual’s knowledge and willingness to improve the product are the most important factors to be considered. Always select individuals who can provide constructive input to the product.

Emphasizing the informality of the process, pulling the team together need only be done with enough notice to allow participants time to schedule the meeting and, if available, do a quick review of the meeting materials. Given the busy schedules of people, be sure to clear a participant’s availability with them and their manager. It is always recommended that a follow-up confirmation be sent as a reminder to all involved.

Although a management level individual (e.g., project manager, test manager) will normally be responsible for this task, there is certainly nothing wrong with another team member doing all the coordination of available dates, times, and meeting places.

### 6.4.2 Conduct the Walkthrough Meeting

The walkthrough meeting is an opportunity for interested participants to review a product or deliverable, ask/answer questions, discuss issues, create action items, and gather ideas and recommendations.

### 6.4.3 Walkthrough Meeting Follow-Up

Important to future walkthroughs and people’s positive impression of the review process is the follow-up. Not only should the issues, recommendations, etc. be acted upon, but it is imperative that copies of that information and the resolutions be prepared and given to all the review team members. Follow-up should happen as soon after the meeting as possible.
6.5 Checkpoint Reviews

Checkpoint Reviews are held at predefined points in the development process in order to evaluate a system as it is being specified, designed, tested, and implemented. These reviews focus on whether certain quality factors are being adequately addressed in the system.

The Checkpoint Review Process consists of two phases. The first phase is planning and preparing, and is done once for each project that has been targeted for Checkpoint Reviews. The second phase includes steps for conducting the reviews. This phase is iterated for each checkpoint review held during the project. Listed below are the two phases and six-steps for checkpoint reviews.

- **Phase 1 – Planning (1 time)**
  - Step 1: Establish Objectives
  - Step 2: Develop Plan
  - Step 3: Kickoff Review Process

- **Phase 2 – Checkpoints (repeated for every defined Checkpoint)**
  - Step 4: Review System
  - Step 5: Conduct Meeting
  - Step 6: Follow-up

6.5.1 Phase I – Planning

This is the organizing phase for all the checkpoint reviews being held during a specific project. The planning phase is a three step process. The three steps are:

- Step 1 – Establish Objectives
- Step 2 – Develop Plan
- Step 3 – Kickoff Review Process

6.5.2 Phase II – Checkpoints

The second phase in the Checkpoint Process is the actual conducting of the reviews. With the review team in place, assignments made, and the kickoff meeting completed, everything is ready for the first Checkpoint Review. This phase is iterative throughout the system life cycle process and is conducted as many times as set forth in the project plan. The three steps in this phase are:

- Step 4 – Review System
- Step 5 – Conduct Meeting
- Step 6 – Follow-up
6.5.3  Sharing Data with Quality Assurance

All review reports/worksheets should be made available to quality assurance for analysis and summation. Note that quality assurance should not disclose which project exhibited which type of defects.

6.6  Conducting Inspections

6.6.1  What are Inspections?

An inspection evaluates the correctness of a single product such as a database or a program module. The evaluation is based on whether or not the product satisfies its specifications or entrance criteria.

6.6.2  The Background on Inspections

The inspection concept was developed by Michael Fagan at IBM in the mid 1970’s. It has become widely accepted that inspections are the most effective defect removal technique. Inspections have been shown to be five times more effective than unit testing. It should be noted that not all products need to be inspected. Most companies begin by inspecting code although any project deliverable can be inspected.

6.6.3  Inspection Process

The Inspection Process consists of six steps. Listed below is the six step process for inspections:

- Step 1 – Planning
- Step 2 – Overview
- Step 3 – Preparation
- Step 4 – Inspection
- Step 5 – Rework
- Step 6 – Follow-up

6.6.3.1  Planning – Step 1

The purpose of this step is to prepare for the inspection. To achieve this, the following three tasks must be completed:
1. Select the Inspection Team Members

2. Train the Inspection Team

3. Schedule the Inspection

**Roles & Responsibilities**

There are six distinct “roles” on the inspection team: the moderator, reader, recorder, producer/author, inspector, and standardizer. Listed below are the responsibilities of each.

1. **Moderator:**
   - Manages the inspection process
   - Accountable for the effectiveness of the inspection
   - Must be impartial

2. **Reader:**
   - Must understand the material
   - Paraphrases the material during the inspection
   - Sets the inspection pace

3. **Recorder:**
   - Must understand error classification
   - Is not the meeting stenographer (captures enough detail for the project team to go forward to resolve errors)
   - Classifies errors as detected
   - Reviews the error list at the end of the meeting

4. **Producer/Author:**
   - Gathers and distributes materials
   - Provides product overview
   - Is available for clarification
   - Should contribute as an inspector*
   - Must not be defensive

5. **Inspectors:**
   - ALL participants are inspectors**
   - Must be prepared
   - Inspect the product, NOT the producer
   - Must be objective and supportive

6. **Standardizer:**
   - Must know IT standards & procedures
Ensures standards are met and procedures are followed
Meets with project leader/manager
Ensures entrance criteria are met (product is ready for review)

Note: It is possible that the producer will not be at the meeting but they must be available for clarification. Also, be aware of the challenges of wearing multiple hats; it’s hard to do two things well at the same time.

### 6.6.3.2 Overview – Step 2

The objective of Step 2 is to deliver and explain the product being inspected to the inspectors. It is the responsibility of the producer to deliver the product and this should be done three or more working days prior to the inspection meeting. To achieve this, the following two tasks must be completed:

1. Distribute Review Materials
2. Provide Tutorial

### 6.6.3.3 Preparation – Step 3

The objective of Step 3 is to prepare for the inspection meeting by reviewing the inspection materials for defects. This step should be performed 1-3 days prior to the inspection meeting. There are three tasks to complete in this step:

1. Perform Individual Inspections
2. Record Defects
3. Assure Readiness for Meeting

### 6.6.3.4 Inspection – Step 4

During this step, hold the inspection meeting to develop a final defect list. This list should represent team consensus. There are four tasks to complete in this step:

1. Inspection Administration
2. Read Product
3. Record Defects
4. Determine Disposition
6.6.3.5 Rework – Step 5

If the moderator determines that the product required rework, he/she will deliver to the author the completed Inspection Report and Inspection Defect List.

It is the responsibility of the author to make the necessary corrections and present the product to the moderator for reexamination or re-inspection depending on the results from Step 4.

6.6.3.6 Follow-up – Step 6

This final step in the inspection process is for the moderator to verify that all defects have been corrected and, if so, certify the product. This can be done so as soon as any rework is complete.

6.7 Summary

It has been stated continuously throughout the STBOK that the most effective approach to managing the cost of defects is to find them as early as possible. Walkthroughs, checkpoint reviews, and inspections, which are often collectedly referred to as “reviews,” are the techniques for delivering this early detection. Reviews emphasize quality throughout the development process, involve users/customers, and permit ‘midcourse’ corrections.
Designing Test Cases

The test objectives established in the test plan should be decomposed into individual test conditions, then further decomposed into individual test cases and test scripts. In Skill Category 5, the test plan included a test matrix that correlates a specific software function to the tests that will be executed to validate that the software function works as specified.

When the objectives have been decomposed to a level that the test case can be developed, a set of tests can be created which will not only test the software during development, but can test changes during the operational state of the software.

<table>
<thead>
<tr>
<th>Skill Category 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

### 7.1 Identifying Test Conditions

The first step in defining detailed test cases that will be used in validation testing is to generate a list of the conditions to be tested. Test conditions describe the specific functional capability, structural design consideration, features, and validation checks which need to be validated.

It is a common practice to use the system specifications (e.g., requirements documents, high level design documents) as the primary source for test conditions. After all, the system specifications are supposed to represent what the application system is going to do. The more
specific the documents are the better the test conditions. However, as was noted in Skill Category 4, (Risk) section 4.1.1.3 “Requirements are the most significant product risks reported in product risk assessments.” The concern is that if we rely too heavily on system specifications, namely the requirements documentation, we will likely miss significant conditions that must be tested. To help mitigate that risk, the application under test must be viewed from as many perspectives as possible when identifying the test conditions.

When identifying test conditions for the application under test, there are five perspectives that should be utilized to ferret out as many testable conditions as possible and to reduce the risk of missing testable conditions. These five perspectives are:

1. the system specifications (specification decomposition)
2. the production environment (population analysis)
3. a predefined list of typical conditions to be tested (test transactions types)
4. business case analysis (business process analysis)
5. structural analysis (source code or technical design analysis)

It is important to follow a logical process when identifying test conditions. This is not to say that ad hoc approaches such as Exploratory Testing (see section 1.9.2.4) are not used or useful when identifying test conditions. Quite the contrary, the process of identifying testable conditions is precisely the process of exploring the application from a variety of viewpoints. As testable conditions are discovered they provide more information about the application and this knowledge uncovers more testable conditions. Software applications can be very complex and the risks associated with missing testable conditions are high.

### 7.1.1 Defining Test Conditions from Specifications

The first perspective is the most common; identifying test conditions from system specifications. Unfortunately, there is no easy step-by-step process which can be used for deriving test conditions from narrative specifications. A first step is to do functional decomposition which isolates the pertinent sections of the application. Once that is completed the respective documents should be reviewed looking for such items as:

- business rules
- capabilities
- causal relationships
- data relationships
- effect relationships
- features
- inputs
- objects
- outputs
• processing functions
• timings
• validations

7.1.2 Defining Test Conditions from the Production Environment

This second approach presupposes that there is production data and that it represents the types of transactions that will be processed by the applications under test. The following types of files, record sets, and tables meet those criteria:

- Existing production files or tables being used as is in the software under test
- Existing production files or tables for which there will be minor changes to the file in the software under test
- Production files or tables that contain approximately the same field/data elements that will be included in the software being tested
- Existing manual files from other systems which contain approximately the same data elements that will be included in the files of the software being tested

7.1.2.1 Population Analysis

If you plan to use production data for test purposes, population analysis is a technique used to identify the kinds and frequency of data that will be found in the production environment. Population analysis is creating reports that describe the type, frequency, and characteristics of the data to be used in testing. For example, for numerical fields, one might want to know the range of values of data contained in that field; for alphabetic data, one may want to know the longest name in a data field; for codes, one may want to know what codes and their frequency of use.

7.1.2.1.1 The Benefit of Population Analysis

Testers will benefit from using population analysis in the following ways:

- Identification of codes/values being used in production which were not indicated in the software specification
- Unusual data conditions, such as a special code in a numeric field
- Provides a model for use in creating test transactions/test scripts
- Provides a model for the type and frequency of transactions that should be created for stress testing
- Helps identify incorrect transactions for testing error processing/error handling routines
7.1.2.1.2 How is Population Analysis Performed?

Population analysis is best performed using a software tool designed to perform ad hoc reports on a database. It can be performed manually but rarely does that permit the full analysis of large-volume files.

7.1.2.1.3 Population Analysis Data Needed

There are three types of population analyses you may wish to perform.

1. **File/Database/Table population analysis** - The objective of this analysis is to identify all of the files, databases, and tables used by application under test, and to gather some basic data about each.

2. **Screen population analysis** - The objective of this is to identify all of the screens that will be used by the application under test, and to gather some background data about each screen (i.e., take screen shots and document).

3. **Field/data element population analysis** - The objective of this is to document the characteristics and frequencies of fields/data elements. This will be the most complex and time-consuming part of population analysis.

7.1.3 Defining Test Conditions from Test Transaction Types

The third approach to identifying test conditions is based on the reality that software has certain characteristics unique to the discipline. The unique characteristics are referred to as Test Transaction Types. Here we identify thirteen different transaction types used to develop test conditions (see Table 7-1). Some of these thirteen categories are narrow, while others are broad in scope. For example, the “search” type involves looking for a specific data field is a very limited area for creating test conditions/cases; while the “error” type category is very broad and could result in literally hundreds or even thousands of test cases. Some of the conditions identified from these 13 transaction types will be more closely aligned with unit testing, others with integration testing and yet others with system testing. However, all individuals involved in the testing of an application should be aware of the conditions associated with all 13 types.

The 13 types of transactions are listed and briefly described on the following pages. For each transaction type the test concern is explained, the area responsible for developing those transaction types is listed, and an approach describing how to create test data of that type is included. The first two transaction types, field and record, include a list of questions to ask when looking for testable conditions. A full list of questions that could be asked for all thirteen types are included as Appendix C.
### Table 7-1 Test Transaction Types

<table>
<thead>
<tr>
<th>Transaction Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIELD</td>
<td>Test coverage related to the attributes of an individual field/data element.</td>
</tr>
<tr>
<td>RECORD</td>
<td>Tests related to the entry, storage, retrieval, and processing of records.</td>
</tr>
<tr>
<td>FILE/DATABASE</td>
<td>Test conditions related to the opening, connecting, using, and closing a file or database.</td>
</tr>
<tr>
<td>RELATIONSHIP</td>
<td>Test transactions dealing with two or more related fields/data elements.</td>
</tr>
<tr>
<td>ERROR</td>
<td>Conditions involving inaccurate, incomplete, or obsolete data.</td>
</tr>
<tr>
<td>USE (Outputs)</td>
<td>Validating the ability to enter the proper data based on end user instructions and to take appropriate actions based on the output provided by the application under test.</td>
</tr>
<tr>
<td>SEARCH</td>
<td>The ability to find data or a record.</td>
</tr>
<tr>
<td>MATCH/MERGE</td>
<td>The ability to properly interface two or more records when their processing must be concurrent or in a predefined sequence.</td>
</tr>
<tr>
<td>STRESS</td>
<td>Subjecting all aspects of application system to their performance limits.</td>
</tr>
<tr>
<td>CONTROL</td>
<td>Ensuring the accurate, complete, and authorized processing of transactions.</td>
</tr>
<tr>
<td>ATTRIBUTES</td>
<td>Testing quality states such as performance, reliability, efficiency, and security.</td>
</tr>
<tr>
<td>STATES</td>
<td>Verifying correct performance in a variety of conditions such as an empty master file and missing or duplicate master records.</td>
</tr>
<tr>
<td>PROCEDURES</td>
<td>Determining that the application under test can be appropriately operated in the areas such as start-up, backup, and recovery.</td>
</tr>
</tbody>
</table>

*Table 7-1 Test Transaction Types*

#### 7.1.3.1 What Transaction Types are Most Important?

Testers never have enough time to create all the test data needed. Thus, there will be tradeoffs between the extensiveness of the test conditions in each category. Using the tester’s background and experience, the results of interaction with end users/project personnel, and the risk assessment, the tester should indicate for each transaction type whether it is high, medium, or low importance. These are of relative importance, meaning that approximately
one-third of all transaction types should be high, one-third, medium, and one-third, low. The purpose of this is to indicate to those creating test conditions where emphasis should be placed.

### 7.1.3.2 Test Transaction Type: FIELD

This test is limited to a specific field/data element. The purpose is to validate that all of the processing related to that specific field is performed correctly. The validation will be based on the processing specifications for that specific field. However, it will be limited to that specific field and does not include a combination or interrelationship of the field being tested with other fields.

#### 7.1.3.2.1 Testing Concern

The major concern is that the specifications relating to processing of a single field/data element will not have been implemented correctly. The reason for this is the error of omission. Specific field conditions properly documented in requirements and design may not have been correctly transferred to program specifications, or properly implemented by the programmer. The concern is one of accuracy of implementation and completeness of program specifications for the specific field.

#### 7.1.3.2.2 Responsibility for Test Type

The owner of the field is responsible for the accuracy/completeness of processing for the field. The tester may want to verify with the owner that the number of conditions to validate the accuracy/completeness of field processing is complete.

#### 7.1.3.2.3 Test Condition Creation Approach

The following three-step approach is recommended to develop the test conditions for a field:

1. Create Test Conditions from Program Specifications
2. Review Requirement and Design Specifications
3. Verify Completeness of Test Conditions with Owner

#### 7.1.3.2.4 Examples

Field edits, field updates, field displays, field sizes, and invalid values processing are examples.

#### 7.1.3.2.5 Questions to Ask
### Table 7-2 FIELD: Questions to Ask

<table>
<thead>
<tr>
<th>#</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Have all codes been validated?</td>
</tr>
<tr>
<td>2.</td>
<td>Can fields be properly updated?</td>
</tr>
<tr>
<td>3.</td>
<td>Is there adequate size in the field for the accumulation of totals?</td>
</tr>
<tr>
<td>4.</td>
<td>Can the field be properly initialized?</td>
</tr>
<tr>
<td>5.</td>
<td>If there are restrictions on the contents of the field, are those restrictions validated?</td>
</tr>
<tr>
<td>6.</td>
<td>Are rules established for identifying and process invalid field data?</td>
</tr>
<tr>
<td></td>
<td>a. If no, develop this data for the error-handling transaction type.</td>
</tr>
<tr>
<td></td>
<td>b. If yes, have test conditions been prepared to validate the specification processing for invalid field data?</td>
</tr>
<tr>
<td>7.</td>
<td>Have a wide range of normal valid processing values been included in the test conditions?</td>
</tr>
<tr>
<td>8.</td>
<td>For numerical fields, have the upper and lower values been tested?</td>
</tr>
<tr>
<td>9.</td>
<td>For numerical fields, has a zero value been tested?</td>
</tr>
<tr>
<td>10.</td>
<td>For numerical fields, has a negative test condition been prepared?</td>
</tr>
<tr>
<td>11.</td>
<td>For alphabetical fields, has a blank condition been prepared?</td>
</tr>
<tr>
<td>12.</td>
<td>For an alphabetical/alpha numeric field, has a test condition longer than the field length been prepared? (The purpose is to check truncation procession).</td>
</tr>
<tr>
<td>13.</td>
<td>Have you verified from the data dictionary that all valid conditions have been tested?</td>
</tr>
<tr>
<td>14.</td>
<td>Have you reviewed systems specifications to determine that all valid conditions have been tested?</td>
</tr>
<tr>
<td>15.</td>
<td>Have you reviewed requirements to determine all valid conditions have been tested?</td>
</tr>
<tr>
<td>16.</td>
<td>Have you verified with the owner of the data element that all valid conditions have been tested?</td>
</tr>
</tbody>
</table>

### 7.1.3.3 Test Transaction Type: RECORD

These conditions validate that records can be properly created, entered, processed, stored, and retrieved. The testing is one of occurrence, as opposed to correctness. The objective of this test is to check the process flow of records through applications.
7.1.3.3.1 Testing Concern

The primary concern is that records will be lost during processing. The loss can occur prior to processing, during processing, during retention, or at the output point. Note that there is a close relationship between record loss and control. Control can detect the loss of a record, while testing under this transaction type has as its objective to prevent records from being lost.

7.1.3.3.2 Responsibility for Test Type

The originator of each record has the responsibility to determine that all records have been processed. Individuals having custodial responsibilities, such as data base administrators, are responsible to see that records are not lost while in storage. Individuals having responsibility to take action on outputs have the responsibility to determine that those actions are taken. However, individuals having output responsibility may not be responsible for knowing what records are to be output.

7.1.3.3.3 Test Condition Creation Approach

The creation of record tests requires some sort of data flow diagram/data model in order to understand the logical flow of data. The creation of record transaction type test conditions is a three-step process, as follows:

1. Create/Use Data Flow Diagram/Data Model
2. Create Record Condition Types
3. Assess the Completeness of the Record Condition Types

7.1.3.3.4 Examples

First and last records, multiple records, and duplicate records.

7.1.3.3.5 Questions to Ask

<table>
<thead>
<tr>
<th>#</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Has a condition been prepared to test the processing of the first record?</td>
</tr>
<tr>
<td>2.</td>
<td>Has a condition been determined to validate the processing of the last record?</td>
</tr>
<tr>
<td>3.</td>
<td>If there are multiple records per transaction, are they all processed correctly?</td>
</tr>
<tr>
<td>4.</td>
<td>If there are multiple records on a storage media, are they all processed correctly?</td>
</tr>
<tr>
<td>5.</td>
<td>Can two records with the same identifier be processed (e.g., two payments for the same accounts receivable invoice number)?</td>
</tr>
</tbody>
</table>
### Test Transaction Type: FILE

These conditions validate that all needed files are included in the system being tested and that the files will properly interconnect with the modules that need data from those files. Note that file is used in the context that it can be any form of file or database table.

#### Testing Concern

The test concern is in the areas normally referred to as integration and system testing. It is determining that the operating environment has been adequately established to support the application’s processing. The concerns cover the areas of file definition, creation of files, and the inter-coupling of the files with and between modules. Note that file conditions will also be covered under the transaction types of search, match/merge, and states.

#### Responsibility for Test Type

Some of the file test conditions must be done by the developers during Unit testing. However, it is a good idea for everyone involved in testing the application be aware of these test transaction types. Rarely would end users/customers have the ability to understand or validate file processing.

#### Test Condition Creation Approach

The preparation for file testing is very similar to the preparation performed by system programmers to create the file job control. However, the testers are concerned with both the system aspect of testing (i.e., job control works) and the correct interaction of the files with the modules. Note that software that does not use job control has equivalent procedures to establish interaction of the file to the modules.

The following three-step process will create the conditions necessary to validate file processing:

<table>
<thead>
<tr>
<th>#</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.</td>
<td>Can the first record stored be retrieved?</td>
</tr>
<tr>
<td>7.</td>
<td>Can the last record stored be retrieved?</td>
</tr>
<tr>
<td>8.</td>
<td>Will all of the records entered be properly stored?</td>
</tr>
<tr>
<td>9.</td>
<td>Can all of the records stored be retrieved?</td>
</tr>
<tr>
<td>10.</td>
<td>Do interconnecting modules have the same identifier for each record type?</td>
</tr>
<tr>
<td>11.</td>
<td>Are record descriptions common throughout the entire software system?</td>
</tr>
<tr>
<td>12.</td>
<td>Do current record formats coincide with the formats used on files created by other systems?</td>
</tr>
</tbody>
</table>

*Table 7-3 RECORD: Questions to Ask*
1. Document System File Flows
2. Develop File Test Conditions
3. Assess the Completeness of the File Test Condition Creation

7.1.3.4.4 Examples
File version, file opening and closing, and empty files.

7.1.3.5 Test Transaction Type: RELATIONSHIP
The objective of this test transaction category is to test relationships between data elements. Note that record and file relationships will be tested in the match/merge transaction type. Relationships will be checked both within a single record, and between records. Relationships can involve two or more fields. The more complex the relationship, the more difficult “relationship” testing becomes.

7.1.3.5.1 Testing Concern
The major concern is that illogical processing will occur because relationships have not been validated. For example, an individual in a specific pay grade could be paid a higher wage than entitled to in that pay grade if the relationship between pay rate and pay grade was not validated. The concern is to first identify these relationships, and then determine that there is adequate relationship checking in the system.

7.1.3.5.2 Responsibility for Test Type
The individual owners of the related data elements jointly share responsibility for relationship checking. They should understand the relationships and be able to verify the completeness of relationships checking.

7.1.3.5.3 Test Condition Creation Approach
Relationship testing is one of the more complex aspects of testing. Relationships tend to be implied in system/program specifications, but rarely stated specifically. Thus, the testers not seeing those relationship specifications may not properly test the relationships without extra effort.

The creation of relationship test conditions is a four-step process, as follows:
1. Create a data element relationship test matrix
2. Verify the reasonableness of the identified relationships
3. Create data relationship test conditions
4. Assess the Completeness of Relationship Test Condition Creation
7.1.3.5.4 Examples

Values limit values, date and time limit values, absence-absence, presence-presence, and values out of line with the norm.

7.1.3.6 Test Transaction Type: ERROR

This condition tests for errors in data elements, data element relationships, records and file relationships, as well as the logical processing conditions. Note that if error conditions are specified, then the test for those would occur under the test transaction type indicated. For example, if the specifications indicated what was to happen if there were no records on a file, then under the file transaction type an error test condition would be included. However, if error condition processing is not specified, then this category would be all-inclusive for the non-specified conditions.

7.1.3.6.1 Testing Concern

There is a major test concern that the system will be built to handle normal processing conditions and not handle abnormal processing conditions appropriately.

7.1.3.6.2 Responsibility for Test Type

The responsibility for incorrect processing conditions is rarely defined. It is recommended that the functional test team and users/customers have responsibility for errors related to functionality; for example, inaccurate, incomplete, or obsolete data; while the developers have responsibility for structural errors, for example, those relating to program and system architecture.

7.1.3.6.3 Test Condition Create Approach – ERROR

It is virtually impossible to specify all of the potential error conditions that could occur in any application system. The best that can be achieved is to identify the most frequent and most serious error conditions. Test conditions will be prepared for these to validate that the system can handle them properly. As new errors occur, the system should be modified to protect against those error conditions, and the set of test conditions should be increased to validate that the system has been adequately modified to handle those conditions.

The creation of error test transaction conditions is a four-step process, as follows:

1. Conduct brainstorming sessions with end users/customers to identify functional error conditions
2. Conduct brainstorming sessions with project personnel for structural error conditions
3. Rank and select conditions for testing
4. Evaluate the completeness of the error conditions
7.1.3.6.4 **Examples**

Incomplete data, improper input data, and incorrect calculations.

7.1.3.7 **Test Transaction Type: USE**

This condition tests the ability of the end user to effectively utilize the system. It includes both an understanding of the system outputs, as well as the ability of those outputs to lead to the correct action. This requires the tester to go beyond validating specifications to validating that the system provides what is needed to lead the end user to the correct business action.

7.1.3.7.1 **Testing Concern**

Two major concerns are addressed by these test conditions. The first is that the end user of the system will not understand how the system is to be used, and thus will use it incorrectly. The second concern is that the end user will take the wrong action based on the information provided by the system. For example, in a bank, a loan officer may make a loan based on the information provided, when in fact a loan officer should not have made the loan.

7.1.3.7.2 **Responsibility for Test Type**

The end user/customer has ultimate responsibility for these test conditions. However, the tester understanding the process used to develop the information delivered to the end user should have a secondary responsibility to assist the end user in preparing these test conditions.

7.1.3.7.3 **Test Condition Creation Approach**

The test approach for this condition requires the identification of the business actions taken. They then must be related to the information to determine that the information leads to the logical business action. This test tends to be a static test, as opposed to a dynamic test. It is a four-step process, as follows:

1. Identify the business actions
2. Identify the system outputs contributing to those business actions
3. Indicate the relationship between the system output and the business action taken
4. Assess the completeness of the tests

7.1.3.7.4 **Examples**

Inventory screen used to communicate quantity on hand, credit information aides in the loan approval process, and customer inquiry provides phone number for payment follow-up.
7.1.3.8 Test Transaction Type: ABILITY TO SEARCH FOR DATA

Search capabilities involve locating records, fields, and other variables. The objective of testing search capabilities is to validate that the search logic is correct. Search activities can occur within a module, within a file, or within a database. They involve both simple searches (i.e., locating a single entity) or complex searches (e.g., finding all of the accounts receivable balances over 90 days old). Searches can be preprogrammed or special-purpose searches.

7.1.3.8.1 Testing Concern

There are two major concerns over search capabilities. The first is that the preprogrammed logic will not find the correct entity, or the totality of the correct entities. The second is that when "what if" questions are asked the application will not support the search.

7.1.3.8.2 Responsibility for Test Type

The functional test team and users of the information (during UAT) are responsible to validate that the needed search capabilities exist, and function correctly. However, because this involves both functionality and structure, the development team has a secondary responsibility to validate this capability.

7.1.3.8.3 Test Condition Creation Approach

Two approaches are needed for testing the search capabilities. The first is validating that the existing logic works and the second is verifying that the “what if” questions can be answered by expending reasonable effort.

There are four steps needed to test the search capabilities, as follows:

1. Identify the specified search capabilities
2. Identify potential one-time searches
3. Create search test conditions
4. Validate the completeness of the search conditions

7.1.3.8.4 Examples

File search for a single record, file search for multiple records, multiple condition searches (e.g., customers in USA only with over $1 million in purchases), and table searches.

7.1.3.9 Test Transaction Type: MATCHING/MERGING CONDITIONS

Matching and merging are various file processing conditions. They normally involve two or more files, but may involve an input transaction and one or more files, or an input transaction and an internal table. The objective of testing the match/merge capabilities is to ensure that all of the combinations of merging and matching are correctly addressed. Generally, merge
inserts records into a file or combines two or more files; while match searches for equals between two files.

7.1.3.9.1 Testing Concern

Because of the complexity of merge/match logic, it is a frequent cause of improper processing. There are two major concerns. One is that the general match/merge logic does not work correctly; and the second is that specific match/merge conditions do not work. Note that these special conditions are addressed in this transaction type, and in the “states” transaction type.

7.1.3.9.2 Responsibility for Test Type

The responsibility for correct matching and merging resides with the designers of the application system. They should create the logic that permits all of the various conditions to be performed. Testers must validate that the logic is correct.

7.1.3.9.3 Test Condition Creation Approach

The approach to testing match/merge conditions is a standardized approach. Generally the application is not a consideration in developing test conditions for this capability. Some examples are:

a. Merge/match of records of two different identifiers (inserting a new item, such as a new employee on the payroll file)

b. A merge/match on which there are no records on the merged/matched file

c. A merge/match for which there are no input file/transactions being merged/matched

d. A merge/match in which the first item on the file is deleted

The approach to developing these test conditions is a three-step approach, as follows:

1. Identify match/merge conditions

2. Create test conditions for each merge/match condition

3. Perform merge/match test conditions completion check

7.1.3.9.4 Examples

Record additions, duplicates, no matches, and out of sequence records.

7.1.3.10 Test Transaction Type: STRESS

Stress testing is validating the performance of application system when subjected to a high volume of transactions. Stress testing has two components. One is a large volume of transactions; and the second is the speed at which the transactions are processed (also referred to as performance testing which covers other types like spike and soak testing). Stress testing can apply to individuals using the system, the communications capabilities associated with the
system, as well as the system's capabilities itself. Any or all of these conditions can be stress tested.

7.1.3.10.1 Testing Concern

The major test concern is that the application will not be able to perform in a production environment. Stress testing should simulate the most hectic production environment possible.

7.1.3.10.2 Responsibility for Test Type

Stress testing is an architectural/structural capability of the system. The system may be able to produce the right results but not at the performance level needed. Thus, stress test responsibility is a development team responsibility.

7.1.3.10.3 Test Condition Creation Approach

Stress testing is normally associated with web-based or mobile applications. Stress testing as applied to most batch systems is considered volume testing. Volume testing is directed more at the determining hardware/program limitations than it is performance.

There are two attributes of computer systems involved in stress testing. One is continuity of processing, and the other is service level. Continuity of processing can be impacted by large volumes of data, while stress testing normally affects service level. Systems can fail stress testing, but still produce accurate and complete results.

Stress testing involves the following five steps:

1. Identify performance capabilities
2. Identify system feature capability
3. Determine the impact of the system features on the performance capabilities desired
4. Develop test conditions to stress the features that have a direct contribution on system performance
5. Assess completeness of stress conditions

7.1.3.10.4 Examples

Website function response time, report turnaround time, and mobile application response time.

7.1.3.11 Test Transaction Type: CONTROL

Control testing validates the adequacy of the system of internal controls to ensure accurate, complete, timely, and authorized processing. These are the controls that are normally validated by internal and external auditors in assessing the adequacy of control. A more detailed discussion of Testing Controls can be found in Skill Category 8.
7.1.3.11.1 Testing Concern

Controls in this context are application controls, and not management controls. The concern is that losses might accrue due to inadequate controls. The purpose of controls is to reduce risk. If those controls are inadequate, risk exists, and thus loss may occur.

7.1.3.11.2 Responsibility for Test Type

Senior management and end user/customer management bear primary responsibility for the adequacy of controls. Some organizations have delegated the responsibility to assess control to the internal audit function. Others leave it to the testers and project personnel to verify the adequacy of the system of internal control.

7.1.3.11.3 Test Condition Creation Approach

Testing the adequacy of controls is complex. For high-risk financial applications, testers are encouraged to involve their internal auditors.

A four-step procedure is needed to develop a good set of control transaction tests, as follows:

1. Document transaction flow and controls over transaction
2. Test each control
3. Evaluate the effectiveness of the controls
4. Perform assessment of correctness of level of stated potential risk

7.1.3.11.4 Examples

Separation of duties, authorization of processing, batching controls, record counts, key verification, control totals, data input validation, passwords, and reconciliation.

7.1.3.12 Test Transaction Type: ATTRIBUTE

Attributes are the quality and productivity characteristics of the software being tested. They are independent of the functional aspects, and primarily relate to the architectural structural aspects of a system. Attribute testing is complex and often avoided because it requires innovative testing techniques. However, much of the dissatisfaction expressed about software by end users/customers relates to attributes rather than functions. In Skill Category 1, section 1.2.5, Software Quality Factors and Software Quality Criteria were explained. Attribute testing relates to that discussion. Table 7-4 lists those Attributes with a few additions.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORRECTNESS</td>
<td>Assurance that the data entered, processed, and output by the application system is accurate and complete. Accuracy and completeness are achieved through controls over transactions and data elements. The control should commence when a transaction is originated and conclude when the transaction data has been used for its intended purpose.</td>
</tr>
<tr>
<td>FILE INTEGRITY</td>
<td>Assurance that the data entered into the application system will be returned unaltered. The file integrity procedures ensure that the right file is used and that the data on the file and the sequence in which the data is stored and retrieved is correct.</td>
</tr>
<tr>
<td>AUTHORIZATION</td>
<td>Assurance that data is processed in accordance with the intents of management. In an application system, there is both general and specific authorization for the processing of transactions. General authorization governs the authority to conduct different types of business while specific authorization provides the authority to perform a specific act.</td>
</tr>
<tr>
<td>AUDIT TRAIL</td>
<td>The capability to substantiate the processing that has occurred. The processing of data can be supported through the retention of sufficient evidential matter to substantiate the accuracy, completeness, timeliness, and authorization of data. The process of saving the supporting evidential matter is frequently called an audit trail.</td>
</tr>
<tr>
<td>CONTINUITY OF PROCESSING</td>
<td>The ability to sustain processing in the event problems occur. Continuity of processing assures that the necessary procedures and backup information are available to recoup and recover operations should the integrity of operations be lost due to problems. Continuity of processing includes the timeliness of recovery operations and the ability to maintain processing periods when the computer is inoperable.</td>
</tr>
<tr>
<td>SERVICE LEVELS</td>
<td>Assurance that the desired results will be available within a time frame acceptable to the user. To achieve the desired service level, it is necessary to match user requirements with available resources. Resources include input/output capabilities, communication facilities, processing, and systems software capabilities.</td>
</tr>
</tbody>
</table>
Table 7-4 Software Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS CONTROL</td>
<td>Assurance that the application system resources will be protected against accidental or intentional modification, destruction, misuse, and disclosure. The security procedure is the totality of the steps taken to ensure the integrity of application data and programs from unintentional or unauthorized acts.</td>
</tr>
<tr>
<td>COMPLIANCE</td>
<td>Assurance that the system is designed in accordance with organizational methodology, policies, procedures, and standards. These requirements need to be identified, implemented, and maintained in conjunction with other application requirements.</td>
</tr>
<tr>
<td>RELIABILITY</td>
<td>Assurance that the application will perform its intended function with the required precision over an extended period of time when placed into production.</td>
</tr>
<tr>
<td>EASE OF USE</td>
<td>The extent of effort required to learn, operate, prepare input for and interpret output from the system. This test factor deals with the usability of the system to the people interfacing with the application system.</td>
</tr>
<tr>
<td>MAINTAINABLE</td>
<td>The effort required to locate and fix an error in an operational system.</td>
</tr>
<tr>
<td>PORTABLE</td>
<td>The effort required to transfer a program from one hardware configuration and/or software system environment to another. The effort includes data conversion, program changes, operating system changes, and documentation changes.</td>
</tr>
<tr>
<td>COUPLING</td>
<td>The effort required to interface one application system with all the application systems in the processing environment which either it receives data from or transmits data to.</td>
</tr>
<tr>
<td>PERFORMANCE</td>
<td>The amount of computing resources and code required by a system to perform its stated functions. Performance includes both the manual and automated segments involved in fulfilling system functions.</td>
</tr>
<tr>
<td>EASE OF OPERATION</td>
<td>The amount of effort required to integrate the system into the operating environment and then to operate the application system. The procedures can be both manual and automated.</td>
</tr>
</tbody>
</table>
the discretion of the project people because implementing functionality takes a higher priority than implementing attributes in most organizations, the attributes are not adequately addressed.

7.1.3.12.2 Responsibility for Test Type

The responsibility for the attributes resides primarily with the development and test team. These should be specified in the standards for building/acquiring software systems. A secondary responsibility resides with the end user/customer to request these attributes.

7.1.3.12.3 Test Condition Creation Approach

There are two aspects of testing the attributes. The first is to define the attributes, and the second is to prioritize the attributes. The first is necessary because defining the attribute determines the levels of quality that the development team intends to implement in the application system. For example, if maintainability is one of the quality attributes, then the IT standards need to state the structural characteristics that will be incorporated into software to achieve maintainability. The second is important because emphasizing one of the quality factors may in fact de-emphasize another. For example, it is difficult to have an operation that is very easy to use and yet highly secure. Ease of use makes access easy, while highly secure makes access more difficult.

Developing test conditions for the attribute test characteristics involves the following four steps:

1. Identify attributes
2. Prioritize the attributes
3. Develop test conditions to test important attributes
4. Assess the completeness of the attribute testing

7.1.3.13 Test Transaction Type: STATES

The states are conditions relating to the operating environment and the functional environment. These are special conditions which may or may not occur, and need to be addressed by the testers.

Note: This is not to be confused with State Transition testing, a black-box testing technique described in section 7.4.2.6.

7.1.3.13.1 Testing Concern

The test concern is that these special states will cause operational problems. If the testers validate that the software can handle these states there is less concern that the system will abnormally terminate or function improperly during operation.
7.1.3.13.2 Responsibility for Test Type
The responsibility for validating the states resides with the development and test teams.

7.1.3.13.3 Test Condition Creation Approach
There are three steps involved in testing states as follows:
   1. Identify states to be tested
   2. Develop test conditions for each state
   3. Assess the completeness of the state’s testing

7.1.3.13.4 Examples
Duplicate master records, empty table, missing record, concurrent reads/updates.

7.1.3.14 Test Transaction Type: PROCEDURE
Procedures are the operating procedures used in operating the software.

7.1.3.14.1 Testing Concern
The primary concern is that the application system will not be operational because the operating procedures will not work.

7.1.3.14.2 Responsibility for Test Type
The primary responsibility for testing the operating procedures resides with the datacenter or other IT group responsible for operations. However, testers may work with them in fulfilling this responsibility.

7.1.3.14.3 Test Condition Creation Approach
A three-step process is needed to test procedures as follows:
   1. Identify procedures to be tested
   2. Develop test conditions for each procedure
   3. Assess the adequacy of procedure testing

7.1.3.14.4 Examples
Start-up, recovery, and server operations.
7.1.4 Defining Test Conditions from Business Case Analysis

The fourth approach to identifying test conditions is to review real world business cases. This approach would look for the entities that are processed in the user/customer’s organization and break them down to their lowest levels. These can be things or people. This broad category of identifying test conditions from business cases would also include Use Cases and User Stories as they apply to the development of test cases. Use Cases and User Stories will be discussed in section 7.2 and 7.3 later in this skill category.

Examples of Business Case Analysis are:

- An insurance company will process policies and insure people.
- A bank will have accounts and account holders.
- A hospital will have patients.
- A manufacturing company will build and track products.

7.1.4.1 Steps to Create Conditions

Shown below are the six steps necessary to create a test case using Business Case Analysis:

1. Identify the major entities processed.
2. Identify the sub-types within each entity.
   Example: An insurance policy may be the major entity, but sub-types can include life policies, health policies, group policies, auto policies, and homeowner’s policies.
3. Continue to break the sub-types down until you have reached the lowest level.
   Example: A life insurance policy may be active, lapsed, whole life, term, etc.
4. Combine sub-types to form test cases.
   Example: A policyholder with a term life policy lets the policy lapse for lack of premium payment.
5. Add the details to support the test case.
   Example: A policyholder, Sanjay Gupta, who lives at 123 Main St. in Toronto, Ontario does not pay his premium for 60 days.
6. Describe the expected results.
   Example: After 30 days of non-payment, Sanjay Gupta is sent an Intent to Cancel notice. After 60 days of non-payment, Sanjay Gupta is sent a policy lapse notice.

Section 7.4.2.7 describes Scenario Testing which aligns with Business Case Analysis.
7.1.5 Defining Test Conditions from Structural Analysis

Structural analysis is a technique used by developers to define unit test cases. Structural analysis usually involves path and condition coverage. The downside of structural cases is that they only find defects in the code logic, not necessarily defects in the software's fitness for use in the user environment. The benefit of structural analysis is that it can find defects not obvious from the black box or external perspective.

It is possible to perform structural analysis without tools on sections of code that are not highly complex. To perform structural analysis on entire software modules or complex sections of code requires automation.

Section 7.4.1 provides considerably detailed information regarding techniques used in structural analysis.

7.2 Test Conditions from Use Cases

In section 7.1, a list of five different approaches to identifying testable conditions was detailed. The objective of those five approaches was to look at the application under test from as many perspectives as possible. By analyzing the application from different angles, the risk that incomplete, incorrect, and missing test cases causing incomplete and erroneous test results would be minimized. Flawed test results cause rework at a minimum, and at worst, a flawed system is developed. There is a need to ensure that all required test conditions (and by extension test cases) are identified so that all system functionality requirements are tested.

Another approach to identifying testable conditions is to utilize, if available, the Use Cases defined for the application under test. Use Cases are typically written by the business analyst during the requirements elicitation process. This section of Skill Category 7 will first describe what Use Cases are and then relate their use to the testing process.

7.2.1 What is a Use Case

A Use Case is a technique for capturing the functional requirements of systems through the interaction between an Actor and the System. The Actor is an individual, group or entity outside the system. In Use Cases, the Actor may include other software systems, hardware components or other entities. Extending the use of actors beyond the perspective of an individual allows the Use Cases to add depth and understanding to applications beyond simply a customer perspective.
Actors can be divided into two groups: a primary actor is one having a goal which requires the assistance of the system. A secondary actor is one from which the system requires assistance.

### 7.2.1.1 System Boundary Diagram

A system boundary diagram depicts the interfaces between the software under test and the individuals, systems, and other interfaces. These interfaces or external agents are referred to as “actors.” The purpose of the system boundary diagram is to establish the scope of the system and to identify the actors (i.e., the interfaces) that need to be developed.

An example of a system boundary diagram for a Kiosk Event Ticketing program is illustrated in Figure 7-1.

![System Boundary Diagram](image)

**Figure 7-1 System Boundary Diagram**

For the application, each system boundary needs to be defined. System boundaries can include:

- Individuals/groups that manually interface with the software
- Other systems that interface with the software
- Libraries
- Objects within object-oriented systems

Each system boundary should be described. For each boundary an actor must be identified.

Two aspects of actor definition are required. The first is the actor description, and the second, when possible, is the name of an individual or group who can play the role of the actor (i.e., represent that boundary interface). For example, in Figure 7-1 the credit card processing system is identified as an interface. The actor is the Internet Credit Card Processing Gateway. Identifying a resource by name can be very helpful to the development team.
The Use Case looks at what the Actor is attempting to accomplish through the system. Use Cases provide a way to represent the user requirements and must align with the system’s business requirements. Because of the broader definition of the Actor, it is possible to include other parts of the processing stream in Use Case development.

### 7.2.1.2 Use Case Flow

Use Cases describe all the tasks that must be performed for the Actor to achieve the desired objective and include all of the desired functionality. Using an example of an individual purchasing an event ticket from a kiosk, one Use Case will follow a single flow uninterrupted by errors or exceptions from beginning to end. This is typically referred to as the Happy Path.

Figure 7-2 uses the Kiosk event to explain the simple flow within the Use Cases.

![Geting tickets](image)

*Figure 7-2  Use Case Flow*

Figure 7-3 expands the simple case into more detailed functionality. All of the identified options are listed. The actions in italics represent the flow of a single Use Case (for example Shop by Date; Select a Valid Option (Date); Ask Customer to Enter Date…).
Figure 7-3  Sample Use Case

And so on through the completion of the transaction.
Where there is more than one valid path through the system, each valid path is often termed a scenario.

Other results may occur; as none of them are intended results, they represent error conditions. If there are alternative paths that lead to a successful conclusion of the interaction (the actor achieves the desired goal) through effective error handling, these may be added to the list of scenarios as recovery scenarios. Unsuccessful conclusions that result in the actor abandoning the goal are referred to as failure scenarios.

### 7.2.1.3 Use Case Activity Diagram

An Activity Diagram is a direct offshoot of the System Boundary Diagram. In the case of Activity Diagrams, mapping the Use Case onto an Activity Diagram can provide a good means of visualizing the overlay of system behavior onto business process. An Activity Diagram representing the Use Case of logging into a system or registering in that system is shown in Figure 7-4.

![Use Case Activity Diagram](image)

**Figure 7-4 Use Case Activity Diagram – Create User**

### 7.2.2 How Use Cases are Created

Use Cases are typically created as a part of the Requirements Definition process. Use Cases can be developed as a part of a JAD process, or as a part of any sound development methodology.

Each Use Case is uniquely identified; Karl Wiegers, author of *Software Requirements*[^1], recommends usage of the Verb-Noun syntax for clarity. The Use Case above would be

[^1]: Wiegers, Karl; Software Requirements; Microsoft Press, 1999.
Purchase Tickets. An alternative flow (and Use Case) that addresses use of the Cancel Option at any point might be captioned Cancel Transaction.

While listing the various events, the System Boundary Diagrams can be developed to provide a graphic representation of the possible entities (Figure 7-1) in the Use Case. In addition to the main flow of a process, Use Cases models, Figure 7-5, can reflect the existence of alternative flows.

![Use Case Model](image)

**Figure 7-5 Use Case Model**

These alternative flows are related to the Use Case by the following three conventions:

<<**extend**>> extends the normal course, inserts another Use Case that defines an alternative path. For example, a path might exist which allows the customer to simply see what is available without making a purchase. This could be referred to as Check Availability.

<<**include**>> is a Use Case that defines common functionality shared by other Use Cases. Process Credit Card Payment might be included as a common function if it is used elsewhere.

**Exceptions** are conditions that result in the task not being successfully completed. In the case above, Option Not Available could result in no ticket purchase. In some cases these may be developed as a special type of alternative path.

The initial development of the Use Case may be very simple and lacking in detail. One of the advantages of the Use Case is that it can evolve and develop over the life of the project. Because they can grow and change Use Cases for large projects may be classified as follows:

- **Essential Use Case** - is described in technology free terminology and describes the business process in the language of the Actor; it includes the goal or object information. This initial business case will describe a process that has value to the Actor and describes what the process does.
• **System Use Case** - is at a lower level of detail and describes what the system does; it will specify the input data and the expected data results. The system Use Case will describe how the Actor and the system interact, not just what the objective is.

### 7.2.3 Use Case Format

Just as there are many workable approaches to Use Case development, so too are there a wide range of recommended formats. The following list represents those found most commonly, with comments regarding specific application or justification. This information should be captured in an organization standard format.

**Case ID** - A unique identifier for each Use Case, it includes cross references to the requirement(s) being tested so that it is possible to trace each requirement through testing.

For example:

1. Customer enters a valid date and time combination
   1.a. Submitted data is invalid
   1.a.1. Kiosk requests valid date/time data
   1.b. Submitted data is incomplete
   1.b.1. Kiosk requests completed data

**Use Case Name** - A unique short name for the Use Case that implicitly expresses the user’s intent or purpose\(^1\). The sample event ticket case above might be captioned *ChooseEventTime*. Using this nomenclature ties the individual Use Case directly to the Use Cases originally described and allows it to be sequenced on a name basis alone.

**Summary Description** - A several sentence description summarizing the Use Cases. This might appear redundant when an effective Use Case naming standard is in place, but with large systems, it is possible to become confused about specific points of functionality.

**Frequency / Iteration Number** - These two pieces of information provide additional context for the case. The first, frequency, deals with how often the actor executes or triggers the function covered by the Use Case. This helps to determine how important this functionality is to the overall system. Iteration number addresses how many times this set of Use Cases has been executed. There should be a correlation between the two numbers.

**Status** - This is the status of the case itself: In Development, Ready for Review, and Passed or Failed Review are typical status designation.

---

1. Ambler, Scott; Web services programming tips and tricks: Documenting a Use Case; (scott.ambler@ronin-intl.com); October 2000.
Actors - The list of actors associated with the case; while the primary actor is often clear from the summary description, the role of secondary actors is easy to miss. This may cause problems in identifying all of the potential alternative paths.

Trigger - This is the starting point for any action in a process or sub-process. The first trigger is always the result of interaction with the primary actor. Subsequent triggers initiate other processes and sub-processes needed by the system to achieve the actor’s goal and to fulfill its responsibilities.

Basic Course of Events - This is called the main path, the happy path or the primary path. It is the main flow of logic an actor follows to achieve the desired goal. It describes how the system works when everything functions properly. If the System Boundary Diagram contains an <<includes>> or <<extends>>, it can be described here. Alternatively any additional categories for <<extends>> and <<includes>> must be created. If there are relatively few, they should be broken out so they will not be overlooked. If they are common, either practice will work.

Alternative Events - Less frequently used paths of logic, these may be the result of alternative work processes or an error condition. Alternative events are often signaled by the existence of an <<exception>> in the System Boundary Diagram.

Pre-Conditions - A list of conditions, if any, which must be met before the Use Case can be properly executed. In the Kiosk examples cited previously, before a payment can be calculated, an event, and the number and location of seats must be selected. During Unit and System testing this situation is handled using Stubs. By acceptance testing, there should be no Stubs left in the system.

Business Rules and Assumptions - Any business rules not clearly expressed in either the main or alternate paths must be stated. These may include disqualifying responses to pre-conditions. Assumptions about the domain that are not made explicit in the main and alternate paths must be recorded. All assumptions should have been verified prior to the product arriving for acceptance testing.

Post Conditions - A list of conditions, if any, which will be true after the Use Case finished successfully. In the Kiosk example the Post Conditions might include:

- The customer receives the correct number of tickets
- Each ticket displays the correct event name and price
- Each ticket shows the requested data, time, and seat location
- The total price for the ticket(s) was properly calculated
- The customer account is properly debited for the transaction
- The ticket inventory is properly updated to reflect tickets issued
- The accounts receivable system receives the correct payment information

Notes - Any relevant information not previously recorded should be entered here. If certain types of information appear consistently, create a category for them.
**Author, Action and Date** - This is a sequential list of all of the authors and the date(s) of their work on the Use Case. Many Use Cases are developed and reworked multiple times over the course of a large project. This information will help research any problems with the case that might arise.

### 7.2.4 How Use Cases are Applied

Because of their flexibility and the vision they provide into the functionality needed by the customer, Use Cases are an excellent requirements definition tool. They take the information derived from a business event and add more detail and greater understanding of what will be involved.

Using the Kiosk example above, it becomes clear this process will require access to many kinds of information from multiple sources. Although no design decisions are ready to be made about how to access that data, the requirement to do so is obvious. A quick survey of entertainment purveyors (the source of the tickets) may reveal that while hockey, theatre, and symphony tickets are readily accessible, football tickets are not. This may lead to a change in scope to exclude football tickets or in an upward revision of the time and cost estimates for achieving that functionality.

Likewise, the Use Case provides an excellent entrée into the testing effort, to such an extent that for many organizations, the benefits of Use Cases for requirements are ignored in the effort to jump start testing! Further to the relationship of Use Cases to testing, the iteration process may require a little explanation. As the Use Case evolves from a purely business event focus to include more system information, it may be desirable to maintain several versions or levels of the Use Case. For example the initial Use Case, developed during the first JAD session(s), might be Iteration 1; as it is expanded to include systems information, it becomes Iteration 2 and when fully configured to include the remaining testing related information it is Iteration 3. Use of common Iteration levels across projects will reduce confusion and aid applicability of the Use Case.

### 7.2.5 Develop Test Cases from Use Cases

**Terminology**

<table>
<thead>
<tr>
<th>Terminology</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sad Path</td>
<td>An unsuccessful execution of a test case, the Sad Path(s).</td>
</tr>
<tr>
<td>Alternate Path</td>
<td>An unsuccessful execution of a test case, the Sad Path(s).</td>
</tr>
</tbody>
</table>

Well-developed Use Cases are a powerful tool for the software tester. By their very definition, the Use Case documents provide a comprehensive list of testable conditions. There should be a one-to-one relationship between Use Case definitions and test conditions. There will then be at least two test cases for each Use Case: one for successful execution of the Use Case, the Happy Path, and one for an unsuccessful execution of a test case, the Sad Path(s). However, there may be numerous test cases for each Use Case.
Additional testable conditions are derived from the exceptions and alternative course of the Use Case (alternate path(s)). Note that additional detail may need to be added to support the actual testing of all the possible scenarios of the Use Case.

### 7.2.5.1 Steps to Test from Use Cases

During the development of Use Cases, the components of pre-conditions, the process flow, the business rules, and the post-conditions are documented. Each of these components will be a focus for evaluation during the test.

The most important part of developing test cases from Use Cases is to understand the flow of events. The flows of events are the happy path, the sad path, and the alternate event paths. The happy path is what normally happens when the Use Case is performed. The sad path represents a correct path but one which does not produce results, which is what the application should do on that path. The alternate paths represent detours off the happy path but which can still yield the results of the happy path. Understanding the flow is a critical first step. The various diagrams (e.g., Use Case activity map) generated during Use Case development are invaluable in this process.

The next step in the process is to take each event flow as identified in the previous step and create Use Case scenarios. A Use Case scenario is a complete path through the Use Case for a particular flow. The happy path would be one of the Use Case scenarios.

Once all the Use Case scenarios are written, at least one test case, and in most cases more than one test case, will be developed for each scenario. The test case should ensure that:

1. The test will not initiate if any of the pre-conditions are wrong.
2. All business rules along the particular path are validated.
3. The Use Case for the particular test procedures the conditions and/or output as specified in the post-conditions.

The detailed development of test cases from Use Cases will utilize test design techniques described in section 7.4.

### 7.3 Test Conditions from User Stories

To begin with, User Stories and Use Cases are not the same. Some experts suggest they are not even remotely the same while other experts consider them closely related. The purpose here is not to debate but rather to define User Stories and further the discussion of how User Stories can be used to identify testable conditions within iterative development frameworks like Agile. In some ways, the results of the User Story and Use Case processes are similar,
but it is the journey taken to arrive at those results that differs. Table 7-5 compares the characteristics of a User Story and Use Case.

<table>
<thead>
<tr>
<th><strong>User Story</strong></th>
<th><strong>Use Case</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Very simple</td>
<td>More complex</td>
</tr>
<tr>
<td>Written by the Customer or</td>
<td>Written by the BA or Developer</td>
</tr>
<tr>
<td>Product Owner</td>
<td></td>
</tr>
<tr>
<td>Incomplete</td>
<td>Attempts to be complete</td>
</tr>
<tr>
<td>Placeholder for a conversation</td>
<td>Intends to answer any question</td>
</tr>
<tr>
<td>Written quickly</td>
<td>May take some time</td>
</tr>
<tr>
<td>Easy to understand</td>
<td>May take training to understand</td>
</tr>
</tbody>
</table>

Table 7-5 User Story vs Use Case

As defined in section 7.2.1, a Use Case describes the interaction between an Actor (which can be person or another system) and the System. The development of Use Cases is a well-defined process resulting in specific documents. While Use Cases can be iterative in nature the primary goal is to document the system early in the life cycle.

Unlike a Use Case, a User Story is a short description of something that a customer will do when they use an application (software system). The User Story is focused on the value or result a customer would receive from doing whatever it is the application does. User Stories are written from the point of view of a person using the application. Mike Cohn, a respected Agile expert and contributor to the invention of the Scrum software development methodology suggests the User Story format as: "As an [actor] I want [action] so that [achievement]." The User Story starts with that simple description and the details of the User Story emerge organically as part of the iterative process.

### 7.3.1 INVEST in User Stories

By its very definition, a quality User Story presents itself as a testable condition. Bill Wake coined the mnemonic device INVEST as a model for quality User Stories. INVEST is:

**Independent** – A User Story is independent of other User Stories and stories do not overlap.

**Negotiable** – A User Story can be changed up until it enters the iteration.

**Valuable** – A User Story must deliver value to the end user.

---

2. William Wake, Senior Consultant, Industrial Logic, Inc.
**Estimable** – User Stories must be created such that their size can be estimated.

**Small** – User Stories should not be so big as to become impossible to plan/task/prioritize with a certain level of certainty. A rule of thumb, a User Story should require no more than four days and no less than a half day to implement.

**Testable** – A User Story must provide the necessary information to make test development possible.

### 7.3.2 Acceptance Criteria and User Stories

Aaron Bjork of Microsoft describes acceptance criteria as “the handshake between the product owner and the team regarding the definition of ‘done.’” The acceptance criteria are those things a user will be able to do with the product after a story is implemented. When a story is implemented correctly the acceptance criteria is satisfied. Within the context of identifying the testable conditions, the acceptance criteria provide specifically that, the condition to be tested to ensure the application performs as desired. The acceptance criteria represent the “conditions of satisfaction.”

### 7.3.3 Acceptance Criteria, Acceptance Tests, and User Stories

User Stories tell what is the desired result or value of the product from the user’s perspective, the acceptance criteria define what the product must do to satisfy the desire, and finally the acceptance tests are the specific steps to check a feature to ensure it behaves as required to satisfy the acceptance criteria. Each User Story may have several acceptance tests. Each acceptance test will likely have several test cases.

### 7.3.4 User Stories Provide a Perspective

Earlier in this skill category the importance of looking at an application from a variety of perspectives was described. User stories are not test cases; test conditions are not test cases. Test conditions are identified for the purpose of developing test cases. Whether a development team is using an Agile methodology or some other development methodology, the need to initially identify high level testable conditions before drilling down to a more granular level is universal.

### 7.3.5 Create Test Cases from User Stories

As stated earlier, the User Story is a short description of something that a customer will do when they use an application (software system) and that the acceptance criteria is defined as those things a user will be able to do with the product after a story is implemented.
Recognizing that User Stories are a high-level description and focused on the value or result the user will receive, it is uncommon to test the User Story directly.

In the Agile development model, unit testing is at the heart of the testing process. Developers will typically break down the User Story into distinct program modules and the unit testing process tests those modules at the code level. By contrast, the tester looks at the system more holistically with a goal to test the system like the user will use the system. Questions like, what would the user do, how might the user misuse the system intentionally or unintentionally, what might derail the process from attaining the objective of the User Story? In essence, testing takes on a much more exploratory testing flavor (see section 1.9.2.4).

As a precursor to discussing the Agile testing process it is important to remember that testing on an Agile project is not a phase but rather a continuous process throughout the life cycle. The success of an Agile developed product mandates this commitment to continuous testing.

### 7.3.5.1 Steps to Test from User Stories

During the process of writing the User Story the product owner also writes acceptance criteria, which defines the boundaries of a User Story, and will be used during development to validate that a story is completed and working as intended. The acceptance criteria will include the functional and non-functional criteria. These criteria would identify specific user tasks, functions or business processes that must be in place at the end of the sprint or project. A functional requirement might be “On acceptance of bank debit card the PIN input field will be displayed.” A non-functional requirement might be “Option LEDs will blink when option is available.” Performance will typically be a criterion and likely measured as a response time. Expected performance should be spelled out as a range such as “1-3 seconds for PIN acceptance or rejection.”

Each User Story will have one or more acceptance tests. Similar to the Use Case test process as described in section 7.2.5.1, the acceptance tests should be scenario based that can then be broken down into one or more test cases. Acceptance tests are by default part of the happy path. Tests that validate the correct function of features required by the acceptance test are also considered happy path tests. Tests for sad and alternate paths are best executed after the happy path tests have passed.

The detailed development of test cases used to validate the acceptance criteria will utilize test design techniques described in section 7.4.

### 7.4 Test Design Techniques

In Skill Category 5, Test Planning, the need to identify test objectives as part of the planning process was explained. In sections 7.1 to 7.3, the procedures to identify testable conditions was described from both a high level point of view as exemplified by Use Cases and User Stories and a much more granular approach was discussed in Test Transaction Types (see
Regardless of the development methodology, the need to understand first the objectives of the tests and then the conditions to be tested is irrefutable.

With the objectives and conditions understood, the next step would be to understand the types of test design techniques that can be employed in the development of test cases. Test design techniques can be grouped into four major categories:

- Structural Testing
- Functional Testing
- Experience-Based Techniques
- Non-Functional Testing

### 7.4.1 Structural Test Design Techniques

In Skill Category 1, section 1.10.1, two categories of Structural testing were defined, Structural System Testing and White-box Testing. Structural System Testing and White-box Testing were grouped together because the objectives of both types of testing are not to validate functionality, what the system is supposed to do, but rather to evaluate if the system does what the system is supposed to do well (either at the application level or code level). Section 1.10.1.1 elaborated on Structural System Testing and described the following test types:

- Stress Testing
- Execution Testing
- Recovery Testing
- Compliance Testing

For the purpose of describing structural test case design techniques in this skill category, the focus will be on white-box testing techniques.

White-box testing (also referred to as clear-box testing, glass-box testing, and structure-based testing) includes the following types of test techniques:

- Statement Testing
- Branch Testing
- Decision Testing
- Branch Condition Testing
- Branch Condition Combination Testing
- Modified Condition Decision Coverage Testing
- Date Flow Testing

Collectively, statement, branch, decision, branch condition, branch condition combination and modified condition decision testing are known as Control Flow Testing. Control Flow Testing tends to focus on ensuring that each statement within a program is executed at least once. Data Flow Testing by contrast focuses on how statements interact through the data flow.
7.4.1.1 Statement Testing

Statement testing requires that every statement in the program be executed. While it is obvious that achieving 100 percent statement coverage does not ensure a correct program, it is equally obvious that anything less means that there is code in the program that has never been executed.

Shown here is an example of code that sets a variable (WithdrawalMax), reads two numbers into variables (AcctBal and WithdrawalAmt). If there are sufficient funds in the account to cover the withdrawal amount the program prints out a message indicating that and if not sufficient funds a message is printed and the program terminates. If sufficient funds exist then the program checks to see if the withdrawal amount exceeds the withdrawal maximum. If so, it prints a message that withdrawal exceeds limit and terminates, else program terminates.

This code is in the flow chart shown as Figure 7-6. Every box in the flowchart represents an individual statement.

```
WithdrawalMax = 100
INPUT AcctBal
INPUT WithdrawalAmt
IF AcctBal >= WithdrawalAmt THEN
    PRINT "Sufficient Funds for Withdrawal"
    IF WithdrawalAmt > WithdrawalMax THEN
        PRINT "Withdrawal Amount Exceeds Withdrawal Limit"
    END IF
    ELSE
        PRINT "Withdrawal Amount Exceeds Account Balance"
    END IF
ELSE
    PRINT "Withdrawal Amount Exceeds Withdrawal Limit"
ENDIF
```

Figure 7-6 Flow Chart
For statement testing, a series of test cases will be written to execute each input, decision, and output in the flowchart.

In this example, two tests are necessary.

1. Set the WithdrawalMax variable, read data, evaluate availability of funds as not sufficient, and print message.

2. Set the WithdrawalMax variable, read data, evaluate availability of funds as sufficient, print message, evaluate the limit is exceeded, and print message.

By executing both tests, all statements have been executed achieving 100% statement coverage. However, in the following section the drawback of just statement testing is described.

### 7.4.1.2 Branch/Decision Testing

Achieving 100 percent statement coverage does not ensure that each branch in the program flow graph has been executed. For example, executing an “if…then” statement, (no “else”) when the tested condition is true, tests only one of two branches in the flow chart. Branch testing seeks to ensure that every branch from every decision has been executed.

Referring to the source code and flow chart from section 7.4.1.1, it is necessary to find the minimum number of paths which will ensure that all true/false decisions are covered. The paths can be identified as follows using the numbers in Figure 7-6.

1. 1A-2B-3C-4D-5E
2. 1A-2B-3C-4F-6G-7J
3. 1A-2B-3C-4F-6G-7H-8I

For this example, the number of tests required to ensure decision or branch coverage is 3.

### 7.4.1.3 Condition Testing

Branch Condition Testing, Branch Condition Combination Testing, and Modified Condition Decision Coverage Testing are closely related and for the purposes of this discussion will collectively referred to as Condition Testing. Examples of each will be given in this section. In conditional testing, each clause in every condition is forced to take on each of its possible values in combination with those of other clauses. Conditional testing thus subsumes branch testing; and therefore, inherits the same problems as branch testing.
Condition testing can be accomplished by breaking compound condition statements into simple conditions and nesting the resulting “if” statements.

For this example the source code in 7.4.1.1 has been modified to remove the nested “if” by creating a compound condition. In this code a CreditLine variable has been added that when a credit line is available (by answering Y) then funds withdrawal is always approved. The resultant code is shown below:

```
WithdrawalMax = 100
INPUT AcctBal
INPUT WithdrawalAmt
INPUT CreditLine
IF CreditLine="Y" or (AcctBal >= WithdrawalAmt
AND WithdrawalAmt <= WithdrawalMax) THEN
    PRINT "Withdrawal of Funds Approved"
ELSE
    PRINT "Withdrawal of Funds Denied"
END IF
```

Figure 7-7 Flow Chart

In this example we will identify CreditLine="Y", AcctBal>WithdrawalAmt, and WithdrawalAmt <=WithdrawalMax as Boolean operands Exp1, Exp2, and Exp3 respectively.

Branch Condition Coverage would require Boolean operand Exp1 to be evaluated both TRUE and FALSE, Boolean operand Exp2 to be evaluated both TRUE and FALSE, and Boolean operand Exp3 to be evaluated both TRUE and FALSE.

Branch Condition Coverage may therefore be achieved with the following set of test inputs (note that there are alternative sets of test inputs which will also achieve Branch Condition Coverage).

<table>
<thead>
<tr>
<th>Case</th>
<th>Exp1</th>
<th>Exp2</th>
<th>Exp3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>2</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>
Table 7-6 Branch Condition Coverage

While this would exercise each of the Boolean operands, it would not test all possible combinations of TRUE / FALSE. Branch Condition Combination Coverage would require all combinations of Boolean operands Exp1, Exp2 and Exp3 to be evaluated. Table 7-7 shows the Branch Condition Combination Testing table for this example.

<table>
<thead>
<tr>
<th>Case</th>
<th>Exp1</th>
<th>Exp2</th>
<th>Exp3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>2</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>3</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>4</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>5</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>6</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>7</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>8</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

Table 7-7 Branch Condition Combination Testing

Branch Condition Combination Coverage is very thorough, requiring 2^n test cases to achieve 100% coverage of a condition containing n Boolean operands. This rapidly becomes unachievable for more complex conditions.

Modified Condition Decision Coverage is a compromise which requires fewer test cases than Branch Condition Combination Coverage. Modified Condition Decision Coverage requires test cases to show that each Boolean operand (Exp1, Exp2, and Exp3) can independently affect the outcome of the decision. This is less than all the combinations (as required by Branch Condition Combination Coverage). This reduction in the number of cases is often referred to as collapsing a decision table and will be describe in further detail in the decision table section of black-box testing.

7.4.1.4 Path Testing

Path Testing is a systematic method of white box testing, where the aim is to identify the execution paths through each module of program code and then create test cases to cover those paths. In practice, such coverage is impossible to achieve for a variety of reasons. For example, any program with an indefinite loop contains an infinite number of paths, one for each iteration of the loop. Thus, no finite set of tests will execute all paths. Also, it is undecided whether an arbitrary program will halt for an arbitrary input. It is therefore impossible to decide whether a path is finite for a given input.

In response to these difficulties, several simplifying approaches have been proposed. Infinitely many paths can be partitioned into a finite set of equivalence classes based on
characteristics of the loops. Boundary and interior testing require executing loops zero times, one time, and if possible, the maximum number of times. Linear sequence code and jump criteria specify a hierarchy of successively more complex path coverage.

7.4.1.5 Data Flow Analysis (Testing)

In data flow analysis, we are interested in tracing the behavior of program variables as they are initialized and modified while the program executes. This behavior can be classified by when a particular variable is referenced, defined, or undefined in the program. A variable is referenced when its value must be obtained from memory during the evaluation of an expression in a statement. For example, a variable is referenced when it appears on the right-hand side of an assignment statement, or when it appears as an array index anywhere in a statement. A variable is defined if a new value for that variable results from the execution of a statement, such as when a variable appears on the left-hand side of an assignment. A variable is unreferenced when its value is no longer determinable from the program flow. Examples of unreferenced variables are local variables in a subroutine after exit and DO indices on loop exit.

Examples of information gleaned from data flow testing include:

- Defining a variable twice with no intervening reference
- Defining a variable multiple times before it is used
- Referencing a variable that is undefined
- Identifying a variable that is declared but never used within the program
- De-allocating a variable before it is used

Data flow analysis, when used in test case generation, exploits the relationship between points where variables are defined and points where they are used.

7.4.1.5.1 Data Flow Analysis Example

A very simplistic example of the data flow concept is shown in the source code below.

```
1. AcctBal......
   IF......
      AcctBal = ..... 3
   END IF
   ...... Lines of code ..... 
   ...... Lines of code ..... 
7. TotalAcctsBalance = AcctBal +
```

- Note that the value of AcctBal can be computed at 1 or 3.
- Bad computation at line 1 or line 3 could be revealed only if AcctBal is used at 7.
- Point (1,7) and (3,7) are referred to as def-use (DU) pairs.
  - defs at 1,3
7.4.2 Functional Test Design Techniques

In Skill Category 1, section 1.10.2, two categories of Functional Testing were defined, Functional System Testing and Black-box Testing. While both types test to ensure that the application under test does what it is supposed to do, functional system testing tests look at the application holistically while black-box testing techniques are used to develop specific test cases testing unique functionality.

Functional System Testing ensures that the system requirements and specifications are achieved by creating test conditions for use in evaluating the correctness of the application. Section 1.10.2.1 elaborated on Functional System Testing by describing the following test types:

- Requirements
- Error Handling
- Intersystem
- Control
- Parallel

For the purpose of describing functional test case design techniques in this section of Skill Category 7, the focus will be on black-box testing techniques.

Black-box testing includes the following types of test techniques:

- Equivalence Partitioning
- Boundary Value Analysis
- Decision Table Testing
- Pair-Wise Testing
- Cause-Effect Graphing
- State Transition Testing
- Scenario Testing
- Use Case Testing (see section 7.2)
- User Story Testing (see section 7.3)

7.4.2.1 Equivalence Partitioning (Classes)

Specifications frequently partition the set of all possible inputs into classes that receive equivalent treatment. Such partitioning is called equivalence partitioning. A result of equivalence partitioning is the identification of a finite set of functions and their associated input
and output results. The benefit of this technique is that you do not need to generate redundant test cases by testing each possible value with identical outcomes.

Equivalence classes (EC) are most suited to systems in which much of the input data takes on values within ranges or within sets, thereby significantly reducing the number of test cases that must be created and executed. One of the limitations of this technique is that it makes the assumption that the data in the same equivalence class is processed in the same way by the system.

Equivalence Partitioning can be defined according to the following guidelines:

- If an input condition specifies a range, one valid and one two invalid equivalence classes are defined.
- If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
- If an input condition is Boolean, one valid and one invalid equivalence class is defined.
- If an input condition is data typed, one valid equivalence class for the correct data type and at least one invalid equivalence class for a different data type are defined.
- If input descriptions have specific mandatory conditions, then identify one valid equivalence class for the specific mandatory condition and one invalid equivalence class where the mandatory condition is not met.

Note: Error masking can occur when more than one invalid value is contained in a single test case. The processing of the test case may be terminated when an earlier invalid value is executed thus never processing or evaluating subsequent invalid data. The result is not knowing how that invalid value processes.

Referring back to the earlier example: Withdrawal limit set to $100, read two variables, account balance and withdrawal amount. If there are sufficient funds in the account to cover the withdrawal amount the program prints out a message indicating that and if there is not sufficient funds a message is printed indicating that. If sufficient funds exist then the program checks to see if the withdrawal amount exceeds the withdrawal limit. If so, it prints a message that withdrawal exceeds limit.

Equivalence classes would be:

- Class of Withdrawal Amounts less than Account balance. Assume input of account balance equals $500. Potential test data points could be $400.
- Class of Withdrawal Amounts greater than or equal to account balance. Assume input of account balance equals $500. Potential test data points could be $600.
- Withdrawal amounts less than or equal to withdrawal limit. Withdrawal limit is $100. Potential test data points could be $50.
- Withdrawal amounts greater than withdrawal limit. Withdrawal limit is $100. Potential test data points could be $400.
7.4.2.1.1 Equivalence Partitioning Advantages

An advantage of the equivalence partitioning technique is that it eliminates the need for exhaustive testing. It enables testers to cover a large domain of inputs or outputs with a smaller subset of input data selected from an equivalence partition. This technique also enables the testers to select a subset of test inputs with a high probability of detecting a defect.

One of the limitations of this technique is that it assumes that data in the same equivalence partition is processed in the same way by the system. Note that equivalence partitioning is not a stand-alone method to determine test cases. It has to be supplemented by the ‘Boundary Value Analysis’ technique, which is discussed in the next section.

7.4.2.2 Boundary Value Analysis (BVA)

Boundary Value Analysis is a technique used for testing decisions made based on input values. When a range of values is validated, you must write test cases that explore the inside and outside edge or “boundaries” of the range. BVA is an important technique as it is well documented that input values at the extreme ends of an input domain cause more errors in system. BVA is used to identify errors at boundaries of Equivalence Classes. Boundary values include maximum, minimum, just inside/outside boundaries, and error values.

7.4.2.2.1 Steps for using Boundary Value Analysis

The steps for using Boundary Value Analysis testing are:

1. Identify the equivalence classes from the given requirements or scenario.
2. Identify the boundaries of each equivalence class.
3. Create a test case by using the following guidelines:
• Minimum – precision value
• Minimum
• Minimum + precision value
• At each EC boundary – precision value
• At each EC boundary
• At each EC boundary + precision value
• Maximum – precision value
• Maximum
• Maximum + precision value

Precision value refers to the number of significant digits to the right of the decimal place. For example, if the boundary is 15 and the unit is Integer, then the “below” point is 14 and the “above” point is 16.

Figure 7-9  Equivalence Partition Boundary

For example, the equivalence classes identified in 7.4.2.1 were:

• Withdrawal Amounts less than Account balance, data point $400
• Withdrawal Amounts greater than or equal to Account Balance, data point $600
• Withdrawal amounts less than or equal to withdrawal limit, data point $50
• Withdrawal amounts greater than withdrawal limit, data point $105

In this example each data point test case would have shown whether the equivalence class as rendered into code worked or not. However, if most errors occur at the boundaries of the EC then a more intelligent set of data points to test would be:

• Account Balance greater than or equal to Withdrawal Amounts. Balance withdrawal data points $499.99 and $500.00. Assume account balance equals $500.00.
• Account Balance less than Withdrawal Amounts, withdrawal data point $500.01. Assume account balance equal to $500.00.
• Withdrawal amounts less than or equal to withdrawal limit, data points $99.99 and $100.00. Assume withdrawal limit of $100.00.
- Withdrawal amounts greater than withdrawal limit, data point $100.01. Assume withdrawal limit of $100.00.
- Also data points selected at Account Balance minimum, $0.00, Account Balance maximum (if there is one designated), special case of withdrawal of $0.00, and potentially negative numbers.

**Figure 7-10 Boundary Value Analysis**

### 7.4.2.2 BCA Advantages

An advantage of the Boundary Value Analysis technique is that the technique helps discover contradictions in the actual system and the specifications, and enables test cases to be designed as soon as the functional specifications are complete. As discussed numerous times, the earlier testing can begin the better. BVA allows early static testing of the functional specifications.

This technique works well when the program to be tested is a function of several independent variables that represent bounded physical quantities.

### 7.4.2.3 Decision Tables

Decision tables are a concise method of representing equivalence partitioning. The rows of a decision table specify all the conditions that the input may satisfy. The columns specify different sets of actions that may occur. Entries in the table indicate whether the actions should be performed if a condition is satisfied. Typical entries are, “Yes,” “No,” or “Immaterial.”

Decision tables are used to describe and analyze problems that contain procedural decision situations characterized by one or more conditions; the state of these conditions determines the execution of a set of actions. Decision tables represent complex business rules based on a set of conditions.
7.4.2.3.1 Decision Table Characteristics

The decision table has the following characteristics:

- Lists all possible “conditions” (inputs) and all possible “actions” (outputs)
- There is a “rule” for each possible combination of “conditions”
- For each “condition,” it is identified as a “yes” (present), a “not” (not present) or an “X” for immaterial (the results is the same for either yes or no)

<table>
<thead>
<tr>
<th>Condition Entry</th>
<th>Rule – 1</th>
<th>Rule – 2</th>
<th>-----</th>
<th>Rule – p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition – 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition – 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition – m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action Entry</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action – 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action – 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action – n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7-8 Decision Table

- The upper left portion of the format is called the condition stub quadrant; it contains statements of the conditions. Similarly, the lower left portion is called the action stub quadrant; it contains statement of the actions. The condition entry and action quadrants that appear in the upper and lower right portions form a decision rule.
- The various input conditions are represented by the conditions 1 through m and the actions are represented by actions 1 through n. These actions should be taken depending on the various combinations of input conditions.
- Each of the rules defines a unique combination of conditions that result in the execution (firing) of the actions associated with the rule.
- All the possible combinations of conditions define a set of alternatives. For each alternative, a test action should be considered. The number of alternatives increases exponentially with the number of conditions, which may be express as $2^{\text{NumberOfConditions}}$. When the decision table becomes too complex, a hierarchy of new decision tables can be constructed.

7.4.2.3.2 Steps to Create Decision Tables

The steps to create a decision table are as follows:
1. Identify the conditions and their entries
2. Identify the actions (results)
3. Determine the maximum number of Rules using the following formula:
   \[2^n\] where \(n\) is the number of conditions
   e.g., 3 conditions: \(2^3 = 8\) rules
   4 conditions: \(2^4 = 16\) rules
4. Divide the “Entries” side into the # of Rules
5. Populate the cells using the following pattern:
   - The entries for condition 1 will be 50% “Yes” and 50% “No”
   - For condition 2, the entries will be a pattern of 25% “Yes” and 25% “No,” repeated across the table
   - This pattern continues for each subsequent condition...for example, condition 3 would have a pattern of 12.5% “Yes” and 12.5% “No”

<table>
<thead>
<tr>
<th>Conditions</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition 1</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Condition 2</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Condition 3</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actions</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Action 1</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action 2</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

6. Determine the action entry for each column
7. Collapse the table by eliminating rules that don’t need to be tested, as follows:
   a. When two rules have the same action entry and their condition entries only vary by one answer, eliminate the rule where it varies
   b. Next, analyze the table for redundancies and combine any rules that are now the same
   c. Repeat this process until all unnecessary test have been eliminated
8. Confirm the accuracy of the table with the following calculation:
   Number of Rules = \(n + 1\), where \(n\) is the number of conditions
### 7.4.2.3.3 Decision Table Example

This example uses the scenario of withdrawing cash from an ATM machine.
- Valid PIN
- Valid Account
- Account Balance \(\geq\) Withdrawal Amount

<table>
<thead>
<tr>
<th>Withdraw cash from ATM</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid PIN</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Valid Account</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Balance (\geq) Withdrawal</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Withdrawal</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No Withdrawal</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

*Table 7-9 Decision Table Example*

With the decision table created, mark the columns to collapse using the guidelines described in step 7 of section 7.4.2.3.3.

<table>
<thead>
<tr>
<th>Withdraw cash from ATM</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid PIN</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Valid Account</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Balance (\geq) Withdrawal</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Withdrawal</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No Withdrawal</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

*Table 7-10 Decision Table Example Updated*

After removing the first set of columns, repeat the process described in step 7 of section 7.4.2.3.3.
The final decision table represents the test cases that will need to be executed to test the possible combinations and results based on the requirements stated.

7.4.2.3.4 Decision Table Advantages

The advantage of decision table testing is that it allows testers to start with a “complete” view, with no consideration of dependence, then looking at and considering the “dependent,” “impossible,” and “not relevant” situations and eliminating some test cases.

The disadvantage of decision table testing is the need to decide (or know) what conditions are relevant for testing. Also, scaling up is challenging because the number of test cases increases exponentially with the number of conditions (scaling up can be massive: $2^n$ for $n$ conditions).

Decision table testing is useful for those applications that include several dependent relationships among the input parameters. For simple data-oriented applications that typically perform operations such as adding, deleting, and modifying entries, this technique is not appropriate.

7.4.2.4 Pair-Wise (All-Pairs) Testing Techniques

Pair-wise testing (also known as all-pairs testing) is a combinatorial method used to generate the least number of test cases necessary to test each pair of input parameters to a system. Pair-wise testing tests all possible discrete combinations of those parameters. Pair-wise testing is based on the notion that most faults are caused by interactions of at most two factors. Using the pair-wise technique provides a method to cover all combinations of two therefore reducing the number of tests yet still being very effective in finding defects. The All Pairs technique is also referred to as 2-way testing. It is also possible to do all triples (3-way) or all quadruples (4-way) testing, of course, but the size of the higher order test sets grows very rapidly.
7.4.2.4.1 Steps to Create Pair-Wise Tables

The steps to create a pair-wise table are as follows:

1. List out the variables in the application to be tested and the various possible values each of the variables can hold
2. Combine or group the values where ever possible
3. Create the all pairs table by putting the variables in the top row and start by filling in the values for the variables in each column
4. If a combination does not exist, then swap around with the values to see if the combination works
   Else add a new row
5. Each row in the table corresponds to a test case

7.4.2.4.2 Pair-Wise Table Example

Consider a Stock Trading Application which allows for Buying and Selling stocks in the equities market. It should support:

- Trading on NYSE and NASDAQ exchanges
- Stock codes for all securities traded (all 500 valid codes)
- Offer the following products - Delivery, Margin, and BTST (buy today sell tomorrow)
- Orders can be placed as a Market order and Limit order
- Orders can be placed at any time of the day; however, Margin orders can be placed only during Trading hours

Following the steps outline in section 7.4.2.4.1 begin by listing out the variables:

- Order Category: Buy, Sell
- Exchange: NYSE, NASDAQ
- Stock Code: Supported codes (500), Not supported codes
- Order Type: Margin, Limit
- Ordering Time: 24 hour period including trading and non-trading hours

Number of possible combinations that can be tested are (without considering Trading Hours)

\[ 3 \times 2 \times 2 \times 500 \times 2 = 12000 \]

If all 24 hours and all possible stock codes are taken into consideration, the number of valid and invalid combinations is enormous. To simplify the number of combination another way of looking at the variable is:

- Ordering Time can be considered as 2 values - Trading Hours and Non Trading Hours
- Stock code can be considered as 2 values - Valid stock code and Invalid stock code

Now the number of possible combinations shrinks to
Create all pairs table as:

<table>
<thead>
<tr>
<th>Product</th>
<th>Order Category</th>
<th>Exchange</th>
<th>Stock Code</th>
<th>Order Type</th>
<th>Trading Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delivery</td>
<td>Buy</td>
<td>NYSE</td>
<td>Valid</td>
<td>Market</td>
<td>Trading Hrs</td>
</tr>
<tr>
<td>Margin</td>
<td>Sell</td>
<td>NASDAQ</td>
<td>Invalid</td>
<td>Limit</td>
<td>Non Trading Hrs</td>
</tr>
<tr>
<td>BTST</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

List all the variable names under each category.

Now, rearrange and list out the values in each column. Start filling in the table column by column. Note the Order Category column has 2 values. That is how many times it is needed to insert the values of the first column, Product.

<table>
<thead>
<tr>
<th>Product</th>
<th>Order Category</th>
<th>Exchange</th>
<th>Stock Code</th>
<th>Order Type</th>
<th>Trading Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delivery</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delivery</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Margin</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Margin</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTST</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTST</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each set of values in column 1, put both values of column 2.
There is a Buy and NYSE, but there’s no Buy and NASDAQ. There is a Sell and NASDAQ, but there’s no Sell and NYSE. Rearrange the values in the second set in the third column.

<table>
<thead>
<tr>
<th>Product</th>
<th>Order Category</th>
<th>Exchange</th>
<th>Stock Code</th>
<th>Order Type</th>
<th>Trading Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delivery</td>
<td>Buy</td>
<td>NYSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delivery</td>
<td>Sell</td>
<td>NASDAQ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Margin</td>
<td>Buy</td>
<td>NASDAQ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Margin</td>
<td>Sell</td>
<td>NYSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTST</td>
<td>Buy</td>
<td>NYSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTST</td>
<td>Sell</td>
<td>NASDAQ</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rearrange the values as necessary. Now in 6 cases we have all the combinations.

<table>
<thead>
<tr>
<th>Product</th>
<th>Order Category</th>
<th>Exchange</th>
<th>Stock Code</th>
<th>Order Type</th>
<th>Trading Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delivery</td>
<td>Buy</td>
<td>NYSE</td>
<td>Valid</td>
<td></td>
<td>Market</td>
</tr>
<tr>
<td>Delivery</td>
<td>Sell</td>
<td>NASDAQ</td>
<td>Invalid</td>
<td></td>
<td>Limit</td>
</tr>
<tr>
<td>Margin</td>
<td>Buy</td>
<td>NASDAQ</td>
<td>Valid</td>
<td></td>
<td>Limit</td>
</tr>
<tr>
<td>Margin</td>
<td>Sell</td>
<td>NYSE</td>
<td>Invalid</td>
<td></td>
<td>Market</td>
</tr>
<tr>
<td>BTST</td>
<td>Buy</td>
<td>NYSE</td>
<td>Invalid</td>
<td>Limit</td>
<td></td>
</tr>
<tr>
<td>BTST</td>
<td>Sell</td>
<td>NASDAQ</td>
<td>Valid</td>
<td>Market</td>
<td></td>
</tr>
</tbody>
</table>
Now include ordering time.

<table>
<thead>
<tr>
<th>Product</th>
<th>Order Category</th>
<th>Exchange</th>
<th>Stock Code</th>
<th>Order Type</th>
<th>Trading Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delivery Buy</td>
<td>NYSE</td>
<td>Valid</td>
<td>Market</td>
<td>Trading hrs</td>
<td></td>
</tr>
<tr>
<td>Delivery Sell</td>
<td>NASDAQ</td>
<td>Invalid</td>
<td>Limit</td>
<td>Non Trading hrs</td>
<td></td>
</tr>
<tr>
<td>Margin Buy</td>
<td>NASDAQ</td>
<td>Valid</td>
<td>Limit</td>
<td>Non Trading hrs</td>
<td></td>
</tr>
<tr>
<td>Margin Sell</td>
<td>NYSE</td>
<td>Invalid</td>
<td>Market</td>
<td>Trading hrs</td>
<td></td>
</tr>
<tr>
<td>BTST Buy</td>
<td>NYSE</td>
<td>Invalid</td>
<td>Limit</td>
<td>Non Trading hrs</td>
<td></td>
</tr>
<tr>
<td>BTST Sell</td>
<td>NASDAQ</td>
<td>Valid</td>
<td>Market</td>
<td>Non Trading hrs</td>
<td></td>
</tr>
</tbody>
</table>

Since we do not have some combinations, include two more cases.

<table>
<thead>
<tr>
<th>Product</th>
<th>Order Category</th>
<th>Exchange</th>
<th>Stock Code</th>
<th>Order Type</th>
<th>Trading Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delivery Buy</td>
<td>NYSE</td>
<td>Valid</td>
<td>Market</td>
<td>Trading hrs</td>
<td></td>
</tr>
<tr>
<td>Delivery Sell</td>
<td>NASDAQ</td>
<td>Invalid</td>
<td>Limit</td>
<td>Non Trading hrs</td>
<td></td>
</tr>
<tr>
<td>Margin Buy</td>
<td>NASDAQ</td>
<td>Valid</td>
<td>Limit</td>
<td>Non Trading hrs</td>
<td></td>
</tr>
<tr>
<td>Margin Sell</td>
<td>NYSE</td>
<td>Invalid</td>
<td>Market</td>
<td>Trading hrs</td>
<td></td>
</tr>
<tr>
<td>BTST Buy</td>
<td>NYSE</td>
<td>Invalid</td>
<td>Limit</td>
<td>Non Trading hrs</td>
<td></td>
</tr>
<tr>
<td>BTST Sell</td>
<td>NASDAQ</td>
<td>Valid</td>
<td>Market</td>
<td>Non Trading hrs</td>
<td></td>
</tr>
</tbody>
</table>

7.4.2.4.3 Pair-Wise Advantages

The Pair-Wise testing technique can significantly reduce the number of test cases. It protects against pair-wise defects which represent the majority of combinatorial defects. There are tools available which can create the All pairs table automatically. Efficiency is improved because the much smaller pair-wise test suite achieves the same level of coverage as larger combinatorial test suites.
7.4.2.5 Cause-Effect Graphing (CEG)

In section 7.4.2.4, Pair-Wise Testing” described a method designed to ensure coverage of all pairs while minimizing the number of necessary test cases. This challenge of minimizing test cases while remaining confident test coverage is realistically maximized is substantial.

Cause-effect graphing is a technique which focuses on modeling the dependency relationships between a program’s input conditions (causes) and output conditions (effects). CEG is considered a Requirements-Based test technique and is often referred to as Dependency modeling. In CEG, the relationship between the input (causes) and the output (effects) are expressed visually as a cause-effect graph. Usually the graph shows the nodes representing the causes on the left side and the nodes representing the effects on the right side. There may be intermediate nodes in between that combine inputs using Boolean operators such as AND and OR.

In practice, CEG diagrams can be quite complex when visualizing complex systems. For simplicity of this discussion, only basic CEG diagrams are used.

7.4.2.5.1 Steps to Create a Cause-Effect Graph

The steps to create a Cause-Effect Graph are:

1. Identify the causes and effects from the application’s requirements. Each cause and effect is assigned a unique identifier.
2. Express the relationship between causes and effects using a cause-effect graph.

Diagram 1: Effect (ef) occurs if cause (c) occurs.

Diagram 2: Effect (ef) occurs if cause (c) does not occur.
Diagram 3: Effect (ef) occurs if cause (c₁) AND cause (c₂) occurs.

7.4.2.5.2 Cause-Effect Graph Example

Referring back to the earlier “cash withdrawal” example, we can list the causes as:

- C₁ - Account Balance > Withdrawal Amount
- C₂ - Withdrawal Amount < Withdrawal Limit
- ef - Withdrawal funds approved

<table>
<thead>
<tr>
<th>Approval to Withdraw Cash</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>AcctBal &gt; WithdrawalAmt</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>WithdrawalAmt ≤ WithdrawalMax</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>Withdrawal</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No Withdrawal</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

*Table 7-12 Decision Table based on Cause-Effect Diagram*
7.4.2.6 State Transition Testing

State Transition diagrams are an excellent tool to capture certain types of system requirements and to document internal system design. These diagrams document the events that come into and are processed by a system, as well as the system's responses.

When a system must remember what happened before, or when valid and invalid orders of operation exist, then state transition testing can be used. It is useful in situations when workflow modeling or dataflow modeling has been done (i.e., the system moves from one state to another).

Important terms related to state transition testing are explained below:

- **State** (represented by a circle) – A state is a condition in which a system is waiting for one or more events. States remember the inputs the system has received in the past and define how the system should respond to subsequent events when they occur. These events may cause state transitions and/or initiate actions. The states are generally represented by values of one or more variables within the system.

- **Transition** (represented by an arrow) – A transition represents a change from one state to another caused by an event.

- **Event** (represented by a label on a transition) – An event is something that causes a system to change state. Events can be independent or casually related (e.g., Event B cannot take place before Event A). When an event occurs, the system can change state or remain in the same state and/or execute an action. Events may have parameters associated with them.

- **Action** (represented by a command following a “/”) – An action is an operation initiated because of a state change. Often these actions cause something to be created that is an output of the system.

7.4.2.6.1 Steps to Use State Transition Testing

The steps for using state transition testing are:

1. Understand the various states that the system, user, or object can be in, including the initial and final states. Example of states can be: “User created a purchase order” or “Vacation request accepted.” These states will be represented as:

   ![Figure 7-11 Representation of a State](image)

2. Identify transitions, events, conditions, and actions that can - and can’t - apply in each state.
3. Use a graph or table to model the system. This graph or table also serves as a predictor of correct system behavior.

4. For each event and condition, that is, each transition, verify that the correct action and next state occur.

5. Create test cases in such a way that all states are visited at least once, all events are triggered at least once, and all paths are executed at least once (i.e., all transitions in the system are tested at least once).

### 7.4.2.6.2 State Transition Example

Using an employee vacation/leave application system in an organization to illustrate the use of state transition testing, the following characteristics describe that system. An employee can request vacation leave, and if he/she is eligible (based on the number of days he/she has already taken, etc.), the application is sent to the manager for approval. The manager then validates and approves the vacation leave or rejects the leave based on the duration, reason for taking leave, etc. You can use a State Transition diagram to represent the scenario and derive the required number of test cases.

The State Transition diagram (Figure 7-13) of the given scenario can be represented as follows:

The number of test conditions required to test this scenario would be three. These three paths are:
1. Employee requests leave that is accepted and approved by HR and the Manager
2. Employee requests leave that is accepted by HR but rejected by the Manager
3. Employee requests leave that is rejected by HR

State Transition testing can be used for applications characterized by a set of states, where data values (click of a button, inputs, etc.) that cause transition from one state to the other are well understood and the method of processing within each state is also understood clearly.

7.4.2.6.3 State Transition Testing Advantages

The advantages of State Transition testing are:
- It eliminates the need for exhaustive testing, which is not feasible.
- It enables a tester to cover a large domain of inputs and outputs, with a smaller subset of input data selected from an equivalence partition.
- It enables a tester to select a subset of test inputs with a high probability of detecting a defect.

The disadvantage of State Transition testing is that it becomes very large and cumbersome when the number of states and events increases.

7.4.2.7 Scenario Testing

Scenario Testing is exactly as it sounds, testing based on a real-world scenario of how the system is supposed to act. In section 7.1.4, Business Case Analysis was described as a method to identify test conditions from actual business cases. Scenario testing is done to make sure that the end to end functioning of software is working correctly or all the business process flows of the software are working correctly. During scenario testing the testers act as end users. Use Case Testing and (Agile) Story Testing are considered types of Scenario Testing. In scenario testing, testers work together with clients, stakeholders and developers to create test scenarios.

7.4.2.7.1 Key Characteristics of Scenario Testing

Dr. Cem Kaner, who has been widely recognized as coining the term “scenario testing,” identifies several key characteristics of scenario testing1:
- The test is based on a story about how the program is used, including information about the motivations of the people involved.
- The story is motivating. A stakeholder with influence would push to fix a program that failed this test. (Anyone affected by a program is a stakeholder. A person who can influence development decisions is a stakeholder with influence.)

• The story is credible. It not only could happen in the real world; stakeholders would believe that something like it probably will happen.
• The story involves a complex use of the program or a complex environment or a complex set of data.
• The test results are easy to evaluate. This is valuable for all tests, but is especially important for scenarios because they are complex.

7.4.2.7.2 Six Uses of Scenario Testing

Dr. Kaner goes on to list six uses of the scenario testing approach:

1. Learn the product.
2. Connect testing to documented requirements.
3. Expose failures to deliver desired benefits.
4. Make defect reports more motivating to stakeholders.
5. Explore expert use of the system.
6. Bring requirements-related issues to the surface, which might involve reopening old requirements discussions (with new data) or surfacing not-yet-identified requirements.

7.4.2.7.3 Twelve Ways to Create Good Scenarios

Finally, Dr. Kaner describes twelve ways to create good scenarios. They are:

1. Write life histories for objects in the system.
2. List possible users, analyze their interests and objectives.
3. Consider disfavored users: how do they want to abuse your system?
4. List “system events.” How does the system handle them?
5. List “special events.” What accommodations does the system make for these?
6. List benefits and create end-to-end tasks to check them.
7. Interview users about famous challenges and failures of the old system.
8. Work alongside users to see how they work and what they do.
9. Read about what systems like this are supposed to do.
10. Study complaints about the predecessor to this system or its competitors.
11. Create a mock business. Treat it as real and process its data.
12. Try converting real-life data from a competing or predecessor application.
7.4.3 Experience-based Techniques

The discussions on structural and functional testing as described in the previous sections do not, by any means, consider the value of an experienced test professional as ancillary. Quite the contrary, while the techniques described in 7.4.1 and 7.4.2 are well defined and in many cases have specific procedures for their use, the value of those techniques is zero and in some cases detrimental to the project when inexperienced team members employ them. Those techniques are systematic in nature and their process and procedures can be learned. However, testing is a challenging sport and of equal or greater importance is the capability of the human mind which brings intuition, imagination, and cumulative experience to the effort.

7.4.3.1 Error Guessing

Some people have a natural intuition for test case generation. While this ability cannot be completely described nor formalized, certain test cases seem highly probable to catch errors. For example, input values of zero and input values that cause zero outputs are cases where a tester may guess an error could occur. Guessing carries no guarantee for success, but neither does it carry any penalty.

7.4.4 Non-Functional Tests

Non-functional testing is the testing of a software application for its non-functional requirements. Non-functional items can include the testing of the software quality factors described in section 1.2.5. The list of non-functional tests includes:

- Compatibility testing
- Compliance testing
- Documentation testing
- Efficiency testing
- Endurance testing
- Functionality testing
- Internationalization testing
- Load testing
- Localization testing
- Maintainability testing
- Performance testing
- Portability testing
- Recovery testing
- Reliability testing
• Scalability testing
• Security testing
• Stress testing
• Usability testing
• Volume testing

7.4.4.1 Non-Functional Test Types

• Compatibility testing: Testing to validate that the application system is compatible with the operating environment, hardware, and other impacted/impacting software.

• Compliance testing: Testing to validate that the system under test complies with the stated requirements specifications, a standard, a contract, or regulation.

• Documentation testing: Testing specific life cycle documents against an accepted standard (e.g., IEEE 829). Document deliverables could include such items as the test plan, test cases, test report, and defect reports.

• Efficiency testing: Testing that measures the amount of computing resources and code required by a program to perform its required functions. (See section 1.2.5.1)

• Endurance testing: Testing that validates that the application under test can withstand the processing load it is expected to have for a significant period of time.

• Functionality testing: Testing that validates that the application under test performs and functions correctly according to design specifications.

• Internationalization testing: Testing that validates that the application under test works under different languages and regional settings which can include the ability to display accented characters, to run on non-English operating systems, to display the correct numbering system for thousands and decimal separators.

• Load testing: Testing that places demand on a system or device and measures its response. Load testing is performed to determine a system’s behavior under both normal and anticipated peak load conditions.

• Localization testing: Testing to ensure that the localized product is fully functional, linguistically accurate, and that no issues have been introduced during the localization process.

• Maintainability testing: Testing that the application has been developed such that the effort required to locate and fix an error in an operational program is acceptable. (See section 1.2.5.1)

• Performance testing: Testing that validates that both the online response time and batch run times meet the defined performance requirements.

• Portability testing: Testing that measures the effort required to transfer software from one configuration to another.

• Recovery testing: Recovery testing evaluates the contingency features built into the application for handling interruptions and for returning to specific points in the application processing cycle, including checkpoints, backups, restores, and restarts. This test also assures that disaster recovery is possible.
• Reliability testing: Testing that validates that the application under test can perform its intended function with required precision. (See section 1.2.5.1)

• Scalability testing: Testing that measures the application’s capability to scale up in terms of any of its non-functional capability which can include the load capacity, concurrent connections, number of transactions, etc.

• Security testing: Testing that evaluate the reasonableness of security procedures to prevent the average person from penetrating the application. Security testing’s intent is to reveal flaws in the security mechanisms that protect data and maintain functionality as intended.

• Stress testing: Testing that subjects a system, or components of a system, to varying environmental conditions that defy normal expectations. For example, high transaction volume, large database size or restart/recovery circumstances. The intention of stress testing is to identify constraints and to ensure that there are no performance problems.

• Usability testing: Testing that evaluates the effort required to learn, operate, prepare input, and interpret output of an application system. This includes the application’s user interface and other human factors of the application. This is to ensure that the design (layout and sequence, etc.) enables the business functions to be executed as easily and intuitively as possible. (See section 1.2.5.1)

• Volume testing: Testing that validates the application’s internal limitations. For example, internal accumulation of information, such as table sizes, or number of line items in an event, such as the number of items that can be included on an invoice, or size of accumulation fields, or data-related limitations, such as leap year, decade change, or switching calendar years.

### 7.5 Building Test Cases

Section 7.1 focused on identifying the testable conditions within the scope of the system under test. Section 7.2 and 7.3 described Use Cases and User Stories (respectively) and how testable conditions would be derived from them. Section 7.4 extensively described the different techniques including structural, functional, and non-functional. The collective objective of those sections was to know what to test and have the understanding of the techniques that could be used to accomplish the testing.

Test cases take what we learned needs to be tested, and combines it with the skillful use of the test techniques to precisely define what will be executed and what is being covered. Experience shows that it is uneconomical to test all conditions in an application system. Experience further shows that most testing exercises less than one-half of the computer instructions. Therefore, optimizing testing through selecting the most important processing events is the key aspect of building test cases.
7.5.1 Process for Building Test Cases

The recommended process for the creation and use of test cases is an five-step process as follows:

1. Identify the conditions to be tested. (Sections 7.1 – 7.3)
   A testing matrix is recommended as the basis for identifying conditions to test. As these matrices cascade through the development process, they identify all possible test conditions. These should be general test conditions at this step.

2. Rank test conditions.
   If resources are limited, the best use of those resources will be obtained by testing the most important test conditions. The objective of ranking is to identify high-priority test conditions that should be tested first. Considerations may include the stability of the system, level of automation, skill of the testers, test methodology, and, most importantly, risk.

   Ranking does not mean that low-ranked test conditions will not be tested. Ranking can be used for two purposes: first, to determine which conditions should be tested first; and second, and equally as important, to determine the amount of resources allocated to each of the test conditions.

3. Select conditions for testing.
   Based on the ranking, the conditions to be tested should be selected. At this point, the conditions should begin to cascade down from the general to the specific.

4. Determine correct results of processing.
   The correct processing results for each test condition should be determined. The correct time to determine the correct processing results is before the test cases have been created. This step helps determine the reasonableness and usefulness of the test case. The process can also show if there are ways to extend the effectiveness of test cases, and whether the same condition has been tested by another case.

5. Create test cases.

7.5.2 Documenting the Test Cases

There are many ways to document the test cases. Regardless of the specifics, certain fundamental items must be included such as a unique identifier for each test case, the inputs into the test case, the test procedure, and the expected results.

7.5.2.1 IEEE 29229-3 Documentation Standard for Test Cases

In section 1.4 of Skill Category 1, the ISO/IEC/IEEE 29119 was identified as an internationally recognized set of standards for software testing. Within the 29119 standard,
section 3 (29119-3) provides a standard for test documentation including the Test Case Specifications. There are many ways to document test cases. The level of detail is often driven by the level of risk associated with the application and stability of the application within the development cycle.

7.5.2.1.1 Documentation Standard for Test Cases

The IEEE 829 template for test case specification is shown below:

1. Test Case Specification Identifier
2. Test Items
3. Input Specifications
4. Output Specifications
5. Environment Needs
6. Special Procedural Requirements
7. Inter-Case Dependencies

**Test Case Specification Identifier** – A unique identifier that ideally follows the same rules as the software to which it is related. This is helpful when coordinating software and testware versions within configuration management.

**Test Items** - Identify the items or features to be tested by this test case. This could include requirements specifications, design specifications, detail design specifications, and code.

**Input Specifications** - Identify all inputs required to execute the test case. Items to include would be: data items, tables, human actions, states (initial, intermediate, final), files, databases, and relationships.

**Output Specifications** - Identify all outputs required to verify the test case. This would describe what the system should look like after the test case is run.

**Environmental needs** – This describes the hardware (configurations and limitations), software (system, operating environment, other interacting applications), facilities, and training.

**Special Procedural Requirements** - Identify any special constraints for each test case.

**Inter-Case Dependencies** - Identify any prerequisite test cases. One test case may require another case to run before it to setup the environment for the next case to run. It is recommended that the relationship of test cases be documented at both ends of the relationship. The precursor should identify any follow-on test cases and the post cases identify all prerequisites.
7.5.2.1.2 Other Test Case Examples

The IEEE 829 standard provides a good template for organizations to customize to their unique needs. Listed below is another example of a test case format.

- Test Suite ID - The ID of the test suite to which this test case belongs.
- Test Case ID - The ID of the test case.
- Test Case Summary - The summary/objective of the test case.
- Related Requirement - The ID of the requirement to which this test case relates/traces
- Preconditions - Any preconditions that must exist prior to executing the test.
- Test Procedure - Step-by-step procedure to execute the test.
- Expected Result - The expected result of the test.
- Actual Result - The actual result of the test.
- Status - Pass, Fail, Blocked, or Not Executed.
- Remarks - Any comments on the test case or test execution.
- Created By - The name of the author of the test case.
- Date of Creation - The date of creation of the test case.
- Executed By - The name of the person who executed the test.
- Date of Execution - The date of execution of the test.
- Test Environment - The environment (Hardware/Software/Network).

7.6 Test Coverage

Based upon the risk, and criticality associated with the application under test, the project team should establish a coverage goal during test planning. The coverage goal defines the amount of code that must be executed by the tests for the application. In those cases where the application supports critical functions, such as air traffic control or military defense systems, the coverage goal may be 100% at all stages of testing.

The objective of test coverage is simply to assure that the test process has covered the application. Although this sounds simple, effectively measuring coverage may be critical to the success of the implementation. There are many methods that can be used to define and measure test coverage, including:

- Statement Coverage
- Branch Coverage
- Basis Path Coverage
- Integration Sub-tree Coverage
- Modified Decision Coverage
- Global Data Coverage
- User-specified Data Coverage
It is usually necessary to employ some form of automation to measure the portions of the application covered by a set of tests. There are many commercially available tools that support test coverage analysis in order to both accelerate testing and widen the coverage achieved by the tests. The development team can also design and implement code instrumentation to support this analysis. This automation enables the team to:

- Measure the “coverage” of a set of test cases
- Analyze test case coverage against system requirements
- Develop new test cases to test previously “uncovered” parts of a system

Even with the use of tools to measure coverage, it is usually cost prohibitive to design tests to cover 100% of the application outside of unit testing or black-box testing methods. One way to leverage a dynamic analyzer during system testing is to begin by generating test cases based on functional or black-box test techniques. Examine the coverage reports as test cases are executed. When the functional testing provides a diminishing rate of additional coverage for the effort expended, use the coverage results to conduct additional white-box or structural testing on the remaining parts of the application until the coverage goal is achieved.

### 7.7 Preparing for Test Execution

Skill Category 8 covers the areas related to execution of the test. Regardless of the development methodology followed, it is critically important to first understand what needs to be tested, then use the appropriate design techniques to create the test cases and to always keep in mind that every action within the development life cycle introduces some form of risk. Risk controls such as peer-to-peer test case reviews are required to minimize the impact of product and process risks across the SDLC.
common mantra throughout the Software Testing Body of Knowledge (STBOK) has been test early, test often, and test throughout the development life cycle. Skill Category 6 covered the static testing processes of walkthroughs, checkpoint review, and inspections. Typically those test processes are used in the earlier phases of the life cycle. Skill Category 7 identified test techniques most of which can only be used when actual programming code exists. Whether the life cycle is organized as a series of Agile Scrum Sprints or a long term waterfall project; ideas coalesce, designs are made, code is written, and checks are done. To that end the goal of this Skill Category is to describe the processes of test execution across the life cycle utilizing the knowledge and skills defined in the previous skill categories of this the Software Testing Body of Knowledge.

<table>
<thead>
<tr>
<th>Skill Category 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceptance Testing</td>
</tr>
<tr>
<td>IEEE Test Procedure Specification</td>
</tr>
<tr>
<td>Test Execution</td>
</tr>
<tr>
<td>Testing Controls</td>
</tr>
<tr>
<td>Recording Test Results</td>
</tr>
<tr>
<td>Defect Management</td>
</tr>
</tbody>
</table>
8.1 Acceptance Testing

To begin with it is important to clarify that Acceptance Testing is not the same as User Acceptance Testing (UAT). User Acceptance Testing is a phase at or near the end of the software development life cycle during which the customer (or customer’s representative) has an opportunity to run the finished program in an environment that parallels the operational environment for the primary purpose of providing the customer with enough confidence in the application system to accept delivery. The V-diagram (Figure 1-20) in section 1.8 as well as table 1-12 in section 1.8.2.2 illustrates where UAT falls within the SDLC. By contrast, the objective of acceptance testing is to determine throughout the development cycle that all aspects of the development process meet the user’s needs. There are many ways to accomplish this. The user may require that the implementation plan be subject to an independent review of which the user may choose to be a part, or he or she may simply prefer to input acceptance criteria into the review process.

8.1.1 Incremental Acceptance

Acceptance across the SDLC is an incremental process of approving or rejecting interim deliverables during development according to how well the deliverable satisfies predefined criteria. Acceptance decisions occur at pre-specified times when processes, support tools, interim documentation, segments of the software, and finally the total software system must meet predefined criteria for acceptance. Subsequent changes to the software may affect previously accepted elements. The final acceptance decision occurs with verification that the delivered documentation is adequate and consistent with the executable system and that the complete software system meets all buyer requirements. Formal final software acceptance testing (UAT) must occur at the end of the development process. It consists of tests to determine whether the developed system meets predetermined functionality, performance, quality, and interface criteria.

8.1.2 Incremental Acceptance Benefits

Acceptance testing benefits include:

- Early detection of application problems (and time for the customer or user to plan for possible late delivery)
- Preparation of appropriate test facilities
- Encourages change tolerance by identifying changes as early as possible
- Accountability for software acceptance belongs to the customer or user of the software, whose responsibilities are:
  - Ensure user involvement in developing system requirements and acceptance criteria
Executing the Test Process

- Identify specific interim products for acceptance, their acceptance criteria, and schedule the acceptance tests
- Plan how and by whom each acceptance activity will be performed
- Plan resources for providing information on which to base acceptance decisions
- Respond to the analyses of project entities before accepting or rejecting them
- Approve the various interim deliverables against quantified criteria at interim points
- Perform the final acceptance activities, including formal acceptance testing, at delivery (i.e., User Acceptance Testing)

The customer or user must be actively involved in defining the type of information required, evaluating that information, and deciding at various points in the development activities if the products are ready for progression to the next activity.

8.1.3 Fitness for Use

Acceptance testing is designed to determine whether the software under development or as a final product is or will be “fit for use”. The concept of fit for use is important in both design and testing. Design must attempt to build the application to fit into the user’s business process; the test process must ensure a prescribed degree of fit. Testing that concentrates on structure and requirements may fail to assess fit, and thus fail to test the value of the application to the business. It is important to recognize that throughout the life cycle at each scheduled acceptance test point that the four components of fit must be considered. If an evaluation of fitness for use is relegated to the latter stages of development or dynamic testing, it is too late in the game to make effective changes.

8.1.3.1 Four Components of Fit

The four components of fit are:

1. **Data** - The reliability, timeliness, consistency, and usefulness of the data included in the application.
2. **People** - People should have the skills, training, aptitude, and desire to properly use and interact with the application.
3. **Structure** - The structure is the proper development of application systems to optimize technology and satisfy requirements.
4. **Rules** - The rules are the procedures to follow in processing the data.

The system must fit into these four components of the business environment as illustrated in Figure 8-1. If any of the components fail to fit properly, the success of the application system will be diminished. Therefore, acceptance testing must ensure that all the components are...
adequately prepared and developed, and that the four components fit together to provide the best possible solution to the business problem.

Acceptance testing should be an ongoing activity that tests both interim and final products. Do not wait until the end of the development process so that unnecessary time is expended making corrections that will prove unacceptable to the system user.

8.1.4 Acceptance Criteria

In Skill Category 5, section 8.2, acceptance criteria was defined as one of the prerequisites for test planning. Further, in Skill Category 7, section 7.3.2, acceptance criteria as it is defined specifically within the context of Agile and Agile User Stories was discussed. Here we define the process for the development of those acceptance criteria.

8.1.4.1 Developing the Acceptance Criteria

The process of developing the acceptance criteria resides with the product owner/customer/user and it is their responsibility to determine the criteria the software must meet to be deemed acceptable. Ideally, this is a deliverable of the software requirements specifications process. In preparation for developing the acceptance criteria, the user should:

- Acquire full knowledge of the application for which the system is intended
- Become fully acquainted with the application as it is currently implemented by the user’s organization (if a current version of the application is in operation)
• Understand the risks and benefits of the development methodology that is to be used in developing or maintaining the software system

• Fully understand the consequences of adding new functions to enhance an existing system

Acceptance requirements that a system must meet can be divided into these four categories:

• **Functionality** – Internal consistency of documents and code, in and between stages; traceability of functionality; adequate verification of logic; functional evaluation and testing; preservation of functionality in the operating environment.

• **Performance** – Feasibility analysis of performance requirements; correct simulation and instrumentation tools; performance analysis in the operating environment.

• **Interface Quality** – Interface documentation; interface complexity; interface and integration test plans; interface ergonomics; operational environment interface testing.

• **Overall Software Quality** – Quantification of quality measures; criteria for acceptance of all software products; adequacy of documentation and software system development standards; quality criteria for operations testing. These requirements also specify limits for factors or attributes such as reliability, testability, correctness, and usability.

### 8.1.4.2 Assessing Criticality

Assessing the criticality of a system is important in determining quantitative acceptance criteria. By definition, all safety criteria are critical; and by law, certain security requirements are critical. Some typical factors affecting criticality include:

• Importance of the system to organization or industry

• Consequence of failure

• Complexity of the project

• Technology risk

• Complexity of the user environment

Products or interim deliverables with critical requirements do not qualify for acceptance if they do not satisfy their acceptance criteria. A product with failed noncritical requirements may qualify for acceptance, depending upon quantitative acceptance criteria for quality factors. Clearly, if a product fails a substantial number of noncritical requirements, the quality of the product is questionable.

### 8.1.5 Process for Acceptance Testing

The end-game for acceptance testing is to determine whether the acceptance criteria have been met in an interim deliverable or a delivered product. This can be accomplished through reviews, which involve looking at interim products and partially developed deliverables at various points throughout the developmental process. It will also involve testing the executable software system at the User Acceptance Testing phase. The determination of
which (or both) of these techniques to use will depend on the criticality of the software, the size of the software program, the resources involved, and the time period over which the software is being developed.

Software acceptance criteria (see section 8.1.4) should be specified in the formal project plan. The plan identifies products to be tested, the specific pass/fail criteria, the reviews, and the types of testing that will occur throughout the entire life cycle.

### 8.1.5.1 Acceptance Decision

Acceptance decisions need a framework in which to operate. Items such as contracts, acceptance criteria, and formal mechanisms are part of this framework. Software acceptance must state or refer to specific criteria that products must meet in order to be accepted.

Typical acceptance decisions include:

- **Make all change** - Required changes are accepted and made before progressing to the next activity
- **Pick and choose** - Some changes must be made and accepted before further development of that section of the product; other changes may be made and accepted at the next major review
- **Defer all changes** - Progress may continue and changes may be accepted at the next review
- **No changes are required and progress may continue.**

The goal is to achieve and accept “perfect” deliverables, but usually some criteria will not be completely satisfied for each deliverable, in which case the user may choose to accept something less-than-perfect.

Final acceptance of software based on User Acceptance Testing usually means that the software project has been completed, with the exception of any caveats or contingencies. Final acceptance for the software occurs, and the development team has no further development obligations (except, of course, for maintenance, which is a separate issue).

### 8.1.5.2 Acceptance is a Contractual Process

Software acceptance is a contractual process during which users and the development team identify criteria for the acceptance of interim deliverables and the application system. The development team must agree to the users’ acceptance criteria. The users must define the acceptance criteria based on the system requirements for functionality, performance, interface quality, and overall software quality (see section 8.1.4.1).

The users must plan and manage the software acceptance program carefully to assure adequate resources are available throughout the acceptance activities. Early in the process, they must include detailed plans for software acceptance testing. Such early planning enables all those involved in the software project to focus on the requirements and how well the
executing system is satisfying those requirements. Software acceptance requires adequate resources and commitment from the beginning. Its completion will result in software that delivers to its users the services they require.

8.1.6 Acceptance Testing Summary

The reason for beginning the discussion of Executing the Test Process with Acceptance Testing, which might at first seem backwards, is to reinforce the fact that testing happens throughout the software development life cycle. Acceptance testing is a process that leads to an acceptance decision. The acceptance decision should be based on the results of testing compared to the acceptance criteria. Acceptance testing is not just about testing code but that all potential product deliverables can be subjected to acceptance testing within their respective life cycle phase.

In section 8.3.10 User Acceptance Testing as a testing phase that occurs late in the life cycle will be specifically discussed within the context of the other SDLC test phases.

8.2 IEEE Test Procedure Specification

Regardless of the test phase within the SDLC (i.e., unit, integration, system and UAT), it is helpful to understand the process steps of a generalized test procedure. The purpose of a test procedure is to specify the steps for executing a set of test cases or, more generally, the steps used to analyze a software item in order to evaluate a set of features. For this purpose, IEEE 829 provides a standard test procedure template. The test procedure specification is composed of the following sections:

Test procedure specification identifier - Specify the unique identifier assigned to this test procedure specification. Supplies a reference to the associated test design specification.

Purpose - Describe the purpose(s) of this procedure. If this procedure executes any test cases, provide a reference for each of them. In addition, provide references to relevant sections of the test item documentation (e.g., references to usage procedures).

Special requirements - Identify any special requirements that are necessary for the execution of this procedure. These may include prerequisite procedures, special skills requirements, and special environmental requirements.

Procedure steps:

Log - Describe any special methods or formats for logging the results of test execution, the incidents observed, and any other events pertinent to the test.
Set up - Describe the sequence of actions necessary to prepare for execution of the procedure.

Start - Describe the actions necessary to begin execution of the procedure.

Proceed - Describe any actions necessary during execution of the procedure.

Measure - Describe how the test measurements will be made (e.g., describe how remote terminal response time is to be measured during a network simulator).

Shut down - Describe the actions necessary to suspend testing—when unscheduled events dictate this.

Restart - Identify any procedural restart points and describe the actions necessary to restart the procedure at each of these points.

Stop - Describe the actions necessary to bring execution to an orderly halt.

Wrap up - Describe the actions necessary to restore the environment.

Contingencies - Describe the actions necessary to deal with anomalous events that may occur during execution.

8.3 Test Execution

Test execution is the operations of a test cycle. Each test cycle needs to be planned, prepared for, and executed, and the test results need to be recorded. This section addresses these components/activities involved in performing tests:

- Test Environment
- Test cycle strategy
- Test data
- Use of tools in testing
- Test Documentation
- Perform Tests
- Unit Testing
- Integration Testing
- System Testing
- User Acceptance Testing
- Testing COTS
- When is Testing Complete?
8.3.1 Test Environment

As discussed in Skill Category 2, Building the Software Testing Ecosystem, a test environment must be established for conducting tests. For example, for testing web-based systems, the test environment needs to simulate the type of platforms that would be used in the web environment.

Since the test scripts and test cases may need to run on different platforms, the platforms must be taken into consideration when designing test cases and test scripts. Since a large number of platforms may be involved in the operation of the software, testers need to decide which platforms to include in the test environment.

8.3.2 Test Cycle Strategy

Each execution of testing is referred to as a test cycle. Ideally the cycles are planned and included in the test plan. However, as defects are uncovered, and change is incorporated into the software, additional test cycles may be needed.

Software testers should determine the number and purpose of the test cycles to be used during testing. Some of these cycles will focus on the level of testing, for example unit, integration and system testing. Other cycles may address attributes of the software such as data entry, database updating and maintenance, and error processing.

8.3.3 Test Data

A significant challenge for the software tester is the creation of test data. Most testing processes require data to exercise the feature(s) being tested. Studies show that up to 60% of application development and testing time is devoted to data related tasks so the management of test data creation and maintenance is critically important.

There are potentially three distinct sets of test data required to test most applications; one set of test data to confirm the expected results (data along the happy path), a second set to verify the software behaves correctly for invalid input data (alternate paths or sad path), and finally data intended to force incorrect processing (e.g., crash the application). Volume testing requires the creation of test data as well.

Test data may be produced in a focused or systematic way or by using other, less-focused approaches such as high-volume randomized automated tests. Test data may be produced by the tester, or by a program or function that aids the tester. Test data may be recorded for re-use, or used once and then forgotten.

Some key issues to consider when creating test data are:
• Ensure the test data represents real world use
• Ensure data integrity
• Work to reduce the size of the test data to the minimum required without sacrificing necessary test execution
• Ensure all test conditions are covered
• Ensure any security concerns regarding the test data are addressed early in the process
• Ensure test data is available when needed and does not become a bottleneck during testing

8.3.3.1 Using Production Data

The use of production data appears to be the easiest and quickest means for generating test data. This is true if production data is used as is. Unfortunately, production data rarely provides good test data. To convert production data into good test data may be as time-consuming as the construction of test data itself.

8.3.3.1.1 Impediments to Using Production Data for Test Purposes

When production data is used for test purposes, the following impediments to effective testing may occur:

• **Needed test transactions missing** – Rarely will more than a few transaction types appear in a production dataset and those may be limited. For example, if the tester wanted to test an override of a standard price, that transaction may not occur within the selected production data.

• **Multiple tests of the same transaction** – In the production environment it is not uncommon for 80-90 percent of the transactions to be of approximately the same type. This means that while some transaction types will not be tested at all, others will literally be tested hundreds of times. The result is that certain paths through the system will be tested repetitively while other paths will not be tested at all.

• **Test results unknown** – An important part of testing is to validate that correct results are produced. When the tester creates the test transaction using known test data points, they have control over the expected results. However, when production data is used the testers must manually calculate the correct processing results. This can lead to misinterpretation of the intent of the transaction, miscalculation, or other errors and thus misinterpretation of the results.

• **Lack of ownership** – Production data is owned by the production area, not the testers. Test data created by the testers is owned by them. Some testers feel greater involvement and interest in test data they created themselves than in test data they “borrowed” from another owner.

• **Lack of security** – Production data may contain information that is confidential or for government applications may even be classified. For example, a production payroll system will contain pay rates.
The bottom-line result of these potential impediments is that testing with production data may not be effective. However, steps can be taken to improve the effectiveness of production data.

8.3.3.1.2 What is Production Data?

The following categories of production data exist for use in testing:

- **Transaction files/tables** – Business transactions being entered into business systems, for example, a list of items to be shipped on a particular day.
- **Business master files/tables** – A grouping of semi-permanent business data. Examples might be files of customers, employees, or vendors.
- **Master files/tables of business data** – Files or tables which record some of the processing data used by business software; for example, a list of products and their prices, a list of taxing communities and sales tax rates, and a list of employee pay rates.
- **Error files/tables** – A list of error transactions or error suspense files (i.e., transactions awaiting correction).
- **Operation/communication/data base/accounting logs** – A variety of files containing information about the operating environment such as logs that contain an indication of files used, execution time, invalid attempts to access, and so forth. These logs can be used to test states, attributes, stress, search, error, file, and match/merge tests.
- **Manual logs** – Logs maintained by users of the software which might record error messages, performance characteristics, results of processing, questions to be answered, and so forth.

8.3.3.1.3 Methods for Using Production Data as Test Data

The type of production data to be used for test data needs to be determined. The previous section listed the various categories of production data that could be used for test purposes. Once that category of production data is processed (for example, a business transaction file), then one or more of the following methods can be used to convert that production file into a test file:

1. Select the first X records
2. Protect the production file from modification, then use it as a test file
3. Select a random sample of transactions
4. Production file population analysis
5. Production file browsing
6. Parallel testing
8.3.3.1.3.1 Select the First X Records

Production files or tables are normally too lengthy to use for test purposes. Thus, they need to be shortened. One method for doing this is to predetermine the number of records wanted and then extract that number of records from the beginning of the file.

8.3.3.1.3.2 Protect the Production File, Then Use It as a Test File

Many files have protection procedures which will protect them from being modified during testing. This is particularly helpful when you want to use business master files or master files of business data. For example, if you wanted to process an accounts receivable file, you could use the production file. But whenever an update or change command to the file was initiated, the file would not be changed but merely a memo record made of that change.

8.3.3.1.3.3 Select a Random Sample of Transactions

This method develops a subset of the production file, but does it on a random basis. The benefit of doing a random basis selection is that there is an opportunity to create a test file which represents the various transactions on the production file, in the same type and frequency that they occur on the production file.

8.3.3.1.3.4 Production File Population Analysis

Population analysis is a technique used to identify the kinds and frequency of data that will be found in the production environment. Population analysis is creating reports that describe the type, frequency, and characteristics of the data to be used in testing. For example, for numerical fields, population analysis could describe the range of values of data contained in a field; for alphabetic data, identifying the longest name in a data field; for codes, identifying what codes and their frequency of use. The population analysis technique provides a profile of the production file (or selected subset) so the tester will understand what tests will actually be executed using the selected data.

By using the population analysis results, specific records can be selected which will address all of the extremes represented by the production file. For example, in a numeric field you can pick a record that has the lowest value and the highest value. For specific codes you can select all of the represented codes. This method can be particularly powerful because events can exist in the production environment which were not indicated on the specifications. For example, if five codes were permitted, but a sixth one was found on the production file, that would be included in the test data.

See section 7.1.2.1 for more information about population analysis.

8.3.3.1.3.5 Production Data Browsing

This method is a combination of using a production file and creating one's own test repository. The production file is browsed to determine the types of transactions on file. Once a desirable type of transaction is encountered, it is copied for test purposes.
The tester does not actually use the production data for test purposes but copies the data and values in all or part of a record for incorporation into test data. This has the benefit of creating test data that uses realistic production values. It also avoids the tester falling in the trap of using the same condition over and over again.

Normally this will require some sort of tool to interpret the production dataset so it can be browsed and records selected.

8.3.3.1.3.6 Parallel Testing

This method is only effective when there have been minimal changes between the current version of the production system and the new version. In this method all of the production files are used and the test results compared to previously performed production results.

There are two ways that this works effectively. The first is to make the changes to the production files that will permit the test results to be compared to the production results. The second method is using this for regression testing. In this case it is unimportant what the new processing performs. What is important is that those areas of processing that have not been affected by the change continue to produce correct results. In this method we are only trying to validate the unchanged portions of the software not validate the changed portions.

8.3.3.2 Manually Created Test Data

Manually created data is a common technique for creating test data. Creating data this way allows for specific data points to be included in the dataset that will test an application function whose contents have been predefined to meet the designed test conditions. For example, master file records, table records, and input data transactions could be generated after conditions are designed but prior to test execution, which will exercise each test condition/case. Once data is created to match the test conditions, specific expected results are determined and documented so that actual results can be checked during or after test execution.

8.3.3.2.1 Advantages of Using Manually Created Test Data

The advantages of using manually created test data include the following:

• Test results can be readily checked
• Happy path, alternate path, sad path data are controlled
• The types and combinations of transactions or procedures to be tested are known
• Test results are reliable without actually tracing data through the processing stages

8.3.3.2.2 Disadvantages of Using Manually Created Test Data

The disadvantages of using manually created test data include the following:

• Test data is valid only for the single application for which it has been specifically created
• Test procedures are valid only for a given point in time and therefore must be updated to incorporate any occurrences that would affect the validity of the tests (e.g., changes in file structure, statues, rules, or regulations)
• Developing good test data is costly and time-consuming
• Manually created test data generally cannot test every conceivable situation in a typical application because of the virtually limitless variety of transactions and conditions that can occur

8.3.3.2.3 Guidelines for Manually Creating Test Data

Guidelines for creating test data include:
• If a condition to be tested involves master record table updating, choose records that will be changed only by this single condition. (This permits an isolated evaluation of the test results and ensures the record is not influenced by other tests).
• Predetermine data input and expected results. Use screen, report, or record layouts to document test input and expected output. This will speed up test execution and analysis (assumes manual method of test execution).
• Store a master copy of test data in its initial state. In this way, test data can be reused throughout the life of the system/unit. Consider storing various “snapshots” of the test data for process/run/batch-oriented systems. “Refresh” test beds from the master backup each time testing is performed. Keep all data in the master backup current.
• Add, change, and delete test data as the system changes. Secure the master set to control changes and prevent inadvertent deletion.

8.3.3.3 Test Data Management (TDM)

Terminology

Test Data Management

Regardless of the approach to developing test data, having a defined strategy for the development, use, maintenance and ultimately destruction of that data is critically important. Like any other part of the development life cycle, test data’s creation and use injects additional risk into the process. A mature TDM strategy will help reduce this associated risk.

8.3.3.3.1 Test Data Management Lifecycle

There are a variety of models describing the TDM lifecycle. A simple but sufficient model includes:
• Analysis
• Design
• Creation
• Use
• Maintenance
• Destruction
**Analysis** – This step identifies the types of data needed based on the defined test conditions. Also, the frequency the data will be refreshed and where the test data will be stored.

**Design** – The design step includes implementing the data storage infrastructure, securing any tools that might be used, and completing any prep work that might be necessary before the data creation step.

**Creation** – This step would follow the sequence of events as described in sections 8.3.3.1 and 8.3.3.2 above.

**Use** – The shaded section of Figure 8-2 represents the use of test data.

---

*Figure 8-2 Test Data Use*

The process for using test data as illustrated in Figure 8-2 is:

1. Determine test cases to be run
2. As necessary, add any new test data
3. Backup the test data
4. Perform test(s)
5. Evaluate test(s)

**Maintenance** – The test data will require maintenance for a variety of reasons. Reasons might include: remove obsolete data, update to align with current version, additional functionality to test, and correction of errors found in test data.
Destruction – At some point, the test data will no longer be of use. Appropriate deletion of
the test data after archiving should be consistent with the security concerns of data.

8.3.4 Use of Tools in Testing

Testing, like program development, generates large amounts of information, necessitates
numerous computer executions, and requires coordination and communication between team
members and various stakeholders. Test tools, as described in Skill Category 2, section 2.4,
can ease the burden of test design, test execution, general information handling, and
communication. Test tool use within the test execution process serves both to help administer
the testing process as well as perform automation on designated test procedures.

8.3.5 Test Documentation

Most guidelines for software documentation during the development phase recommend that
test documentation be prepared for all multipurpose or multi-user projects and for other large
software development projects. The preparation of a Test Plan (see Skill Category 5) and
issuing a Test Analysis Report (see Skill Category 9) is recommended. As with all types of
documentation, the extent, formality, and level of detail of the test documentation are
functions of IT standards and may vary depending upon the size, complexity, and risk of the
project.

As noted in section 1.4.3.3 of Skill Category 1, the ISO 29119’s part 3 provides standard
templates for test documentation that cover the entire software testing life cycle. For more
information on the ISO 29119 standard visit www.softwaretestingstandard.org.

8.3.6 Perform Tests

In a life cycle approach to testing, tests can be performed throughout the project life cycle,
from testing requirements through conducting user acceptance testing. This discussion will
focus on the performance of the dynamic testing that is planned for an application.

The more detailed the test plan, the easier this task becomes for the individuals responsible for
performing the test. The test plan (Skill Category 5) should have been updated throughout the
project in response to approved changes made to the application specifications or other project
constraints (i.e., resources, schedule). This process ensures that the true expected results have
been documented for each planned test.

The roles and responsibilities for each stage of testing should also have been documented in
the test plan. For example, the development team (programmers) might be responsible for unit
testing in the development environment, while the test team is responsible for integration and
system testing in the test environment.
The Test Manager is responsible for conducting the Test Readiness Review prior to the start of testing. The purpose of this review is to ensure that all of the entrance criteria for the test phase have been met, and that all test preparations are complete.

The test plan should contain the procedures, environment, and tools necessary to implement an orderly, controlled process for test execution, defect tracking, coordination of rework, and configuration and change control. This is where all of the work involved in planning and set-up pays off.

For each phase of testing, the planned tests are performed and the actual results are compared to the documented expected results. When an individual performs a test script, they should be aware of the conditions under test, the general test objectives, as well as specific objectives listed for the script. All tests performed should be logged in a test management tool (or in a manual log if not using a tool) by the individual performing the test.

The Test Log (manual or automated) records test activities in order to maintain control over the test. It includes the test ID, test activities, start and stop times, pass or fail results, and comments. Be sure to document actual results. Log the incidents into the defect tracking system once a review determines it is actually a defect.

The IEEE 829 provides a standard for Software Test Documentation which defines the Test Log as a chronological record of relevant details about the execution of test cases. The IEEE template contents include: 1) Test Log Identifier; 2) Description; and 3) Activity and Event Entries.

When the development team communicates the defect resolution back to the test team, and the fix is migrated to the test environment, the problem is ready for retest and execution of any regression testing associated with the fix.

### 8.3.6.1 Regression Testing

Regression, to relapse to a less perfect state, in section 1.10.1 of Skill Category 1 described regression testing this way:

“Regression testing isn’t an approach, a style, or a testing technique. Regression testing is a ‘decision.’ It is a decision to re-test something that has already been tested for the express purpose of looking for defects that may have been inadvertently introduced or manifested as an unintended consequence of other additions or modifications to the application code, operating system, or other impacting program. Simply stated, the purpose of regression testing is to make sure unchanged portions of the system work as they did before a change was made.”

### 8.3.6.1.1 When is Regression Testing Perform?

Regression testing is not a separate phase of testing, and is not maintenance testing. Regression testing must occur whenever changes to tested units, environments, or procedures occur. For that reason the discussion about regression testing processes happens now before
detailed discussion about unit, integration and system testing. Unfortunately, regression testing is often inadequate in organizations and poses a significant risk to an application.

Regression testing should be performed when:

- New releases of packages software are received
- Application software is enhanced or any changes made
- Support software changes (OS, utilities, object libraries)
- Either side of a system interface is changes
- Changes to configuration
- Whenever changes are made after a testing stage is completed

Regression testing can be one of the most challenging testing processes within an organization because testers are looking for defects in applications that have already passed in previous test cycles.

8.3.6.1.2 A Regression Test Process

Regression testing will happen throughout the life cycle and for that reason a regression testing approach will vary depending on at what stage it is conducted. Shown here is an example of a regression test process which carefully introduces changes on the test side so not to mask off defects or allow defects to be injected into our test suite.

An eight-step process can be used to perform regression testing. As a prerequisite, the assumption is that a complete set of test cases and test data (TS1) exists that thoroughly exercises the current unchanged version of the software (V1). The following steps show the process (see Figure 8-3):

Step 1 – Change is introduced into the application system V1 which creates V2 of the application.

Step 2 – Test the changed software V2 against the unchanged test cases and test data TS1. The objective is to show that unchanged sections of the software continue to produce the same results as they did in version 1. If this is done manually, testing the changed portions of the system can be bypassed. If an automated tool is used, running TS1 against the V2 may produce totally invalid results. These should be disregarded. Only the unchanged portions are evaluated here.

Step 3 – Create an updated version of the test cases and test data (TS2) by removing tests that are no longer needed due to changes in the system.

Step 4 – Run test of TS2 against V2. This not only tests the changed portions, but provides another regression test of the unchanged portions. By creating the clean TS2 it prevents the tester from introducing errors in the form of new tests and test data added to test suite.

Step 5 – Create TS3 which is new cases and data designed to exercise just the new functionality.
Executing the Test Process

Step 6 – Run test of TS3 against V2. This tests new functionality only but more importantly it tests the tests with little interaction or impact from other test suite (TS2).

Step 7 – Combine TS2 and TS3 to create a full test suite of all cases and data (TS4) necessary to thoroughly exercise the entire system (V2).

Step 8 – Run test of TS4 against V2.

Figure 8-3  Regression Test Process

8.3.6.1.3 Using Selective Regression Testing

Selective regression testing is the process of testing only those sections of a program where the tester’s analysis indicates programming changes have taken place and the related components. Limited regression testing versus complete regression testing should only be considered if:

- Configuration control is in place for application releases (consisting of enhancements and/or fixes).
- The application’s components and their relationships or dependencies are documented.
- Test resources are limited or when regression testing is not automated.
- Changes can be highlighted on existing documents to isolate analysis.

8.3.6.1.3.1 A Process for Analyzing Changes

To perform selective regression testing, each change must be analyzed and the impact on other system components assessed. A process for analyzing changes is:
• Identify each changes component. A component may be a SCREEN, REPORT, DATA ELEMENT, DATA-FLOW PROCESS, etc.
• Identify the nature of the change and its relationship to other affected components. Use of data flow diagrams, case repositories, or other tools which cross-reference or relate components is helpful.
• If the changes are “local” (i.e., processes, data flows, and I/Os), then only the unchanged portions of those components need to be regression tested.
• If the changes are “global” (i.e., existing, new, deleted data elements, data element rules, values/validation, data store layouts, global variables, table contents, etc.), then all components related to those components must be regression tested.

8.3.7 Unit Testing

Units testing is normally performed by the programmer who developed the program. Unit testing can be performed in many ways but the result of unit testing should be that the unit is defect free. In other words, the program performs as specified. Integration testing should not occur until the units included in integration testing are defect free.

There is no universal definition of a unit; it depends on the technologies and scope of work. A unit can be:
• On program module
• One function/feature
• In Object-Oriented:
  o A Class or
  o The functionality implemented by a method
• A window or elements of a window
• A web page
• A Java applet or servlet
• An Active Server Page (ASP)
• A Dynamic Link Library (DLL) object
• A Common Gateway Interface (CGI) script

Basically, a unit is the smallest item that can be created or modified.

Ideally, the developer is in the best position to perform unit testing. The developer can test both function and structure and from a dynamic testing point of view unit testing is the earliest opportunity to test.

However, there are challenges for developers. Often developers lack of objectivity because they are so closely tied to the code. Also, pressures of meeting extremely tight schedules, a lack of training for developers in test techniques, and few if any processes, environment or tools for testing cause issues. John Dodson, manager of software engineering at Lockheed
Martin stated that, “Most of the college folks I get have a real good background in building software and a horrible background in testing it.”

The procedural flow of unit testing is not dissimilar from other testing phases. It is what is tested that differs. The majority of white-box testing techniques discussed in section 7.4.1 (e.g., statement testing) are used most often in unit testing.

Other concerns the development team must consider as part of the development, unit test, and fix process is Software Entropy. That is the tendency for software, over time, to become difficult and costly to maintain. It is by the very nature of software that systems tend to undergo continuous change resulting in systems that become more complex and disorganized. Software refactoring is the process of improving the design of existing software code. Refactoring doesn't change the observable behavior of the software; it improves its internal structure. For example, if a programmer wants to add new functionality to a program, they may decide to refactor the program first to simplify the addition of new functionality and to prevent software entropy.

### 8.3.7.1 IEEE 1008 Standard for Software Unit Testing

IEEE 1008 provides a standard for unit testing which includes the following:

1. **Scope and References**
   1.1. Inside the Scope.
   1.2. Outside the Scope.

2. **Definitions**

3. **Unit Testing Activities**
   3.2. Determine Features to be Tested
   3.3. Refine the General Plan
   3.4. Design the Set of Tests
   3.5. Implement the Refined Plan and Design
   3.6. Execute the Test Procedures
   3.7. Check for Termination
   3.8. Evaluate the Test Effort and Unit

8.3.8 Integration Testing

Integration testing should begin once unit testing for the components to be integrated is complete, and should follow the basic testing procedure as described earlier (section 8.2). The objectives in this stage of testing are to validate the application design, and prove that the application components can be successfully integrated to perform one or more application functions.

Depending on the sequence and design of the integration test “builds,” the application may be ready to enter System Test once the pre-defined exit criteria have been met.

8.3.8.1 Test Harness, Drivers, and Stubs

In section 1.10.4, incremental testing was defined as a disciplined method of testing the interfaces between unit-tested programs as well as between system components. It involves adding unit-tested programs to a given module or component one by one, and testing each resultant combination. Incremental testing is the process that happens during the integration test phase. Within the context of incremental testing is top down and bottom up. A test harness is utilized when other related code has not yet been completed but testing must proceed.

8.3.8.1.1 Top Down Testing using Stubs

The top down approach requires stubs for each lower component and only one “top level” driver is needed. Stubs can be difficult to develop and may require changes for different test cases.
8.3.8.1.2 Bottom Up Testing using Drivers

The bottom up approach begins with the components with the fewest dependencies. A driver causes the component under test to exercise the interfaces. As you move up the drivers are replaced with the actual components.
8.3.9 System Testing

System testing should begin as soon as a minimal set of components has been integrated and successfully completed integration testing. System testing ends when the test team has measured system capabilities and corrected enough of the problems to have confidence that the system will operate successfully in production. That is to say, system testing ends when the system test plan has been fully executed.

Once test planning is complete, preparation activities and actual test execution begins. Although many activities may be included in this process, the major steps are outlined below:

- Set up system test environment, mirroring the planned production environment as closely as possible.
- Establish the test data.
- Identify test cases that will be included in the system test.
- Identify test cycles needed to replicate production where batch processing is involved.
- Assign test cases to test cycles; note that in applications where the processing is real-time the test sets may still need to be grouped by planned days if the sequence of test execution is critical.
- Assign test scripts to testers for execution.
- Review test results and determine whether problems identified are actually defects.
- Record defect(s) in a tracking system, making sure the developer responsible for fixing the defect(s) is notified.
- When a defect is fixed and migrated to the test environment, re-test and validate the fix. If the defect is fixed, close the defect. If the defect is not fixed, return it to the developer for additional work.

The system test focuses on determining whether the requirements have been implemented correctly. This includes verifying that users can respond appropriately to events such as month-end processing, year-end processing, business holidays, promotional events, transaction processing, error conditions, etc.

In web and mobile testing, the test team must also prove that the application runs successfully on all supported hardware and software environments. This can be very complex with applications that must also support various web browsers and mobile devices.

8.3.10 User Acceptance Testing (UAT)

In section 8.1, Acceptance Testing, a clear delineation between Acceptance Testing and User Acceptance Testing was made. As stated in that section, User Acceptance Testing is a phase at or near the end of the software development life cycle during which the customer (or customer’s representative) has an opportunity to run the finished program in an environment that parallels the operational environment for the primary purpose of providing the customer
with enough confidence in the application system to accept delivery. The V-diagram (Figure 1-20) in section 1.8 as well as Table 1-12 in section 1.8.2.2 illustrates where UAT falls within the SDLC. User Acceptance Testing is performed by user personnel and may include assistance from software testers.

User Acceptance Testing should focus on input processing, use of the software in the user organization, and whether or not the applications meet the true processing needs of the user. Sometimes these user needs are not included in the specifications; sometimes these user needs are incorrectly specified in the software specifications; sometimes the user was unaware that without certain attributes of the system, the system would not be acceptable; and sometimes the user just does know exactly what they want. Examples include users not specifying the skill level of the people who will be using the system; processing may be specified but turnaround time not specified, and the user may not know how to articulate security concerns.

8.3.10.1 User's Role in UAT

The user’s role in UAT begins with the user making the determination as to whether UAT will or will not occur. If UAT is to occur the user has the primary responsibility for planning and conducting this phase of testing. This assumes that the users have the necessary testing competency to develop and execute a UAT test plan.

If the user does not have the needed competency to develop and execute the UAT test plan, the user will need to acquire that competency. Normally, the IT organization’s software testers would assist the user in the UAT process if additional competency is needed.

At a minimum the user will have the following roles in user acceptance testing:

- Defining acceptance criteria in a testable format
- Providing the necessary resources, primarily user staff personnel, for UAT
- Comparing the actual testing results with the acceptance criteria
- Making decisions as to whether additional work is needed prior to placing the software in operation, whether the software can be placed in operation with additional work to be done after promotion, or whether the software is fully acceptable and can be placed into production as is

If the software does not fully meet the user needs, but will be placed into operation, the user must develop a strategy to anticipate problems and pre-define the actions to be taken should those problems occur.

8.3.10.2 Software Tester's Role

Software testers can have one of three roles in acceptance testing. First is no involvement at all. In that instance the user accepts full responsibility for developing and executing the UAT plan. The second role is that of an advisor. The user will develop and execute the test plan, but rely on software testers to compensate for a lack of competency on the part of the users, or to
provide a quality control role. The third role is an active participant in software testing. This role can include any or the entire UAT activities.

The role of the software tester cannot include defining the acceptance criteria, or making the decision as to whether or not the software can be placed into operation. If software testers are active participants in UAT, then they may conduct any part of the testing up to and including the point where the results of the testing are documented.

A role that software testers should accept is to work with the users to develop the UAT process.

8.3.10.3 Execute the UAT Plan

The objective of this step is to determine whether the acceptance criteria have been met. The processes defined in sections 8.1.4 and 8.1.5 are consistent in intent and are applicable to User Acceptance Testing.

8.3.11 Testing COTS Software

Many organizations purchase Commercial Off The Shelf Software (COTS) for use within their organizations. Sometimes the COTS programs run completely “as-is” out of the box and while other COTS programs might have significant customization and interface software written in-house. Regardless of whether the COTS software is a multimillion dollar enterprise wide accounting system or a mobile application the conceptual challenges for the software testers are the same.

8.3.11.1 Testing COTS Challenges

COTS software is normally developed prior to an organization selecting that software for its use. For smaller, less expensive software packages the software is normally “shrink wrapped” and is purchased as is. As the COTS software becomes larger and more expensive, the contracting organization may be able to specify modifications to the software.

Challenges faced with testing COTS software include:

- Tasks or items missing
- Software fails to perform
- Extra features
- Does not meet business needs
- Does not meet operational needs
- Does not meet people needs
8.3.11.2 COTS Software Test Process

COTS software is developed and tested by the development company. Once delivered to the acquiring organization, the acquiring organization begins its software testing activities. The amount of testing to be performed on COTS software will be determined by the risk associated with placing that software into operation. If the risk is minimal, a decision may be made not to test. If the risk is high, extensive testing may be required.

The following seven step test process is designed to test high-risk COTS software. As the risk decreases so should the test effort. Reducing the test effort can be accomplished by eliminating all or parts of these seven steps that are discussed here:

- Assure completeness of needs specification
- Define critical success factor
- Determine compatibility with your computer environment
- Assure the software can be integrated into your business system work flow
- Demonstrate the software in operation
- Evaluate the people fit
- Acceptance test the COTS software

8.3.11.2.1 Assure Completeness of Needs Specification

This step determines whether you have adequately defined your needs. Your needs should be defined in terms of the following two categories of outputs:

1. Output products and reports
   
   Output products and reports are specific documents that you want produced by the computer system. In many instances, such as printing payroll checks for example, the style and format of these output products is important. Computer-produced reports may also be important for tax information (e.g., employee withholding forms sent to governmental units), financial statements where specific statements are wanted (e.g., balance sheets or statements of income and expense), or contracting organization invoice and billing forms which you might want preprinted to include your logo and conditions of payment.

2. Management decision information
   
   This category tries to define the information needed for decision-making purposes. In the computer product/report category you were looking for a document; in this case you are looking for information. How that information is provided is unimportant. Thus, the structure of the document, what the documents are, or their size, frequency, or volume are not significant. All you need is information.
8.3.11.2.2 Define Critical Success Factor

This step determines whether the software package will be successful in meeting the business needs. Critical Success Factors (CSFs) are those criteria or factors that must be present in the acquired software for it to be successful. A question that is often asked is, “Are the ‘needs’ the same as critical success factors?” They are, but needs are not defined in a manner that makes them testable, and they may be incomplete.

Often the needs do not take into account some of the intangible criteria that make the difference between success and failure. In other words, the needs define what we are looking for, and the critical success factors tell us how we will evaluate that product after we get it. They are closely related and complementary, but different in scope and purpose. The following list helps to illustrate the differences by using the needs and requirements for the automobile, and then the CSFs on which the automobile will be evaluated:

- Automobile requirements and needs: seats six people, has four doors, has a five-year guarantee on motor, gets 20 miles per gallon or greater, and costs under $12,000.
- Critical success factors: operates at 20.5 cents or less per mile, experiences no more than one failure per year, maintains its appearance without showing signs of wear for two years.

Use some of these more common CSFs when testing COTS software:

- Ease of use – the software is understandable and usable by the average person.
- Expandability – the vendor plans to add additional features in the future.
- Maintainability – the vendor will provide support/assistance to help utilize the package in the event of problems.
- Cost-effectiveness – the software package makes money for your business by reducing costs, and so on.
- Transferability – if you change your computer equipment the vendor indicates that they will support new models of hardware.
- Security – the system has adequate safeguards to protect the data against damage (for example, power failures, operator errors, or other situations that could cause you to lose your data).

8.3.11.2.3 Determine Compatibility with Your Computer Environment

This is not a complex step. It involves a simple matching between the processing capabilities and limitations of the current operating environment, and what the vendor of the software says is necessary to run the software package. The most difficult part of this evaluation is ensuring the multiple software packages can properly interface.

This step is best performed by preparing a checklist defining your compatibility needs. Software vendors are generally good about identifying the needed hardware and operating system compatibility.
In addition to the hardware on which the software runs, and the operating system with which it must interact to run, there are two other important compatibilities:

- Compatibility with other software packages (software vendors are generally not good in identifying compatibility with other software packages)
- Compatibility with available data

If you have no other software packages that you want to have interact with this one, or no data on computer-readable media, you need not worry about these aspects of compatibility. However, as you do more with your computer these aspects of compatibility will become more important, and the hardware and operating compatibility will become routine and easy to verify.

Systems compatibility is defined as interoperability. This term refers to the amount of effort required to intercouple or interconnect computer systems. In other words, how do you tie two or more programs together so that they will work and pass data between them? For example, if you have a payroll system it may be desirable to pass that payroll summary information to your general-ledger system. The ability to pass information from system to system is an extremely important part of most applications.

To help assure compatibility, prepare a list for the following items:

- Hardware Compatibility
- Operating Systems Compatibility
- Software Compatibility
- Data Compatibility

8.3.11.2.4 Assure the Software can be Integrated into Your Business System Work Flow

The objective of this process is to illustrate the type and frequency of work flow changes that will be occurring. When the COTS system is brought into your organization. For example, there might be tasks performed now that weren’t performed before, or tasks that were previously performed but are no longer necessary, or tasks which had been performed by people which will now be performed by the COTS system. Having the COTS system perform those tasks might mean that the oversight that people had been giving will not be completed any more.

At the end of this test, you will need to decide whether you are pleased with the revised work flow. If you feel the changes can be effectively integrated into your work flow, the potential COTS system has passed the test. If you feel the changes in work flow will be disruptive, you may want to fail the COTS in this test and either look for other software or continue your current processing.

If the testing is to continue, you should prepare a clean data flow diagram indicating what actions need to be taken to integrate the COTS system into your organization’s work flow. This new data flow diagram becomes your installation plan of action. It will tell you what changes need to be made, who is involved in them, what training might be necessary, and areas of potential work flow problems.
8.3.11.2.5 Demonstrate the Software in Operation

This step entails observing the functioning of software during operation (or as close to operation as possible). Demonstrations can be performed in either of the following ways:

- **Vendor offices - controlled demonstration**
  
  In this mode, the demonstration is conducted at the vendor’s site, by vendor personnel, using their data. The objective is to show you various aspects of the application, but not to let you get too involved in the process.

  - **On-site demonstration -** In this mode, the demonstration takes place at acquiring organization’s site, under the acquirer’s control, by the acquirer’s personnel, using the acquirer’s information. It is by far the most desirable of all demonstrations.

  - **Off-site demonstration –** In this mode the demonstration takes place at an organization which has already implemented the COTS software under consideration by the acquiring organization. In this situation the acquirer can see how the system works in a real environment by a similar organization. This also allows the acquiring team to ask questions of the users. Be aware that the vendor normally hand picks these sites because they have been successful in implementing the COTS program.

These aspects of computer software should be observed during the demonstration:

- **Understandability** - As you watch and listen to the demonstration, you need to evaluate the ease with which the operating process can be learned. If the commands and processes appear more like magic than logical steps, you should be concerned about implementing the concept in your organization. If you have trouble figuring out how to do it, think about how difficult it may be for some of your clerical personnel who understand neither the business application nor the computer.

- **Clarity of communication** - Much of the computer process is communication between man and machine. That is, you must learn the language of the computer software programs in order to communicate with the computer. Communication occurs through a series of questions and responses. If you do not understand the communications, you will have difficulty using the routine.

- **Ease of use of instruction manual** - While monitoring the use of the equipment, the tasks being demonstrated should be cross-referenced to the instruction manual. Can you identify the steps performed during the demonstration with the same steps included in the manual? In other words, does the operator have to know more than is included in the manual, or are the steps to use the process laid out so clearly in the manual that they appear easy to follow?

- **Functionality of the software** - Ask to observe the more common functions included in the software: Are these functions described in the manual? Are these the functions that the salesperson described to you? Are they the functions that you expected? Concentrate extensively on the applicability of those functions to your business problem.

- **Knowledge to execute** - An earlier test has already determined the extent of the salesperson’s knowledge. During the demonstration, you should evaluate whether a lesser-skilled person could as easily operate the system with minimal training.
Question the demonstrator about how frequently they run the demonstration and how knowledgeable they are about the software.

- **Effectiveness of help routines** - Help routines are designed to get you out of trouble when you get into it. For example, if you are not sure how something works you can type the word “help” or an equivalent and the screen should provide you additional information. Even without typing “help” it should be easy to work through the routines from the information displayed on the screen. Examine the instructions and evaluate whether you believe you could have operated the system based on the normal instructions. Then ask the operator periodically to call the help routines to determine their clarity.

- **Evaluate program compatibility** - If you have programs you need to interact with, attempt to have that interaction demonstrated. If you purchased other software from the same store where you are now getting the demonstration, they should be able to show you how data is passed between the programs.

- **Data compatibility** - Take one of your data files with you. Ask the demonstrator to use your file as part of the software demonstration. This will determine the ease with which existing business data can be used with the new software.

- **Smell test** - While watching the demonstration, let part of your mind be a casual overseer of the entire process. Attempt to get a feel for what is happening and how that might impact your business. You want to end up being able to assess whether you feel good about the software. If you have concerns, attempt to articulate them to the demonstrator as well as possible to determine how the demonstrator responds and addresses those concerns. Sometimes a “warm and fuzzy feeling” or NOT, is a good indicator along with the more objective methods.

To determine whether an individual has the appropriate skill level to use the software it is recommended to involve one or more typical users of the software in software demonstrations.

**8.3.11.2.6 Evaluate the People Fit**

The objective of this step is to determine whether the acquiring organization’s employees can use the software. This step evaluates whether employees possess the skills necessary to effectively use computers in their day-to-day work. The evaluation can be of current skills, or the program that will be put into place to teach individuals the necessary skills.

**8.3.12 Acceptance Test the COTS Software**

There is little difference between acceptance testing in-house developed software and acceptance testing acquired COTS software. As is with other acceptance testing processes, COTS acceptance testing is a user responsibility.
8.3.13 When is Testing Complete?

How do you know when testing is complete? Most testers might answer, “When I run out of time!” but there is only one factor the Test Manager can use to make this decision, “when the master test plan is completed” (see section 5.3). The Test Manager must be able to report, with a high degree of confidence, that the application will perform as expected in production, whether the quality goals defined at the start of the project have been met, and that the system is “fit for use” (see 8.1.3.1). The purpose of the test plan is to provide a roadmap to reach that level of confidence so “subjectively” speaking testing is complete when confidence that the system meets the need has been established and “objectively” speaking testing is complete when the test plan has been fully executed. They should be one and the same.

On most software test projects the probability of everything going exactly as planned is small. Scope creep, development delays, and other intervening events may require that the test plan be updated during the test execution process to keep the plan aligned with the reality of the project. This contingency would have been planned for in the test plan. Regardless of why a change is necessary what is critical is that the impact on project risk be identified and documented and that all stakeholders sign-off on the changes.

8.4 Testing Controls

From an academic perspective, the sole purpose of control is to reduce risk. Therefore, if there is no risk, there is no need for control. It is important that the test organization test the application’s controls to ensure that the controls are in place and working.

8.4.1 Environmental versus Transaction Processing Controls

It is important for the quality professional to know that there are two components of controls. The first is environmental (sometimes called general controls), and the second is the transaction processing controls within an individual business application.

8.4.2 Environmental or General Controls

Environmental controls are the means which management uses to manage the organization. They include such things as:

- Organizational policies
- Organizational structure in place to perform work
- Method of hiring, training, supervising and evaluating personnel
- Processes provided to personnel to perform their day-to-day work activities, such as a system development methodology for building and testing software systems.
Auditors state that without strong environmental controls the transaction processing controls may not be effective. For example, if passwords needed to access computer systems (a transactional control) are not adequately protected (environmental control) the password system will not work. Individuals will either protect or not protect their password based on environmental controls such as the attention management pays to password protection, the monitoring of the use of passwords that exist, and management’s actions regarding individual worker’s failure to protect passwords.

Two examples of management controls are the review and approval of a new system and limiting computer room access.

- **Review and Approval of a New System**
  
  This control should be exercised to ensure management properly reviews and approves new IT systems and conversion plans. This review team examines requests for action, arrives at decisions, resolves conflicts, and monitors the development and implementation of system projects. It also oversees user performance to determine whether objectives and benefits agreed to at the beginning of a system development project are realized.

  The team should establish guidelines for developing and implementing system projects and define appropriate documentation for management summaries. They should review procedures at important decision points in the development and implementation process.

- **Limiting Access to Computer Resources**

  Management controls involve limiting access to computer resources. It is necessary to segregate the functions of business analysts, programmers, testers, and computer operators. Business analysts and programmers should not have physical access to the operating programs, and the computer files. Use of production files should be restricted to computer operating personnel. Such a restriction safeguards assets by making the manipulation of files and programs difficult. For example, assume a bank’s programmer has programmed the demand deposit application for the bank. With his knowledge of the program, access to the files in the computer room on which information about the demand depositors is contained may allow him to manipulate the account balances of the bank’s depositors (including his own balance if he is a depositor).

### 8.4.3 Transaction Processing Controls

The object of a system of controls in a business application is to minimize business risks. Risks are the probability that some unfavorable event may occur during processing. Controls are the totality of means used to minimize those business risks.

There are two systems in every business application. As illustrated in Figure 8-6, the first is the system that processes business transactions, and the second is the system that controls the processing of business transactions. From the perspective of the
system designer, these two are designed and implemented as one system. For example, edits that determine the validity of input are included in the part of the system in which transactions are entered. However, those edits are part of the system that controls the processing of business transactions.

Because these two systems are designed as a single system, most testers do not conceptualize the two systems. Adding to the difficulty is that the system documentation is not divided into the system that processes transactions and the system that controls the processing of transactions.

![Diagram of the Two Systems in Every Business Application](image)

Figure 8-6  The Two Systems in Every Business Application

When one visualizes a single system, one has difficulty in visualizing the total system of control. For example, if one looks at edits of input data by themselves, it is difficult to see how the totality of control over the processing of a transaction is implemented. For example, there is a risk that invalid transactions will be processed. This risk occurs throughout the system and not just during the editing of data. When the system of controls is designed it must address all of the risks of invalid processing from the point that a transaction is entered into the system to the point that the output deliverable is used for business purposes.

A point to keep in mind when designing tests of controls is that some input errors may be acceptable if they do not cause an interruption in the application’s processing. A simple example of this would be a misspelled description of an item. In deciding on controls, it is necessary to compare the cost of correcting an error to the consequences of accepting it. Such trade-offs must be determined for each application. Unfortunately there are no universal guidelines available.

It is important that the responsibility for control over transaction processing be separated as follows:

- Initiation and authorization of a transaction
- Recording of the transaction
- Custody of the resultant asset

In addition to safeguarding assets, this division of responsibilities provides for the efficiencies derived from specialization, makes possible a cross-check that promotes accuracy without duplication or wasted effort, and enhances the effectiveness of a management control system.
8.4.3.1 Preventive, Detective, and Corrective Controls

This section describes three different categories of transaction processing controls, preventive, detective, and corrective controls and provides examples of these types of controls. Also provided is a detailed process to follow when building controls within an information system.

While this activity falls outside the scope of testing, knowing how the software is designed can greatly improve your ability to design appropriate test plans and processes.

The objectives of transaction processing controls are to prevent, detect, or correct incorrect processing. Preventive controls will stop incorrect processing from occurring; detective controls identify incorrect processing; and corrective controls correct incorrect processing. Since the potential for errors is always assumed to exist, the objectives of transaction processing controls will be summarized in five positive statements:

- Assure that all authorized transactions are completely processed once and only once.
- Assure that transaction data is complete and accurate.
- Assure that transaction processing is correct and appropriate to the circumstances.
- Assure that processing results are utilized for the intended benefits.
- Assure that the application can continue to function.

In most instances controls can be related to multiple exposures. A single control can also fulfill multiple control objectives. For these reasons transaction processing controls have been classified according to whether they prevent, detect, or correct causes of exposure. The controls listed in the next sections are not meant to be exhaustive but, rather, representative of these types of controls.

8.4.3.1.1 Preventive Controls

Preventive controls act as a guide to help things happen as they should. This type of control is most desirable because it stops problems from occurring. Application designers should put their control emphasis on preventive controls. It is more economical and better for human relations to prevent a problem from occurring than to detect and correct the problem after it has occurred.

Preventive controls include standards, training, segregation of duties, authorization, forms design, pre-numbered forms, documentation, passwords, consistency of operations, etc.

One question that may be raised is, “At what point in the processing flow is it most desirable to exercise computer data edits?” The answer to this question is simply, “As soon as possible, in order to uncover problems early and avoid unnecessary computer processing.”

Preventive controls are located throughout the entire application. Many of these controls are executed prior to the data entering the program’s flow. The following preventive controls will be discussed in this section:
Data Input - The data input process is typically a manual operation; control is needed to ensure that the data input has been performed accurately.

Turn-Around Documents - Other control techniques to promote the accuracy of input preparation include the use of turn-around documents which are designed to eliminate all or part of the data to be recorded at the source. A good example of a turn-around document is the bill which you may receive from a utility company. Normally the bill has two parts: one part is torn off and included with the remittance you send back to the utility company as payment for your bill; the other you keep for your records. The part you send back normally includes pre-recorded data for your account number and the amount billed so that this returned part can be used to record the transaction.

Pre-Numbered Form - Sequential numbering of the input transaction with full accountability at the point of document origin is another traditional control technique. This can be done by using pre-numbered physical forms or by having the application issue sequential numbers.

Input Validation - An important segment of input processing is the validation of the input itself. This is an extremely important process because it is really the last point in the input preparation where errors can be detected before processing occurs. The primary control techniques used to validate the data are associated with the editing capabilities of the application. Editing involves the ability to inspect and accept (or reject) transactions according to validity or reasonableness of quantities, amounts, codes, and other data contained in input records. The editing ability of the application can be used to detect errors in input preparation that have not been detected by other control techniques.

The editing ability of the application is achieved by installing checks in the program of instructions, hence the term program checks. They include:

- Validity tests - Validity tests are used to ensure that transactions contain valid transaction codes, valid characters, and valid field size.
- Completeness tests - Completeness checks are made to ensure that the input has the prescribed amount of data in all data fields. For example, a particular payroll application requires that each new employee hired have a unique User ID and password. A check may also be included to see that all characters in a field are either numeric or alphabetic.
- Logical tests - Logical checks are used in transactions where various portions, or fields, of the record bear some logical relationship to one another. An application can check these logical relationships to reject combinations that are erroneous even though the individual values are acceptable.
• Limit tests - Limit tests are used to test record fields to see whether certain predetermined limits have been exceeded. Generally, reasonable time, price, and volume conditions can be associated with a business event.

• Self-checking digits - Self-checking digits are used to ensure the accuracy of identification numbers such as credit card numbers. A check digit is determined by performing some arithmetic operation on the identification number itself. The arithmetic operation is formed in such a way that typical errors encountered in transcribing a number (such as transposing two digits) will be detected.

• Control totals - Control totals serve as a check on the completeness of the transaction being processed. Control totals are normally obtained from batches of input data. For example, daily batch control totals may be emailed to a company allowing them to cross check with the credit card receipts for that day.

**Computer Updating of Files** - The updating phase of the processing cycle entails the computer updating files with the validated transactions. Normally computer updating involves sequencing transactions, comparing transaction records with master-file records, computations, and manipulating and reformatting data, for the purpose of updating master files and producing output data for distribution to user departments for subsequent processing.

**Controls over Processing** - When we discussed input validation, we saw that programmed controls are a very important part of application control. Programmed controls in computer updating of files are also very important since they are designed to detect loss of data, check arithmetic computation, and ensure the proper posting of transactions.

Three examples of programmed controls are:

• A control total is made from amount or quantity fields in a group of records and is used to check against a control established in previous or subsequent manual or computer processing.

• A hash total is another form of control total made from data in a non-quantity field (such as vendor number or customer number) in a group of records.

• Programmed checks of arithmetic calculations include limit checks, cross-footing balance checks, and overflow tests.

### 8.4.3.1.2 Detective Controls

Detective controls alert individuals involved in a process so that they are aware of a problem. Detective controls should bring potential problems to the attention of individuals so that action can be taken. One example of a detective control is a listing of all time cards for individuals who worked over 40 hours in a week. Such a transaction may be correct, or it may be a systems error, or even fraud.

Detective controls will not prevent problems from occurring, but rather will point out a problem once it has occurred. Examples of detective controls are batch control documents, batch serial numbers, clearing accounts, labeling, and so forth.

The following detective controls will be discussed here:
• Control totals
• Control register
• Documentation and testing
• Output Checks

**Control totals** - Control totals are normally obtained from batches of input data. These control totals are prepared manually, prior to processing, and then are incorporated as input to the data input process. The application can accumulate control totals internally and make a comparison with those provided as input. A message confirming the comparison should be printed out, even if the comparison did not disclose an error. These messages are then reviewed by the respective control group.

**Control Register** - Another technique to ensure the transmission of data is the recording of control totals in a log so that the input processing control group can reconcile the input controls with any control totals generated in subsequent computer processing.

**Output Checks** - The output checks consist of procedures and control techniques to:

- Reconcile output data, particularly control totals, with previously established control totals developed in the input phase of the processing cycle
- Review output data for reasonableness and proper format
- Control input data rejected by the computer during processing and distribute the rejected data to appropriate personnel

Proper input controls and file-updating controls should give a high degree of assurance that the output generated by the processing is correct. However, it is still useful to have certain output controls to achieve the control objectives associated with the processing cycle. Basically, the function of output controls is to determine that the processing does not include any unauthorized alterations by the computer operations section and that the data is substantially correct and reasonable.

**8.4.3.1.3 Corrective Controls**

Corrective controls assist individuals in the investigation and correction of causes of risk exposures that have been detected. These controls primarily collect evidence that can be utilized in determining why a particular problem has occurred. Corrective action is often a difficult and time-consuming process; however, it is important because it is the prime means of isolating system problems. Many system improvements are initiated by individuals taking corrective actions on problems.

It should be noted that the corrective process itself is subject to error. Many major problems have occurred in organizations because corrective action was not taken on detected problems. Therefore detective control should be applied to corrective controls. Examples of corrective controls are: error detection and re-submission, audit trails, discrepancy reports, error statistics, and backup and recovery. Error detection and re-submission, and audit trail controls are discussed below.
• Error Detection and Re-submission - Until now we have talked about data control techniques designed to screen the incoming data in order to reject any transactions that do not appear valid, reasonable, complete, etc. Once these errors have been detected, we need to establish specific control techniques to ensure that all corrections are made to the transactions in error and that these corrected transactions are reentered into the system. Such control techniques should include:

  o Having the control group enter all data rejected from the processing cycle in an error log by marking off corrections in this log when these transactions are reentered; open items should be investigated periodically.

  o Preparing an error input record or report explaining the reason for each rejected item. This error report should be returned to the source department for correction and re-submission. This means that the personnel in the originating or source department should have instructions on the handling of any errors that might occur.

  o Submitting the corrected transactions through the same error detection and input validation process as the original transaction.

• Audit Trail Controls - Another important aspect of the processing cycle is the audit trail. The audit trail consists of documents, journals, ledgers, and worksheets that enable an interested party (e.g., the auditor) to track an original transaction forward to a summarized total or from a summarized total backward to the original transaction. Only in this way can they determine whether the summary accurately reflects the business’s transactions.

8.4.3.1.4 Cost versus Benefit of Controls

In an application system there is a cost associated with each control. The cost of these controls needs to be evaluated as no control should cost more than the potential errors it is established to detect, prevent, or correct. Also, if controls are poorly designed or excessive, they become burdensome and may not be used. The failure to use controls is a key element leading to major risk exposures.

Preventive controls are generally the lowest in cost. Detective controls usually require some moderate operating expense. On the other hand, corrective controls are almost always quite expensive. Prior to installing any control, a cost/benefit analysis should be made. Controls need to be reviewed continually.

8.5 Recording Test Results

A defect is a condition that exists within the application system that needs to be addressed. Carefully and completely documenting a defect is the first step in correcting the defect. Realistically most test organization today utilize some form of automation for recording and tracking defects. These tools drive the tester through the process. This section provides the underlying rationale for why tools require what they do.

The following four attributes should be developed for all defects:
• Statement of condition – Tells what is.
• Criteria – Tells what should be.
• Please note that the two attributes above are the basis for a finding. If a comparison between the two gives little or no practical consequence, no finding exists.
• Effect – Tells why the difference between what is and what should be is significant.
• Cause – Tells the reasons for the deviation. Identification of the cause is necessary as a basis for corrective action.

A well-developed defect statement will include each of these attributes. When one or more of these attributes is missing, questions usually arise, such as:
• Criteria – Why is the current state inadequate?
• Effect – How significant is it?
• Cause – What could have caused the problem?

8.5.1 Deviation from what should be

Defect statements begin to emerge by a process of comparison. Essentially the user compares “what is” with “what should be.” When a deviation is identified between what is found to actually exist and what the user thinks is correct or proper, the first essential step toward development of a defect statement has occurred. It is difficult to visualize any type of defect that is not in some way characterized by this deviation. The “what is” can be called the statement of condition. The “what should be” shall be called the criteria. These concepts are the first two, and most basic, attributes of a defect statement.

The documenting of deviation is describing the conditions, as they currently exist, and the criteria, which represents what the user desires. The actual deviation will be the difference or gap between “what is” and “what is desired.”

The statement of condition is uncovering and documenting the facts, as they exist. What is a fact? If somebody tells you something happened, is that “something” a fact? On the other hand, is it only a fact if someone told you it’s a fact? The description of the statement of condition will of course depend largely on the nature and extent of the evidence or support that is examined and noted. For those facts making up the statement of condition, the IT professional will obviously take pains to be sure that the information is accurate, well supported, and worded as clearly and precisely as possible.

The statement of condition should document as many of the following attributes as appropriate for the defect:
• Activities involved – The specific business or administrative activities that are being performed.
• Procedures used to perform work – The specific step-by-step activities that are utilized in producing output from the identified activities.
• Outputs/Deliverables – The products that are produced from the activity.
• Inputs – The triggers, events, or documents that cause this activity to be executed.
• User/Customers served – The organization, individuals, or class users/customers serviced by this activity.
• Deficiencies noted – The status of the results of executing this activity and any appropriate interpretation of those facts.

Table 8-1 is an example of the types of information that should be documented to describe the defect and document the statement of condition and the statement of criteria. Note that an additional item could be added to describe the deviation.

<table>
<thead>
<tr>
<th>Name of Application Under Tested</th>
<th>Put the name of the software system or subsystem tested here.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Description</td>
<td>Write a brief narrative description of the variance uncovered from expectations.</td>
</tr>
<tr>
<td>Statement of Conditions</td>
<td>Put the results of actual processing that occurred here.</td>
</tr>
<tr>
<td>Statement of Criteria</td>
<td>Put what the testers believe was the expected result from processing.</td>
</tr>
<tr>
<td>Effect of Deviation</td>
<td>If this can be estimated, testers should indicate what they believe the impact or effect of the problem will be on computer processing.</td>
</tr>
<tr>
<td>Cause of Problem</td>
<td>The testers should indicate what they believe is the cause of the problem, if known. If the testers are unable to do this, the worksheet will be given to the development team and they should indicate the cause of the problem.</td>
</tr>
<tr>
<td>Location of Problem</td>
<td>The testers should document where the problem occurred as closely as possible. It can be related to a specific instruction or processing section that is desirable. If not, the testers should try to find the location as accurately as possible.</td>
</tr>
<tr>
<td>Recommended Action</td>
<td>The testers should indicate any recommended action they believe would be helpful to the project team. If the testers feel unable to indicate the action needed, the project team would record the recommended action here. Once approved, then the action would be implemented. If not approved, an alternate action should be listed or the reason for not following the recommended action should be documented.</td>
</tr>
</tbody>
</table>

Table 8-1 Defect Documentation Guide
8.5.2 Effect of a Defect

Whereas the legitimacy of a defect statement may stand or fall on criteria, the attention that the defect statement gets after it is reported depends largely on its significance. Significance is judged by effect.

Efficiency, economy, and effectiveness are useful measures of effect and frequently can be stated in quantitative terms such as dollars, time, and units of production, number of procedures and processes, or transactions. Where past effects cannot be ascertained, potential future effects may be presented. Sometimes, effects are intangible, but nevertheless of major significance.

In thought processes, effect is frequently considered almost simultaneously with the first two attributes of the defect. Testers may suspect a bad effect even before they have clearly formulated these other attributes in their minds. After the statement of condition is identified the tester may search for a firm criterion against which to measure the suspected effect.

The tester should attempt to quantify the effect of a defect wherever practical. While the effect can be stated in narrative or qualitative terms, that frequently does not convey the appropriate message to management; for example, statements like “Service will be delayed,” do not really tell what is happening to the organization.

8.5.3 Defect Cause

The cause is the underlying reason for the condition. In some cases the cause may be obvious from the facts presented. In other instances investigation will need to be undertaken to identify the origin of the defect.

Most findings involve one or more of the following causes:

- Nonconformity with standards, procedures, or guidelines
- Nonconformity with published instructions, directives, policies, or procedures from a higher authority
- Nonconformity with business practices generally accepted as sound
- Employment of inefficient or uneconomical practices

The determination of the cause of a condition usually requires the scientific approach, which encompasses the following steps:

Step 1. Define the defect (the condition that results in the finding).
Step 2. Identify the flow of work and information leading to the condition.
Step 3. Identify the procedures used in producing the condition.
Step 4. Identify the people involved.
Step 5. Recreate the circumstances to identify the cause of a condition.
8.5.4 Use of Test Results

Decisions need to be made as to who should receive the results of testing. Obviously, the developers whose products have been tested are the primary recipients of the results of testing. However, other stakeholders have an interest in the results including:

- End users
- Software project manager
- IT quality assurance

It is important to note that the individual whose results are being reported receive those results prior to other parties. This has two advantages for the software tester. The first is that the individual, whom testers believe may have made a defect, will have the opportunity to confirm or reject that defect. Second it is important for building good relationships between testers and developers to inform the developer who made the defect prior to submitting the data to other parties. Should the other parties contact the developer in question prior to the developer receiving the information from the tester, the developer would be put in a difficult situation. It would also impair the developer-tester relationship.

8.6 Defect Management

A major test objective is to identify defects. Once identified, defects need to be recorded and tracked until appropriate action is taken. This section explains a philosophy and a process to find defects as quickly as possible and minimize their impact.

This section also outlines an approach for defect management. This approach is a synthesis of the best IT practices for defect management. It is way to explain a defect management process within an organization.

Although the tester may not be responsible for the entire defect management process, they need to understand all aspects of defect management. The defect management process involves these general principles:

- The primary goal is to prevent defects. Where this is not possible or practical, the goals are to both find the defect as quickly as possible and minimize the impact of the defect.
- The defect management process, like the entire software development process, should be risk driven. Strategies, priorities and resources should be based on an assessment of the risk and the degree to which the expected impact of a risk can be reduced.
- Defect measurement should be integrated into the development process and be used by the project team to improve the development process. In other words, information on defects should be captured at the source as a natural by-product of doing the job. It should not be done after the fact by people unrelated to the project or system.
• As much as possible, the capture and analysis of the information should be automated.
• Defect information should be used to improve the process. This, in fact, is the primary reason for gathering defect information.
• Imperfect or flawed processes cause most defects. Thus, to prevent defects, the process must be altered.

8.6.1 Defect Naming

It is important to name defects early in the defect management process. This will enable individuals to begin articulating more specifically what the defect is.

Name of the Defect - Name defects according to the phase in which the defect most likely occurred such as, requirements defect, design defect, documentation defect, and so forth.

Defect Severity - Use three categories of severity as follows:
• Critical - The defect(s) would stop the software system from operating.
• Major - The defect(s) would cause incorrect output to be produced.
• Minor - The defect(s) would be a problem but would not cause improper output to be produced, such as a system documentation error.

Defect Type - Indicates the cause of the defect. For example, code defects could be errors in procedural logic, or code that does not satisfy requirements or deviates from standards.

Defect Class - The following defect categories are suggested for each phase:
• Missing - A specification was not included in the software.
• Wrong - A specification was improperly implemented in the software.
• Extra - An element in the software was not requested by a specification

8.6.1.1 Defect-Naming Example

If a requirement was not correct because it had not been described completely during the requirements phase of development, the name of that defect using all 3 levels might be:
• Name – Requirement defect
• Severity – Minor
• Type - Procedural
• Class – Missing

8.6.2 The Defect Management Process

Figure 8-7 illustrates the key elements of a defect management process. Each element is described below.
As many quality experts have pointed out, the best approach to defects is to eliminate them altogether. Until the technology exists to guarantee that defects will not be created, strategies will be needed to find them as quickly as possible and minimize their impact. Nevertheless, there is much that organizations can do to prevent defects. Identifying the best defect-prevention techniques (which is a large part of identifying the best software development processes) and implementing them should be a high-priority activity in any defect management program.

Figure 8-8 illustrates a defect prevention process with three major steps that are described below.

Defect prevention should begin with an assessment of the critical risks associated with the system. Once the critical risks are identified, it is possible to know the types of defects that are most likely to occur and the types of defects that can have the greatest impact on the system. Strategies can then be developed to prevent the defects.

**Identify Critical Risks**

The first step in preventing defects is to understand the critical risks facing the project or system. The best way to do this is to identify the types of defects that pose the largest threat – defects that could jeopardize the successful construction, delivery, and operation of the system. These risks can vary widely from project to project depending on the type of system, the technology, the users of the software, etc. These risks might include:
A key requirement is missing
Critical application software does not function properly
Vendor-supplied software does not function properly
Software does not support major business functions – necessitates process re-engineering
Performance is unacceptably poor
Hardware that malfunctions
Hardware and software do not integrate properly
Hardware that is new to installation site
Users are unable or unwilling to embrace new system
User’s inability to actively participate in project, etc.

It should be emphasized that the purpose of this step is not to identify every conceivable risk, rather to identify those critical risks, which could jeopardize the success of the project and therefore merit special attention

**Estimate Expected Impact**

The “expected impact” of a risk is affected by both the probability that the risk will become a problem and the potential loss to the organization. Estimating the expected impact provides insight into the issues involved in reducing the risk. For each critical risk, an assessment can be made of the impact, in dollars, if the risk does become a problem. The probability that the risk will become a problem can also be assessed. The product of these two numbers is the expected impact. Risks should be prioritized by the expected impact and the degree to which the expected impact can be reduced.

While there will almost surely be a significant amount of guesswork in producing these numbers, precision is not important. What will be important is to identify the risk, and determine the order of magnitude of the risk.

Large, complex systems will have many critical risks and it will be important to reduce the probability of each individual critical risk becoming a problem to a very small number. In this circumstance the cumulative probability that one or more critical risks will become a problem is essentially the probability that the project will be successful. One should assume that an individual critical risk has a low probability of becoming a problem only when there is specific knowledge justifying why it is low. For example, the likelihood that an important requirement was missed may be high if users have not been very involved in the project. It may be low if the users have actively participated in the requirements definition using a good verification or validation process, and the new system is not a radical departure from an existing system or process.

One of the more effective methods for estimating the expected impact of a risk is the annual loss expectation formula. This formula states that the annual loss expectation (ALE) equals the loss per event multiplied by the number of events. For the annual calculation, the number of events should be the number of events per year. The estimated loss can be calculated by determining the average loss for a sample of loss events. For example, if the risk is that the software system will abnormally terminate,
then the average cost of correcting an abnormal termination is calculated and multiplied by the expected number of abnormal terminations associated with this risk.

The expected impact may be strongly affected not only by whether or not a risk becomes a problem, but also by how long it takes a problem to become recognized and how long it takes to be fixed, once recognized. In one reported example, a telephone company had an error in its billing system which caused it to under bill its customers by about $30 million. By law, the telephone company had to issue corrected bills within thirty days, or write off the under billing. By the time the telephone company recognized it had a problem, it was too late to collect much of the revenue.

Minimize Expected Impact

Expected impact is also affected by the action that is taken once a problem is recognized. Once Johnson & Johnson realized it had a problem with Tylenol tampering, it greatly reduced the impact of the problem by quickly notifying doctors, hospitals, distributors, retail outlets, and the public, of the problem. While the tampering itself was not related to a software defect, software systems had been developed by Johnson & Johnson to quickly respond to drug-related problems. In this case, the key to Johnson & Johnson’s successful management of the problem was how it minimized the impact of the problem, once the problem was discovered.

Minimizing expected impact involves a combination of the following three strategies:

- **Eliminate the risk.** While this is not always possible, there are situations where the best strategy will be simply to avoid the risk. For example, reducing the scope of a system, or deciding not to use the latest unproven technology, are ways to avoid certain risks.

- **Reduce the probability of a risk becoming a problem.** Most strategies will fall into this category. Inspections and testing are examples of approaches, which reduce, but do not eliminate, the probability of problems.

- **Reduce the impact if there is a problem.** In some situations, the risk cannot be eliminated, and even when the probability of a problem is low, the expected impact is high. In these cases, the best strategy may be to explore ways to reduce the impact if there is a problem. Contingency plans and disaster recovery plans would be examples of this strategy.

From a conceptual viewpoint, there are two ways to minimize the risk. These are deduced from the annual loss expectation formula. The two ways are:

- Reduce the expected loss per event
- Reduce the frequency of an event

If both of these can be reduced to zero, the risk will be eliminated. If the frequency is reduced, the probability of a risk becoming a problem is reduced. If the loss per event is reduced, the impact is reduced when the problem occurs.

There is a well-known engineering principle that says that if you have a machine with a large number of components, even if the probability that any given component will fail is small, the probability that one or more components will fail may be
unacceptably high. Because of this phenomenon, engineers are careful to estimate the mean time between failures of the machine. If the machine cannot be designed with a sufficiently large mean time between failures, the machine cannot be made. When applied to software development, this principle would say that unless the overall expected impact of the system can be made sufficiently low, do not develop the system.

Appropriate techniques to reduce expected impact are a function of the particular risk. Techniques to prevent defects include:

- **Quality Assurance** - Quality assurance techniques are designed to ensure that the processes employed are adequate to produce the desired result and that the process is being followed.

- **Training and Education (Work Force)** - It goes without saying that the better trained a work force is, the higher the quality of its work. Many defects are simply the result of workers not understanding how to do their job. Computer technology is significantly more complex today than it was just a few years ago. Moreover, the complexity will increase significantly in the coming years. Thus, it appears that the training needs at most organizations will increase sharply in the coming years.

- **Training and Education (Customers)** - Unlike the problem of training workers, more creative strategies will be required to train customers – especially when customers are not technically sophisticated. Many software vendors have recognized this problem and developed strategies to address it (more elaborate Help facilities, webinar based training with the product, etc.).

- **Methodology and Standards** - As Deming emphasizes, reducing variation is key to ensuring quality. As the nature of a process becomes understood, it evolves from art to science. At some point in this evolution, it becomes appropriate to standardize the process. This has occurred in the past with the development of standard life cycles, design approaches, etc. This is occurring today with many diverse efforts – various IEEE and ISO standards, as identified at various places in the STBOK. As the root cause of defects becomes understood, consideration should be given to developing or enhancing an organization’s methodology and standards to produce a repeatable process that prevents the defects from reoccurring.

- **Defensive Design** - While there are many variations of defensive design, the concept generally refers to designing the system so that two or more independent parts of the system must fail before a failure could occur. As technology gets more and more complicated, there should be significantly more emphasis on designing systems defensively to prevent, discover, and minimize the impact of defects. While some organizations have been doing this for years, it is a new concept to many organizations and the industry provides very little guidance on how to do it. Design techniques to improve reliability should receive more attention as the complexity of technology grows.
Defensive Code - The concept of designing a program to prevent, discover, and minimize the impact of defects is not new; it is, however, not widely practiced. Like defensive design, the concept of defensive code involves adding code to a program so that two parts of the program must fail before a major problem can occur. One form of defensive design, assertions, has been around for many years, but has received relatively little attention. An assertion is a code which tests for expected conditions and brings unexpected conditions to the attention of the programmer or users. This area also deserves to receive more attention as the complexity of technology grows.

The best defect-prevention techniques will be the ones, which reduce the expected impact the most. This, in turn, will be a function of the nature of the risks and systems within an organization. Very critical software (e.g., NASA's launch vehicle software, health care equipment) and widely distributed software (e.g., Microsoft Windows) may need to use all of the above techniques and more to adequately reduce the overall risk of highly critical software.

8.6.2.2 Deliverable Baseline

You baseline a deliverable, or work product when it reaches a predefined milestone in its development. This milestone involves transferring the product from one stage of development to the next. As a work product moves from one milestone to the next, the cost to make changes becomes much more expensive and defects in the deliverable have a much larger impact on the rest of the system. Once you baseline a deliverable, it is subject to configuration management (e.g., change control).

A defect is an instance of one or more baseline product components not satisfying their given set of requirements. Thus, errors caught before a deliverable is baselined are not considered to be defects. For example, if a programmer had responsibility for both the programming and the unit testing of a module, the program would not become baselined until after the program was unit tested. Therefore, a potential defect discovered during unit testing is not considered a defect. If, on the other hand, an organization decided to separate the coding and unit testing, it might decide to baseline the program after it was coded, but before it was unit tested. In this case, a potential defect discovered during unit testing would be considered a defect as illustrated in Figure 8-9.
Even if an organization does not formally recognize the concept of baselining deliverables, a deliverable is, for practical purposes, baselined when the person or group responsible for the deliverable passes it to the next stage of development. For example, a program specification should be considered baselined when a programmer is using it as the basis to code a program; a program should be considered baselined when it is passed on for integration testing; and a requirements specification should be considered baselined if it is being used as the basis for a technical design.

The concept of baselining is important because it requires an organization to decide both the level of formality that is appropriate and the point in the process when the formality takes effect. In general, a deliverable should be baselined when changes to the deliverable, or defects in the deliverable, can have an impact on deliverables on which other people are working.

Deliverable baseline involves the following activities:

- Identify key deliverables - Select those deliverables which will be baselined, and the point within the development process where those deliverable will be baselined.
- Define standards for each deliverable - The standards should define the requirements for each deliverable and the criteria that must be met before the deliverable can be baselined.

**8.6.2.3 Defect Discovery**

If technology cannot guarantee that defects will not be created, and this is certainly the case in software development, then the next best thing is to find defects quickly before the cost to fix them is too great. A defect is considered to have been discovered when the defect has been formally brought to the attention of the developers, and the developers acknowledge that the defect is valid. A defect has not necessarily been discovered when the user finds a problem with the software. The user must also report the defect and the developers must acknowledge that the defect is valid. There are examples where users reported problems for years before the developers of the software admitted there was a defect. Since it is important to minimize the
time between defect origination and defect discovery, strategies that not only uncover the defect, but also facilitate the reporting and developer acknowledgment of the defect can be very important.

To make it easier to recognize defects, organizations should attempt to predefine defects by category. This is a one-time event, or an event that could be performed annually. It would involve knowledgeable, respected individuals from all major areas of the IT organization. A facilitator should run the group. The objective is to identify the errors or problems that occur most frequently in the IT organization and then get agreement that they are, in fact, defects. A name should be attached to the defects. The objective of this activity is to avoid conflict when developers do not acknowledge identified defects as valid defects. For example, developers may not want to acknowledge that a missing requirement is a defect, but if it has previously been defined as a defect, the developer’s concurrence is not necessary.

The steps involved in defect discovery are illustrated in Figure 8-10 and described below.

![Figure 8-10 Defect Discovery](image)

- Find Defect - Defects are found either by pre-planned activities specifically intended to uncover defects (e.g., quality control activities such as inspections, testing, etc.) or, in effect, by accident (e.g., users in production).

Techniques to find defects can be divided into three categories:

- Static techniques – A deliverable is examined (manually or by a tool) for defects. Reviews, walkthroughs, and inspections, are examples of static techniques.
- Dynamic techniques – A deliverable is used to discover defects. Testing is an example of a dynamic technique.
- Operational techniques – An operational system produces a deliverable containing a defect found by users, customers, or control personnel, i.e., the defect is found as a result of a failure.

Research shows the following conclusions when you compare and contrast the various static, dynamic, and operational techniques. Each of the three categories of techniques is generally required for an effective defect management program. In each category, the more formally the techniques were integrated into the development process, the more effective they were.

Since static techniques will generally find defects earlier in the process, they are more efficient at finding defects. Inspection, in particular, can be very effective at removing defects. NASA and Shell Oil both use them extensively and have had impressive results. When the full inspections process was followed, Shell Oil found that for each
staff-hour spent in the inspection process, ten hours were saved! Shell Oil also found that even when groups decided not to follow the prescribed inspections process, but conducted more informal (and less effective) reviews, the results showed that the reviews saved as much time as they cost. In other words, worst case (informal reviews) – no extra cost; best case (formal inspections) – a 10-1 savings. Shell Oil found that their defect removal efficiency with inspections was 95-97% versus roughly 60% for systems that did not use inspections.

- Record Defect

Recording the defects identified at each stage of the test process is an integral part of a successful life cycle testing approach. The purpose of this activity is to create a complete record of the discrepancies identified during testing. The information captured is used in multiple ways throughout the project, and forms the basis for quality measurement.

A defect can be defined in one of two ways. From the Producer’s viewpoint, a defect is a deviation from specifications, whether missing, wrong, or extra. From the Customer’s viewpoint, a defect is anything that causes customer dissatisfaction, whether in the requirements or not; this view is known as “fit for use.” It is critical that defects identified at each stage of the project life cycle be tracked to resolution.

You should record defects for these four major purposes:

- To correct the defect
- To report status of the application
- To gather statistics used to develop defect expectations in future applications
- To improve the software development process

Most project teams utilize some type of tool to support the defect tracking process. This tool could be as simple as a white board or a table created and maintained in a word processor, or one of the more robust tools available today on the market, such as HP’s Test Director. Tools marketed for this purpose usually come with some number of customizable fields for tracking project specific data in addition to the basics. They also provide advanced features such as standard and ad hoc reporting, e-mail notification to developers and testers when a problem is assigned to them, and graphing capabilities.

At a minimum, the tool selected should support the recording and communication of all significant information about a defect. For example, a defect log could include:

- Defect ID number
- Descriptive defect name and type
- Source of defect – test case or other source
- Defect severity
- Defect priority
- Defect status (e.g., open, fixed, closed, user error, design, and so on) – more robust tools provide a status history for the defect
- Date and time tracking for either the most recent status change, or for each change in the status history
Executing the Test Process

- Detailed description, including the steps necessary to reproduce the defect
- Component or program where defect was found
- Screen prints, logs, etc., that will aid the developer in the resolution process
- Stage of origination
- Person assigned to research and correct the defect

8.6.2.3.1 Severity versus Priority

The test team, based on pre-defined severity descriptions, should assign the severity of a defect objectively. For example a “severity one” defect may be defined as one that causes data corruption, a system crash, security violations, etc. In large projects, it may also be necessary to assign a priority to the defect, which determines the order in which defects should be fixed. The priority assigned to a defect is usually more subjective based upon input from users regarding which defects are most important to them, and therefore should be fixed first.

It is recommended that severity levels be defined at the start of the project so that they are consistently assigned and understood by the team. This foresight can help test teams avoid the common disagreements with development teams about the criticality of a defect.

8.6.2.3.2 A Sample Defect-Tracking Process

After a defect is recorded it needs to be tracked. The steps below describe a simple defect tracking process. Depending on the size of the project or project team, this process may be substantially more complex.

1. Execute the test and compare the actual results to the documented expected results. If a discrepancy exists, log the discrepancy with a status of “open.” Supplementary documentation, such as screen prints or program traces, should be attached if available.
2. The Test Manager or tester should review the problem log with the appropriate member of the development team to determine if the discrepancy is truly a defect.
3. Assign the defect to a developer for correction. Once the defect is corrected, the developer will usually enter a description of the fix applied and update the defect status to “Fixed” or “Retest.”
4. The defect is routed back to the test team for retesting. Additional regression testing is performed as needed based on the severity and impact of the fix applied.
5. If the retest results match the expected results, the defect status is updated to “Closed.” If the test results indicate that the defect is still not fixed, the status is changed to “Open” and sent back to the developer.

Steps 3-5 should be repeated until the problem is resolved. Test reports are issued periodically throughout the testing process to communicate the test status to the rest of the team and management. These reports usually include a summary of the open defects, by severity. Additional graphs and metrics can also be provided to further describe the status of the application.
8.6.2.3.3  Report Defects

Once found, defects must be brought to the attention of the developers. When the defect is found by a technique specifically designed to find defects, such as those mentioned above, this is a relatively straightforward process and is almost as simple as writing a problem report. Some defects, however, are found more by accident – i.e. by people who are not trying to find defects. These may be development personnel or users. In these cases, techniques that facilitate the reporting of the defect may significantly shorten the defect discovery time. As software becomes more complex and more widely used, these techniques become more valuable. These techniques include computer forums, electronic mail, help desks, etc.

It should also be noted that there are some human factors and cultural issues involved with the defect discovery process. When a defect is initially uncovered, it may be very unclear whether it is a defect, a change, a user error, or a misunderstanding. Developers may resist calling something a defect because that implies “bad work” and may not reflect well on the development team. Users may resist calling something a “change” because that implies that the developers can charge them more money. Some organizations have skirted this issue by initially labeling everything by a different name – for example, “incidents” or “issues.” From a defect management perspective, what they are called is not an important issue. What is important is that the defect be quickly brought to the developers’ attention and formally controlled.

8.6.2.3.4  Acknowledge Defect

Once a defect has been brought to the attention of the developer, the developer must decide whether or not the defect is valid. Delays in acknowledging defects can be very costly. The primary causes of delays in acknowledging a defect appears to be an inability to reproduce the defect. When the defect is not reproducible and appears to be an isolated event (“no one else has reported anything like that”), there will be an increased tendency for the developer to assume the defect is not valid – that the defect is caused by user error or misunderstanding. Moreover, with very little information to go on, the developer may feel that there is nothing they can do anyway. Unfortunately, as technology becomes more complex, defects, which are difficult to reproduce, will become more and more common. Software developers must develop strategies to quickly pinpoint the cause of a defect.

Strategies to address this problem include:
• Instrument the code to trap the state of the environment when anomalous conditions occur.
   In the early releases of Windows, Microsoft included features to trap the state of the system when a significant problem occurred. This information was then available to Microsoft when the problem was reported and helped them analyze the problem.
• Write code to check the validity of the system.
   This is actually a very common technique for hardware manufacturers. Unfortunately, diagnostics may give a false sense of security – they can find defects, but they cannot show the absence of defects. Virus checkers would be an example of this strategy.
• Analyze reported defects to discover the cause of a defect.

While a given defect may not be reproducible, quite often it will appear again (and again) perhaps in different disguises. Eventually patterns may be noticed, which will help in resolving the defect. If the defect is not logged, or if it is closed prematurely, then valuable information can be lost. In one instance reported to the research team, a development team was having difficulty reproducing a problem. They noticed, however, that the defect was showing up at only one location. Finally, during a visit to the location they discovered how to reproduce the problem. The problem was caused when one of the users fell asleep with her finger on the enter key. In order to protect the user, the circumstances surrounding the problem were not reported to the developers until the on-site visit.

A resolution process needs to be established for use in the event there is a dispute regarding a defect. For example, if the group uncovering the defect believes it is a defect but the developers do not, a quick-resolution process must be in place. While many approaches can address this situation, the two most effective are:

• Arbitration by the software owner – the customer of the software determines whether or not the problem shall be called a defect.

• Arbitration by a software development manager – a senior manager of the software development department will be selected to resolve the dispute.

8.6.2.4 Defect Resolution

Once the developers have acknowledged that a reported defect is a valid defect, the defect resolution process begins. The steps involved in defect resolution are illustrated in Figure 8-11 and described below.

![Figure 8-11 Defect Resolution](image)

8.6.2.4.1 Prioritize Fix

The purpose of this step is to answer the following questions and initiate any immediate action that might be required:

• Is this a previously reported defect, or is it new?

• What priority should be given to fixing this defect?

• What steps should be taken to minimize the impact of the defect prior to a fix? For example, should other users be notified of the problem? Is there a work-around for the defect?
A suggested prioritization method is a three-level method, as follows:

- Critical – Would have a serious impact on the organization’s business operation.
- Major – Would cause an output of the software to be incorrect or stop.
- Minor – Something is wrong, but it does not directly affect the user of the system, such as a documentation error or cosmetic GUI (graphical user interface) error.

8.6.2.4.2 Schedule Fix

Based on the priority of the defect, the fix should be scheduled. It should be noted that some organizations treat lower priority defects as changes. All defects are not created equal from the perspective of how quickly they need to be fixed. (From a defect-prevention perspective, they may all be equal).

8.6.2.4.3 Fix Defect

This step involves correcting and verifying one or more deliverables (e.g., programs, documentation) required to remove the defect from the system. In addition, test data, checklists, etc., should be reviewed and perhaps enhanced, so that, in the future, this defect would be caught earlier.

8.6.2.4.4 Report Resolution

Once the defect has been fixed and the fix verified, appropriate developers and users need to be notified that the defect has been fixed, the nature of the fix, when the fix will be released, and how the fix will be released. As in many aspects of defect management, this is an area where automation of the process can help. Most defect management tools capture information on who found and reported the problem and therefore provides an initial list of who needs to be notified.

8.6.2.5 Process Improvement

This is perhaps the activity that is most ignored by organizations today, but offers one of the greatest areas of payback. NASA emphasizes the point that any defect represents a weakness in the process. Seemingly unimportant defects are, from a process perspective, no different than critical defects. It is only the developer’s good luck that prevents a defect from causing a major failure. Even minor defects, therefore, represent an opportunity to learn how to improve the process and prevent potentially major failures. While the defect itself may not be a big deal, the fact that there was a defect is a big deal.

This activity should include the following:

- Go back to the process where the defect originated to understand what caused the defect.
- Go back to the validation process, which should have caught the defect earlier in the process. Not only can valuable insight be gained as to how to strengthen the review process, this step serves to make everyone involved in these activities take them more seriously. This human factors dimension alone, according to some of the people the
research team interviewed, can have a very large impact on the effectiveness of the review process.

NASA takes an additional step of asking the question: If this defect could have gotten this far into the process before it was captured, what other defects may be present that have not been discovered? Thus, not only is the process strengthened to prevent defects, it is strengthened to find defects which have been created but not yet discovered. This aggressiveness should be mandatory on life-critical systems.
Measurement, Test Status, and Reporting

Management expert Peter Drucker is often quoted as saying that “you can't manage what you can't measure.” He extends that thought to “if you can’t measure it, you can’t improve it.” To accomplish both the necessary management of the test project and the continuous improvement of the test processes, it is important that the tester understand what and how to collect measures, create metrics and use that data along with other test results to develop effective test status reports. These reports should show the status of the testing based on the test plan. Reporting should document what tests have been performed and the status of those tests. Good test reporting practices are to utilize graphs, charts, and other pictorial representations when appropriate to help the other project team members and users interpret the data. The lessons learned from the test effort should be used to improve the next iteration of the test process.

### 9.1 Prerequisites to Test Reporting

From the project team and user perspective, the value of software testing is in the reports issued by the testers. The testers uncover facts, document those facts into a finding, and then report that information to project stakeholders. They may also provide opinions and recommendations under findings. The test reporting process begins with the prerequisites to collect test status data, analyze the data, and supplement the data with effective metrics.
The prerequisites to the process of reporting test results are:

- A well-defined measurement process in place
- Well-defined list of test measurements and other test status data to be collected
- Well-defined test metrics to be used in reporting test results

### 9.1.1 Define and Collect Test Status Data

Processes need to be put into place to collect the data on the status of testing that will be used in reporting test results. Before these processes are built testers need to define the data they need to collect. Four categories of data that testers most frequently collect are:

- Testing context information
- Results from verification tests
- Results from test case execution
- Defects
- Efficiency

#### 9.1.1.1 Test Context Information

This data will include but is not limited to:

- Test factors – The factors incorporated in the plan, the validation of which becomes the test objective.
- Business objectives – The validation that specific business objectives have been met.
- Interface objectives – Validation that data/objects can be correctly passed among software components.
- Functions and sub-functions – Identifiable software components normally associated with the requirements for the software.
- Units – The smallest identifiable software components.
- Platform – The hardware and software environment in which the software system will operate.

#### 9.1.1.2 Results from Verification Tests

These are the test processes used by the test team (or other project team members) to perform static testing. They include, but are not limited to:

- Inspections – A verification of process deliverables against deliverable specifications.
- Reviews – Verification that the process deliverables/phases are meeting the user’s true needs.
9.1.1.3 Results from Test Case

These are the results from dynamic test techniques used by the test team to perform testing. They include, but are not limited to:

- Functional test cases - The type of tests that will be conducted during the execution of tests, which will be based on software requirements.
- Structural test cases - The type of tests that will be conducted during the execution of test which will be based on validation of the design.
- Non-functional test cases - The type of tests that will be conducted during the execution of tests which will validate the attributes of the software such as portability, testability, maintainability, etc.

9.1.1.4 Defects

This category includes a description of the individual defects uncovered during testing. See section 8.6.2.3.

9.1.1.5 Efficiency

As the Test Plan is being developed, the testers decompose requirements into lower and lower levels. Conducting testing is normally a reverse of the test planning process. In other words, testing begins at the very lowest level and the results are rolled up to the highest level. The final Test Report determines whether the requirements were met. How well documenting, analyzing, and rolling up test results proceeds depends partially on the process of decomposing testing through to a detailed level. The roll-up is the exact reverse of the test strategy and tactics. The efficiency of these processes should be measured.

Two types of efficiency can be evaluated during testing: efficiency of the software system and efficiency of the test process. If included in the mission of software testing, the testers can measure the efficiency of both developing and operating the software system. This can involve simple metrics such as the cost to produce a function point of logic, complex metrics using measurement software.

9.1.2 Define Test Measures and Metrics used in Reporting

It is not uncommon in many test organizations for the measurement process to be weak if not non-existent. As stated at the beginning of this section, “you can't manage what you can't measure.” Regardless of whether the organization is starting with little to no measures and metrics or if a mature process is in place and maintenance of the measurement program is the objective, the tasks are the same.

1. Establish a test measurement team.
   The measurement team should include individuals who:
   - Have a working knowledge of quality and productivity measures
• Have a working understanding of benchmarking techniques
• Know the organization’s goals and objectives
• Are respected by their peers and management

The measurement team may consist of two or more individuals, depending on the size of the organization. Representatives should come from management and the project teams. For an average-size organization, the measurement team should be between three and five members.

2. Inventory existing IT measures.

The inventory of existing measures should be performed in accordance with a plan. The formal inventory is a systematic and independent review of all existing measures and metrics captured and maintained. All identified data must be checked to determine if they are valid and reliable.

3. Define a consistent set of measures.

To implement a common set of test metrics for reporting that enables senior management to quickly access the status of each project, it is critical to develop a list of consistent measures spanning all project lines. Initially, this can be challenging, but with cooperation and some negotiating, a reasonable list of measures can be drawn up. Organizations with mature processes as well as those with automated tools that collect data will have an easier time completing this step.

4. Develop desired test metrics.

The objective of this task is to use the information collected in tasks 2 and 3 to define the metrics for the test reporting process. Major criteria of this task includes:

- Description of desired output reports
- Description of common measures
- Source of common measures and associated software tools for capture
- Definition of data repositories (centralized and/or segregated)

5. Develop and implement the process for collecting measurement data.

The objective of this step is to document the process used to collect the measurement data. The implementation will involve these activities:

- Document the workflow of the data capture and reporting process
- Procure software tool(s) to capture, analyze, and report the data, if such tools are not currently available
- Develop and test system and user documentation
- Beta-test the process using a small to medium-size project
- Resolve all management and project problems
- Conduct training sessions for management and project personnel on how to interpret the reports
9.1.3 Define Effective Test Measures and Metrics

A measure is, for lack of a better word, “raw data”. For example, 100 lines of code would be a measure, or 20 severe defects would be a measure. A software metric is a number that shows a relationship between two measures. An example of a software metric might be 20% of defects are severe defects. This would be calculated from the number of severe defects found divided by the total defects found.

9.1.3.1 Objective versus Subjective Measures

Measures can be either objective or subjective. An objective measure is a measure that can be obtained by counting. For example, objective data is hard data, such as defects, hours worked, and number of completed unit tests. Subjective data are not hard numbers but are generally perceptions by a person of a product or activity. For example, a subjective measure would involve such attributes as how easy it is to use and the skill level needed to execute the system.

9.1.3.2 How Do You Know a Measure is Good?

Before a measure is approved for use, there are certain tests that it must pass. Shown here are tests that each measure and metric should be subjected to before it is approved for use:

**Reliability**

This refers to the consistency of measurement. If taken by two people, would the same results be obtained?

**Validity**

This indicates the degree to which a measure actually measures what it was intended to measure.

**Ease of Use and Simplicity**

These are functions of how easy it is to capture and use the measurement data.

**Timeliness**

This refers to whether the data was reported in sufficient time to impact the decisions needed to manage effectively.

**Calibration**

This indicates the movement of a measure so it becomes more valid, for example, changing a customer survey so it better reflects the true opinions of the customer.
9.1.3.3 Standard Units of Measure

A measure is a single attribute of an entity. It is the basic building block for a measurement program. Measurement cannot be used effectively until the standard units of measure have been defined. You cannot intelligently talk about lines of code until the measure lines of code has been defined. For example, lines of code may mean lines of code written, executable lines of code written, or even non-compound lines of code written. If a line of code was written that contained a compound statement, such as a nested IF statement two levels deep, it would be counted as two or more lines of code.

9.1.3.4 Productivity versus Quality

Quality is an attribute of a product or service. Productivity is an attribute of a process. They have frequently been called two sides of the same coin. This is because one has a significant impact on the other.

There are two ways in which quality can drive productivity. The first, and undesirable method, is to lower or not meet quality standards. For example, if one chose to eliminate the testing and rework components of a system development process, productivity as measured in lines of code per hours worked would be increased. This is sometimes done on development projects under the guise of completing projects on time. While testing and rework may not be eliminated, they are not complete when the project is placed into production. The second method for improving productivity through quality is to improve processes so that defects do not occur, thus minimizing the need for testing and rework.

9.1.3.5 Test Measure and Metric Categories

While there are no generally accepted categories of measures and metrics, it has proved helpful to many test organizations to establish categories for status and reporting purposes.

In examining many reports prepared by testers the following categories are commonly used:

- Measures unique to test
- Metrics unique to test
- Complexity measurements
- Project metrics
- Size measurements
- Satisfaction metrics
- Productivity metrics

9.1.3.5.1 Measures Unique to Test

This category includes the basic measures collected during the test process including defect measures. The following are examples of measures unique to test. Note that all measurements collected for analysis would be collected using a standardized time frame (e.g., test cycle, test
phase, sprint). Also, time is often referenced in terms of days but could be a different time factor (e.g., hour, minutes, 10ths of an hour):

- Number of test cases – The number of unique test cases selected for execution.
- Number of test cases executed – The number of unique test cases executed, not including re-execution of individual test cases.
- Number of test cases passed – The number of unique test cases that currently meet all the test criteria.
- Number of test cases failed – The number of unique test cases that currently fail to meet the test criteria.
- Number of test cases blocked – The number of distinct test cases that have not been executed during the testing effort due to an application, configuration, or environmental constraint.
- Number of test cases re-executed – The number of unique test cases that were re-executed, regardless of the number of times they were re-executed.
- Total executions – The total number of test case executions including test re-executions.
- Total number of test cases passes – The total number of test case passes, including re-executions of the same test case.
- Total failures – The total number of test case failures, including re-executions of the same test case.
- Number of first run failures – The total number of test cases that failed on the first execution.
- Number of defects found (in testing) – The number of defects uncovered in testing.
- Number of defects found by severity – The number of defects as categorized by severity (e.g., critical, high, medium, low)
- Number of defects fixed – The number of reported defects that have been corrected and the correction validated in testing.
- Number of open defects – The number of reported defects that have not been corrected or the correction has not been validated in testing.
- Number of defects found post-testing – The number of defects found after the application under test has left the test phase. Typically this would be defects found in production.
- Defect age – The number of days since the defect was reported.
- Defect aging – The number of days open (defect closed date – defect open date).
- Defect fix time retest – The number of days between the date a corrected defect is released in the new build and the date the defect is retested.
- Person days – The number of person days expended in the test effort
- Number of test cycles – The number of testing cycles required to complete testing.
- Number of requirements tested – The total number of requirements tested.
- Number of passed requirements – Number of requirements meeting success criteria.
• Number of failed requirements – Number of requirements failing to meet the defined success criteria.

9.1.3.5.2 Metrics Unique to Test

This category includes metrics that are unique to test. Most are computed from the measures listed in section 9.1.3.5.1. The metrics are (note the / represents divided by):

• Percent complete – Number of test cases passed / total number of test cases to be executed.
• Test case coverage – Number of test cases executed / total number of test cases to be executed.
• Test pass rate – Number of test cases passed / number of test cases executed.
• Test failure rate – Number of test cases failed / number of test cases executed.
• Tests blocked rate – Number of tests blocked / total test cases
• First run failure rate – Number of first run failures / number of test cases executed.
• Percent defects corrected – Number of closed defects / total number of defects reported
• Percent rework – (Number of total executions – number of test cases executed) / number of test cases executed
• Percent bad fixes – (Total failures – first run failures) / first run failures
• Defect discovery rate – Total defects found / person days of test effort
• Defect removal efficiency – Total defects found in testing / (total defects found in testing + number of defects found post-testing).
• Defect density – Total defects found / standard size measure of application under test (size measure could be KLOCs, Function points, Story Points)
• Requirements Test Coverage – Number of requirements tested / total number of requirements

9.1.3.5.3 Complexity Measurements

This category includes quantitative values accumulated by a predetermined method, which measure the complexity of a software product. The following are examples of complexity measures:

• Size of module/unit (larger module/units are considered more complex).
• Logic complexity – the number of opportunities to branch/transfer within a single module.
• Documentation complexity – the difficulty level in reading documentation usually expressed as an academic grade level.
9.1.3.5.4 Project Metrics

This category comprises the status of the project including milestones, budget and schedule variance and project scope changes. The following are examples of project metrics:

- Percent of budget utilized
- Days behind or ahead of schedule
- Percent of change of project scope
- Percent of project completed (not a budget or schedule metric, but rather an assessment of the functionality/structure completed at a given point in time)

9.1.3.5.5 Size Measurements

This category includes methods primarily developed for measuring the size of software systems, such as lines of code, and function points. These can also be used to measure software testing productivity. Sizing is important in normalizing data for comparison to other projects. The following are examples of size metrics:

- KLOC – thousand lines of code; used primarily with statement level languages.
- Function points (FP) – a defined unit of size for software.
- Pages or words of documentation

9.1.3.5.6 Satisfaction Metrics

This category includes the assessment of customers of testing on the effectiveness and efficiency of testing. The following are examples of satisfaction metrics:

- Ease of use – the amount of effort required to use software and/or software documentation.
- Customer complaints – some relationship between customer complaints and size of system or number of transactions processed.
- Customer subjective assessment – a rating system that asks customers to rate their satisfaction on different project characteristics on a scale.
- Acceptance criteria met – the number of user defined acceptance criteria met at the time software goes operational.
- User participation in software development – an indication of the user desire to produce high quality software on time and within budget.

9.1.3.5.7 Productivity Measures and Metrics

This category includes the effectiveness of test execution. Examples of productivity metrics are:

- Cost of testing in relation to overall project costs – assumes a commonly accepted ratio of the costs of development versus tests.
- Under budget/Ahead of schedule.
- Software defects uncovered after the software is placed into an operational status (measure).
9.2 Analytical Tools used to Build Test Reports

Testers use many different tools to help analyze the results of testing, and to create the information contained in the test reports. The use of these tools has proven very effective in improving the value of the reports prepared by testers for the stakeholders of the software system.

Experience has shown the analysis and reporting of defects and other software attributes is enhanced when those involved are given analysis and reporting tools. Software quality professionals have recognized the following tools as the more important analysis tools used by software testers. Some of these analytical tools are built into test automation tool packages. For each tool the deployment, or how to use, is described, as well as examples, results, and recommendations.

9.2.1 Histograms

A histogram is an orderly technique of grouping data by predetermined intervals to show the frequency of the data set. It provides a way to measure and analyze data collected about a process or problem. Pareto charts are a special use of a histogram. When sufficient data on a process is available, a histogram displays the process central point (average), variation (standard deviation, range) and shape of distribution (normal, skewed, and clustered).

Figure 9-1 illustrates a simple histogram.
9.2.1.1 Variation of a Histogram

1. Polygon: Draw line from midpoints of bars.
2. Add range of acceptable values (e.g., within plus or minus 5 of budget) to show if actual values lie within acceptable range.

9.2.1.2 Deployment

A histogram requires some understanding of the data set being measured to consolidate and condense into a meaningful display. To do this, the following steps should be taken:

1. Gather data and organize from lowest to highest values.
2. Calculate the range \( r \): largest less smallest.
3. Determine number of cells \( k \) – normally between 7 and 13.
4. Calculate the interval or width \( m \) of the cells: \( m = \frac{\text{range}}{k} \).
5. Sort the data or observations into their respective cells.
6. Count the data points of each cell (frequency) to determine the height of the interval.
7. Create a frequency table.
8. Plot the results.
9. Distribution pattern from histograms: normal, double peak, isolated island, cliff, cog-wheel, and skewed.

9.2.1.3 Results
- Helps explain graphically if a process is in or out of control - define a process.
- Provides a basis for what to work on first, especially using the Pareto chart application.
- Provides insight on the process capability to meet end user specifications.
- Establishes a technique to measure a process.
- Analyze for improvement opportunities.

9.2.1.4 Examples
- Defects by type
- Defects by source
- Delivery rates or times
- Experience or skill levels
- Cycle times
- End user survey responses

9.2.1.5 Recommendations
Everyone should use the histogram technique, especially teams that want to understand the nature of the processes they are accountable for or own.

9.2.2 Pareto Charts

The Pareto Principle is the statistical expectation that 20% of the potential causes will impact 80% of the group. A Pareto chart is a special type of bar chart used to view the causes of a problem in order of severity: largest to smallest. The Pareto chart provides an effective tool to graphically show where significant problems and causes are in a process.

A Pareto chart can be used when data is available or can be readily collected from a process. The use of this tool occurs early in the continuous improvement process when there is a need to order or rank, by frequency, problems and causes. Team(s) can focus on the vital few problems and the root causes contributing to these problems. This technique provides the ability to:
• Categorize items, usually by content or cause factors.
  o Content: type of defect, place, position, process, time, etc.
  o Cause: materials, machinery or equipment, operating methods, manpower, measurements, etc.
• Identify the causes and characteristics that most contribute to a problem.
• Decide which problem to solve or which basic causes of a problem to work on first.
• Understand the effectiveness of the improvement by doing pre- and post-improvement charts.

9.2.2.1 Deployment

A process for using Pareto charts requires a series of tasks, which fall into the following steps:

1. Define the problem clearly – Normally, this results from a team’s brainstorming sessions including using techniques such as development of affinity diagrams and cause and effect (fishbone) diagrams.
2. Collect data – The process needs a sufficient sample size over specified time, or historical data can be used if available or retrievable.
3. Sort or tally data in descending order by occurrence or frequency.
4. Construct chart – Use scales on “x” and “y” axes to correspond to data collected or sorted as shown in Figure 9-2.

![Figure 9-2 Pareto X, Y Chart](image)

5. Draw bars based on the frequency to correspond to sorted data in descending order as shown in Figure 9-3.
6. Determine vital few causes (80-20 Rule) as shown in Figure 9-4.
7. Compare and select major causes and repeat process until problem's root causes are reached sufficiently to resolve the problem.

9.2.2.2 Examples

- Problem-solving for vital few causes and characteristics.
- Defect analysis
- Cycle or delivery time reductions
- Unexpected computer processing terminations found in production
- Employee satisfaction or dissatisfaction

9.2.2.3 Results

- A necessary first step in continuous process improvement.
- Graphically demonstrates the 80-20 Rule or vital few number of items that contribute the large percentage (80%) of problems or causes.
- Provides the ability to identify which problem or cause to work on first by its severity or impact.

9.2.2.4 Recommendations

The Pareto chart is a tool that is easy to understand. To use this tool effectively requires discipline by management teams, facilitators, and teams involved with the continuous improvement process.

9.2.3 Cause and Effect Diagrams

A useful tool to visualize, clarify, link, identify, and classify possible causes of a problem is sometimes referred to as a “fishbone diagram,” or an “Ishikawa diagram,” or a “characteristics diagram.” The champion of the use of this diagram was the late Kaoru Ishikawa, a quality leader from Japan.

It is a team tool used to help identify the causes of problems related to processes, products and services. This technique keeps teams focused on a problem and potential causes. By better understanding problems within the work processes, teams can reach probable and root causes of a problem. A diagnostic approach for complex problems, this technique begins to break down root causes into manageable pieces of a process. A cause and effect diagram visualizes results of brainstorming and affinity grouping through major causes of a significant process problem. Through a series of “why-why” questions on causes, a lowest-level root cause can be discovered by this process.
9.2.3.1 Deployment

Developing a cause and effect diagram requires a series of steps:

1. Generally, as a result of a brainstorming session, identify a problem (effect) with a list of potential causes.

2. Construct a fishbone diagram with basic materials like: flip-chart, paper, tape, markers, brainstorm cards, or post-its.

3. Write the effect (problem) at the right side as shown in Figure 9-5.

4. Identify major causes of the problems, which become “big branches” as shown in Figure 9-6.

5. Use the results of brainstorming or affinity diagrams to fill “small branches” as shown in Figure 9-7.
6. Continue the process until lowest-level sub-cause is identified as shown in Figure 9-8.

7. After the team completes the fishbone diagram, review, check, and verify with the work process that these causes (factors) do strongly affect the problem or causes being resolved.

8. Select the most important causes to work on first. Many causes or root causes may need to use nominal grouping or the Pareto voting technique before reaching consensus as to what is important.

9. Verify root causes by collecting appropriate data (sampling) to validate a relationship to the problem.
10. Continue this process to identify all validated root causes.

9.2.3.2 Results

- Provides a visual relationship between cause and effect.
- Breaks down a problem into a manageable group of root causes that contribute most to a problem.
- Separates symptoms of a problem from real causes.
- Provides interaction within a team to analyze problems.

9.2.3.3 Examples

- Analysis of problems
- Source for potential process improvements
- Identify sources of defect causes
- Identify improper use of test routines and testing problems
- Identify scheduling problems and cycle times
- Confirm compliance to standards and implementation of standards
- Non-manufacturing – Display possible cause of work-related problem or conditions with the 4-Ps: policy, procedures, plant (environment and facilities), and people

9.2.3.4 Recommendation

Use to analyze problems related to the workplace or processes owned by a team.

9.2.4 Check Sheets

A check sheet is a technique or tool to record the number of occurrences over a specified interval of time; a data sample to determine the frequency of an event. The recording of data, survey, or sample is to support or validate objectively the significance of the event. This usually follows the Pareto analysis and cause and effect diagram to validate and verify a problem or cause. The team uses this technique in problem solving to support the understanding of a problem, cause, or process. This technique or tool is often used to establish a frequency or histogram chart.

9.2.4.1 Deployment

Developing a check sheet requires a series of steps:

1. Clarify what must be collected objectively.
2. Establish the format for the data collection; one easily understood by the collector.
3. Everyone involved needs to understand the objectives to ensure accuracy of the collection process.
4. Establish the sample size and time frame of data collection.
5. For consistency, instruct or train data collectors.
6. Observe, record, and collect data.
7. Tally results, using a Pareto chart or histogram.
8. Evaluate results – a team evaluation process provides better understanding and verification of data collected to support its original analysis.

9.2.4.2 Results

- Provides objective factual data to evaluate problems. Provides causes or processes early in the problem-solving process. Provides a tracking method.
- Detects patterns occurring in the process where problems are suspect. Provides data for Pareto charts or histograms.

9.2.4.3 Examples

- Project review results – defect occurrences, location, and type
- Documentation defects by type and frequency
- Cycle times – requirements to design, design to implementation
- Late deliveries
- End user complaints – all types
- Conformance to standards
- End user surveys

9.2.4.4 Recommendations

Use check sheets as a standard for problem solving whenever data is available or can be collected to validate what is happening to a process or underlining a problem early in the project.

The advantages and disadvantages are listed below:

- Advantages
  - Defines areas to discuss
  - Limits scope
  - Consistency
  - Organized approach
  - Documents results

- Disadvantages
  - Over reliance
Applicability

Limiting

The following are suggestions for preparing checklists:

- Avoid bias
- Mix questions by topic
- Test questions prior to use
- Allow for “I don't know”

The following are suggestions on using checklists:

- Learn reason for question
- Determine applicability or completeness
- Prepare or rehearse
- Anticipate response
- Ask without questionnaire
- Document when interview complete

9.2.4.5 Example Check Sheet

Figure 9-9 is an example of a check sheet.

<table>
<thead>
<tr>
<th>Unexpected Computer Processing Terminations</th>
<th>February 21-25</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Day 1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 9-9 Check Sheet Example*

9.2.5 Run Charts

A run chart is a graph of data (observation) in chronological order displaying shifts or trends in the central tendency (average). The data represents measures, counts or percentages of outputs from a process (products or services).

Run charts track changes or trends in a process as well as help to understand the dynamics of a process. This technique is often used before a control chart is
developed to monitor a process. A run chart is established for measuring or tracking events or observations in a time or sequence order.

### 9.2.5.1 Deployment

1. Decide which outputs of a process you need to measure.
2. Label your chart both vertically (quantity) and horizontally (time).
3. Plot the individual measurements over time (once per time interval or as they become available).
4. Connect data points for easy use and interpretation.
5. Follow these guidelines to interpret certain patterns when monitoring the data points:
   - Unusual events – If visual inspection looks as though a disproportionate number of data points are above or below the average then the average may have changed and must be investigated. If shift is favorable, it should be made a permanent part of the process. If unfavorable, it should be eliminated from the process.

![Example of a Run Chart](image)

**Figure 9-10  Example of a Run Chart**

- Trend – Six or more data points of continuous increase or decrease. Neither pattern would be expected to happen based on random chance. This is a strong indication that an important change has occurred and needs to be investigated.
- Two processes – If a pattern of data points alternates up and down with consistency, for example if fourteen or more data points in a row alternate up or down, this could indicate two distinct patterns caused by two groups, two shifts, or two people.
- All special causes need to be investigated to determine these patterns.
9.2.5.2 Results
- Monitor outputs of a process to better understand what is happening in the process.
- Provides a means to detect shifts or trends in a process.
- Provides input for establishing control charts after a process has matured or stabilized in time.

9.2.5.3 Examples
- Total unexpected computer processing terminations
- Complaint levels
- End user satisfaction level
- Suggestion levels
- Training efforts
- Production yields
- Number of invoices
- Number of system errors
- Down time (minutes,%)

9.2.5.4 Recommendations
Use to quantify and determine what is happening in or with a process. Several samples of run charts begin the basis for a control chart.

9.2.6 Control Charts
Control charts are a statistical technique to assess, monitor and maintain the stability of a process. The objective is to monitor a continuous repeatable process and the variation of that process from specifications. Two types of variation are being observed: 1) common, or random; and, 2) special or unique events.

Control charts are used to evaluate variation of a process to determine what improvements are needed and are meant to be used on a continuous basis to monitor processes.
A decision to use control charts is serious business and should not be taken lightly. Normally, this occurs when a process is thought to be out of control. Initially, a team evaluates what is going wrong by using brainstorming, Pareto analysis, and cause and effect diagrams to understand the problem. The listed steps detail how to build and populate a control chart:

1. Identify characteristics of a process to monitor: defects, cycle times, unexpected computer processing terminations, cost, or maintenance.
2. Select the appropriate type of control chart based on the characteristics to monitor.
3. Determine methods for sampling – how many, over what time frame; use check sheets.
4. Collect sample data.
5. Analyze and calculate sample statistics: average, standard deviation, upper limit, lower limit.
6. Construct control chart based on statistics.
7. Monitor process for common and special causes.
8. Since a process is in control when observations fall within limits, evaluate and analyze any observation outside the limits for causes related to the situation.
9. Investigate unusual patterns when observations have multiple runs above or below central line (average). A process shift is occurring and the reason needs to be understood. This may lead to process improvements or corrections.
9.2.6.2 Results
- Objectively defines a process and variation.
- Establishes measures on a process.
- Improves process analysis and opportunities.
- Process improvements are based on and managed by facts.

9.2.6.3 Examples
- Unexpected computer processing terminations in production.
- Defects by life cycle phase.
- Complaint or failures by application or software.
- Response time to change request.
- Cycle times or delivery times.
- Mean time to failure.

9.3 Reporting Test Results

Reporting test results should be a continuous process. Whenever significant problems are encountered they should be reported to the decision-makers who can determine the appropriate action. Testing reports should also be prepared at pre-defined checkpoints and at the end of testing.

In preparing test reports testers should answer these questions:
- What information do the stakeholders need?
- How can testers present that information in an easy-to-understand format?
- What can I tell the stakeholder that would help in determining what action to take?

The following aspects of test reporting are covered in this section:
- Current status test report
- Final test reports

The test reports indicating the current status of reporting, or interim test reports are needed for project management. Those responsible for making project decisions need to know the status from the tester’s perspective throughout the project. These interim reports can occur in any phase of the life cycle, at pre-defined checkpoints, or when important information needs to be conveyed to developers.

The final test reports are prepared at the conclusion of each level of testing. The ones occurring at the end of unit and integration testing are normally informal and have the primary purpose of indicating that there are no remaining defects at the end of those test levels. The test reports at the conclusion of system testing and acceptance testing are primarily for the
customer or user to make decisions regarding whether or not to place the software in operation. If it is placed in operation with known defects, the user can develop strategies to address potential weaknesses.

### 9.3.1 Current Status Test Reports

Testers need to develop reports that show the status of testing throughout the process of testing the software. The test process should produce a continuous series of reports that describe the status of testing. The current status test reports are for use by the testers, the test manager, and the software development team. The frequency of the test reports should be at the discretion of the team, and based on the extensiveness of the test process. Generally, large projects will require more interim reporting than will small test projects with a very limited test staff.

Thirteen current status reports are proposed here. Testers can use all thirteen or select specific ones to meet individual test needs. However, it is recommended that, if test data is available at the end of the testing phase, all thirteen test reports be prepared and incorporated into the final test report. Each of the thirteen reports is described in the following pages, with examples.

While today’s automated test tools can provide a plethora of test results in a variety of forms, for the purpose of discussing test reporting in this section of the STBOK a manual approach will be used to explain the general process.

#### 9.3.1.1 Function Test Matrix

A key component of any test report is the function test matrix. The function test matrix is used to determine which tests are needed, as well as their sequencing. It will also be used to determine the status of testing.

Use a spreadsheet format to maintain test results. The intersection can be color coded or coded with a number or symbol to indicate the following:

1. Test is needed, but not performed
2. Test is currently being performed
3. Test was performed and a minor defect noted
4. Test was performed and a major defect noted
5. Test complete and function is defect-free for the criteria included in this test

#### 9.3.1.1.1 Report Example

The matrix example shown in Table 9-1 uses checkmarks to indicate that a test is to be conducted on the functions indicated. The check mark can then be supplemented or replaced to show the results of testing.
9.3.1.1.2 How to Interpret the Report

The report is designed to show the results of performing a specific test on a function. A low-level report indicates the results of each test. The report is designed to show the status of each test; therefore no interpretation can be made about the results of the entire software system, only about the results of individual tests. However, if all of the tests for a specific function are successful, one could assume that function works. In this case, “working” means that it has met the criteria in the Test Plan.

9.3.1.2 Defect Status Report

A report is needed for each defect found by testers. The information collected about each defect can be as simple or as complex as desired. Section 8.6.1 described a simple Defect Naming approach.

9.3.1.2.1 Report Example

The defect status report information should be completed each time the testers uncover a defect. An example of a defect status report is shown in Table 9-2.

<table>
<thead>
<tr>
<th>Software/System Tested</th>
<th>Name of software being tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>Date on which the test occurred.</td>
</tr>
<tr>
<td>Defect Found (Name/Type)</td>
<td>The name and type of a single defect in the software being tested.</td>
</tr>
<tr>
<td>Location Found (Unit/Module)</td>
<td>The individual unit or system module in which the defect was found.</td>
</tr>
<tr>
<td>Severity of Defect</td>
<td>Critical, major, or minor. Critical means the system cannot run without correction; major means the defect will impact the accuracy of operation; minor means it will not impact the operation.</td>
</tr>
</tbody>
</table>
### Type of Defect
Whether the defect represents something missing, something wrong, or something extra.

### Test Data/Script Locating Defect
Which test was used to uncover the defect.

### Origin of Defect/Phase of Development
The phase in which the defect occurred.

### Date Corrected
The date on which the defect was corrected.

### Retest Date
The date on which the testers were scheduled to validate whether the defect had been corrected.

### Result of Retest
Whether the software system functions correctly and the defect no longer exists; or if additional correction and testing will be required. If so, the “To be added later” section will need to be reentered.

<table>
<thead>
<tr>
<th>Table 9-2 Defect Status Report</th>
</tr>
</thead>
</table>

#### 9.3.1.2.2 How to Interpret the Report

The report is designed to both describe a defect and to report the current status of that defect. The individual responsible for the function containing the defect needs to decide what action(s) to take, make any desired correction, and retest the function if changed.

The information from the defect status report can be used to produce the function/test matrix as shown in Table 9-1. In this example, the intersection between the function and test is check marked to indicate test status. Status can be color-coded or numbered 1 to 5, as described earlier.

#### 9.3.1.3 Functional Testing Status Report

The purpose of this report is to list the percent of the functions that have been fully tested; the functions that have been tested, but contain errors; and the functions that have not been tested. The report should include 100 percent of the functions to be tested in accordance with the Test Plan.

#### 9.3.1.3.1 Report Example

A sample of this Test Report is illustrated in Figure 9-12. It shows that 50 percent of the functions tested have errors, 40 percent were fully tested, and 10 percent were not tested.
9.3.1.3.2 How to Interpret the Report

The report is designed to show status to the test manager and customer of the software system. How status is interpreted will depend heavily on the point in the test process at which the report was prepared. As the implementation date approaches, a high number of functions tested with uncorrected errors and functions not tested should raise concerns about meeting the implementation date.

9.3.1.4 Functions Working Timeline

The purpose of this report is to show the status of testing and the probability that the development and test groups will have the system ready on the projected implementation date.

9.3.1.4.1 Report Example

The example of the Functions Working Timeline in Figure 9-13 shows the normal projection for having functions working. This report assumes a September implementation date and shows from January through September the percent of functions that should be working correctly at any point in time. The actual line shows that the project is doing better than anticipated.
9.3.1.4.2 How to Interpret the Report

If the actual performance is better than planned, the probability of meeting the implementation date is high. On the other hand, if the actual percent of functions working is less than planned, both the test manager and development team should be concerned, and may want to extend the implementation date or add resources to testing and/or development.

9.3.1.5 Expected versus Actual Defects Uncovered Timeline

The purpose of this report is to show whether the number of defects uncovered is above or below the expected number. This assumes that the organization has sufficient historical data to project defect rates. It also assumes that the development process is sufficiently stable so that the defect rates from that process are relatively consistent.

9.3.1.5.1 Report Example

The example chart for the Expected versus Actual Defects Uncovered Timeline in Figure 9-14 shows a project beginning in January with a September implementation date. For this project, 500 defects are expected; the expected line shows the cumulative anticipated rate for uncovering those defects. The actual line shows that a higher number of defects than expected have been uncovered early in the project.
9.3.1.5.2 How to Interpret the Report

If the actual defect rate varies from the expected rate, generally, there is a special cause, and investigation is warranted. In Figure 9-14, the cause may be because a very inexperienced project team is developing the software. Even when the actual defects are significantly less than expected, testers should be concerned, because it may mean that the tests have not been effective and therefore a large number of undetected defects remain in the software.

9.3.1.6 Defects Uncovered versus Corrected Gap Timeline

The purpose of this report is to list the backlog of detected but uncorrected defects. It requires recording defects as they are detected, and then again when they have been successfully corrected.

9.3.1.6.1 Report Example

The example in Figure 9-15 shows a project beginning in January with a projected September implementation date. One line on the chart shows the cumulative number of defects uncovered during testing, and the second line shows the cumulative number of defects corrected by the development team, which have also been retested to demonstrate that correctness. The gap represents the number of uncovered but uncorrected defects at any point in time.
9.3.1.6.2 How to Interpret the Report

The ideal project would have a very small gap between these two timelines. If the gap becomes wide, it indicates that the backlog of uncorrected defects is growing, and that the probability the project team will be able to correct them prior to implementation date is decreasing. The project team must manage this gap to ensure that it remains narrow.

9.3.1.7 Average Age of Uncorrected Defects by Type

The purpose of this report is to show the breakdown of the gap presented in Figure 9-15 by the number of days it has taken to correct defects.

9.3.1.7.1 Report Example

The Average Age of Uncorrected Defects by Type report example in Figure 9-16 shows the three severity categories aged according to the average number of days since the defect was detected. For example, it shows that the average critical defect is about 3 days old, the average major defect is about 10 days old, and the average minor defect is about 20 days old. The calculation is to accumulate the total number of days each defect has been waiting to be corrected, divided by the number of defects. Average days should be working days.
9.3.1.7.2 How to Interpret the Report

Figure 9-16 is the desirable status, demonstrating that critical defects are being corrected faster than major defects, which are being corrected faster than minor defects. Organizations should have guidelines for how long defects at each level should be maintained before being corrected. Action should be taken accordingly based on actual age.

9.3.1.8 Defect Distribution Report

The purpose of this report is to explain how defects are distributed among the modules/units being tested. It lists the total cumulative defects uncovered for each module being tested at any point in time.

9.3.1.8.1 Report Example

The Defect Distribution Report example in Figure 9-17 shows eight units under test along with the number of defects uncovered in each of those units to date. The report could be enhanced to show the extent of testing that has occurred on the modules, for example, by color-coding the number of tests; or by incorporating the number of tests into the bar as a number, such as 6 for a unit that has undergone six tests when the report was prepared.
### 9.3.1.8.2 How to Interpret the Report

This report can help identify modules that have an excessive defect rate. A variation of the report could list the cumulative defects by test. For example, defects uncovered in test 1, the cumulative defects uncovered by the end of test 2, the cumulative defects uncovered by test 3, and so forth. Frequently, modules that have abnormally high defect rates are those that have ineffective architecture, and thus are candidates for rewrite rather than additional testing.

### 9.3.1.9 Relative Defect Distribution Report

The purpose of this report is to normalize the defect distribution presented. The normalization can be by function points or lines of code. This will permit comparison of defect density among the modules/units.

#### 9.3.1.9.1 Report Example

The Normalized Defect Distribution Report illustrated in Figure 9-18 shows the same eight modules presented in Figure 9-17. However, in this example, the defect rates have been normalized to defects per 100 function points or defects per 1,000 lines of code, to enable the reader of the report to compare defect rates among the modules. This was not possible in Figure 9-17 because there was no size consideration. Again, a variation that shows the number of tests can be helpful in drawing conclusions.
9.3.1.9.2 How to Interpret the Report

This report can help identify modules that have excessive defect rates. A variation of the report could show the cumulative defects by test; for example, the defects uncovered in test 1, the cumulative defects uncovered by the end of test 2, the cumulative defects uncovered by test 3, and so forth. Frequently, modules that have abnormally high defect rates are those that have ineffective architecture, and thus are candidates for rewrite rather than additional testing.

9.3.1.10 Testing Action Report

This is a summary action report prepared by the test team. It is designed for the test manager and the software development manager. The information contained in the report should be listed as necessary to the test manager and/or the development manager to properly direct the team toward a successful implementation date.

9.3.1.10.1 Report Example

The Testing Action Report example in Figure 9-19 lists four pieces of information helpful to most test managers:

- Tests Behind Schedule – Total number of tests behind schedule, meaning either they have not been performed or contain an excessive number of defects that prevent their completion on the scheduled date.
- Uncorrected Critical Defects – The total number of critical defects not yet corrected.

Figure 9-18 Normalized Defect Distribution Report
• Major Uncorrected Defects Over 5 Days Old – The absolute number of major defects waiting more than five days to be corrected.
• Number of Uncovered Defects Not Corrected – The total number of defects awaiting correction.

<table>
<thead>
<tr>
<th>Tests Behind Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncorrected Critical Defects</td>
</tr>
<tr>
<td>Major Uncorrected Defects Over 5 Days Old</td>
</tr>
<tr>
<td>Number of Uncovered Defects Not Corrected</td>
</tr>
</tbody>
</table>

*Figure 9-19 Testing Action Report*

These items are examples of what could be included in the Testing Action Report. Most are included in the other reports, but this report is a summation, or a substitute, for the other reports.

9.3.1.10.2 How to Interpret the Report

The test manager should carefully monitor the status of testing and take action when testing falls behind schedule.

9.3.1.11 Individual Project Component Test Results

As testing is completed on each project component, the tester should issue test reports for the individual component.

9.3.1.11.1 Report Example

An Individual Projects and Interface Report is illustrated in Figure 9-20. It describes a standard for such a report, indicating it should discuss the scope of the test, the test results, what works and does not work, and recommendations.
1. **Scope of Test**
   This section indicates which functions were and were not tested.

2. **Test Results**
   This section indicates the results of testing, including any variance between what is and what should be.

3. **What Works / What Does Not Work**
   This section defines the functions that work and do not work and the interfaces that work and do not work.

4. **Recommendations**
   This section recommends actions that should be taken to:
   - Fix functions/interfaces that do not work.
   - Make additional improvements.

---

**Figure 9-20 Individual Project Component Test Report**

- **Scope of Test** – In any report on testing it is important to show the scope, otherwise, the reader will assume that exhaustive testing has occurred, which is never the case. Testing is a risk-oriented activity in which resources should be expended to minimize the major risks. Exhaustive testing is not possible, practical or economical. Thus testing is never designed to assure that there are no defects remaining in the software and the scope will explain what the testers accomplished.

- **Test Results** – This is straightforward, describing the result of the testing including any variance between the expected and actual outputs.

- **What Works/What Does Not Work** – Where detailed interim Test Reports are available, the “what works” and “what does not work” sections may merely reference those reports or attach those reports.

- **Recommendations** – This section is a critical part of the report, because the reader is usually removed from the project being tested and the technical recommendations provided by the testers can help with the reader’s business decision. For example, testers may indicate that there is a 50/50 probability that the system will terminate abnormally in production due to dating problems. With that information in hand, a business decision might be made to put the software into operation, but develop effective backup recovery procedures in case the termination occurs.
9.3.1.12 Summary Project Status Report

The Summary Project Status Report is illustrated in Figure 9-21. It provides general information about software and uses graphics to summarize the status of each project component. The design of the report and use of color enables the reader to quickly and easily access project information.

![Summary Project Status Report Example]

**Figure 9-21 Summary Project Status Report Example**

The Summary Project Status Report is divided into these four sections:

1. **Report Date Information**
   The date of the report should be on the report. The information that is contained in the report should be current as of that date.

2. **Project Information**
   Project information appears in a column on the left side of the report. Each project has its own “cell” where information about the project appears. Each cell contains the official project name, the name of the project manager, the phase of the project (e.g., planning, requirements, development, and implementation) and the name of the executive sponsor.

3. **Timeline Information**
   Timeline information appears in a chart that displays project status over a 20-month period. It shows project status by measuring technical, budgeting, and scheduling
considerations. The year and month (abbreviated with initials) appear along the top of the chart to indicate the month-by-month status of each project.

Technical (T), Scheduling (S), and Budget (B) information also appears in the chart, and is specific to each project. These three considerations measure the status of each project:

- Technical status (T) shows the degree to which the project is expected to function within the defined technical and/or business requirements.
- Scheduling status (S) shows the degree to which the project is adhering to the current approved schedule.
- Budgeting status (B) shows the degree to which the project is adhering to the current approved budget. Expenditures for the budget include funds, human resources, and other resources.

4. Legend Information

The report legend, which is located along the bottom of the page, defines the colors and symbols used in the report, including category and color codes. The following colors could be used to help to quickly identify project status:

- A green circle could mean there are no major problems and that the project is expected to remain on schedule.
- A yellow circle could indicate potentially serious deviation from project progression.
- A red circle could mean a serious problem has occurred and will have a negative effect on project progression.

9.3.1.13 Individual Project Status Report

The Individual Project Status Report as illustrated in Figure 9-22 provides information related to a specific project component. The design of the report enables the reader to quickly and easily access project information.

9.3.1.13.1 Project Information

The project information normally includes the following six sections:

1. Name of the report
2. Date the report is issued
3. Name of the executive sponsoring the project
4. Name of the project manager
5. General project information
6. Quick-status box containing a color-coded rectangle indicating the overall status of the project
9.3.1.13.2 General Project Information

This section of the report contains general information about the project. It should include the work request number; a brief description of the project; and show the phase of the project (e.g., planning, requirements, development, and implementation), as well as important project dates and figures, which include:

- Project start date, determined by official approval, sponsorship, and project management
- Original target date for project completion
- Current target date for project completion
- Phase start date of the current phase
- Original target date for completion of the current phase
- Current target date for completion of the current phase
- Original budget allotted for the project
- Current budget allotted for the project
- Expenses to date for the project

9.3.1.13.3 Project Activities Information

The Project Activities section of the report gives a history of the project over an extended period. The Project Activities chart measures the status according to the phase of the project. The project phases used might be:

- Planning
- Requirements
- Development
- Implementation

Comments may be added below each phase to track specific project developments or occurrences. A timeline should be included in the chart to measure each phase of the project. Color-coded circles could indicate the status of each phase.

Future activities for the project should be indicated showing the expected date of project completion, or the current target date.
Figure 9-22  Example of an Individual Project Status Report

9.3.1.13.4 Essential Elements Information

The Essential Elements section of the report also contains a chart. It measures the status of the project by comparing it to the previous status of the project. The chart could use the color-
coded circles and list considerations that allow the reader to quickly gather project statistics. These considerations ask:

- Is the project on schedule?
- Do the current project results meet the performance requirements?
- Are the project costs within the projected budget?
- Is the project cost over-budget?
- What is the dollar amount of the project budget overrun?

These questions can be answered by comparing the previous report results to the current report results.

This section of the report also includes a graph that compares projected costs to actual costs. The projected cost line can appear in one color; the actual cost line appears in another color. This graph shows you whether the project is adhering to the current approved budget.

9.3.1.13.5 Legend Information

The report legend, which should be located along the bottom of the page, defines the colors and symbols used in the report, including category and color codes. The following symbols can be used to help to quickly identify project status:

- The ● indicates there are no major problems and that the project is expected to remain on schedule.
- The ♦ means there is a potentially serious deviation from project progression.
- The ■ indicates a serious problem has occurred and will have a negative effect on project progression.

9.3.1.13.6 Project Highlights Information

The project highlights should appear at the bottom of the report. This section may also contain comments explaining specific project developments or occurrences that affect progression.

9.3.2 Final Test Reports

Test reports should be prepared at the conclusion of each level of testing. This might include:

- Unit Test Report
- Integration Test Report
- System Test Report
- Acceptance Test Report

The test reports are designed to report the results of testing as defined in the Test Plan. Without a well-developed Test Process, which has been executed in accordance with the plan, it is difficult to develop a meaningful test report.
All final test reports should be designed to accomplish the following three objectives:

- Define the scope of testing – this is normally a brief recap of the Test Plan
- Present the results of testing
- Draw conclusions and recommendations from those test results

The final test report may be a combination of electronic data and printed information. For example, if the Function Test Matrix is maintained electronically, there is no reason to print that, as the detail is available electronically if needed. The printed final report will summarize that data, draw the appropriate conclusions, and present recommendations.

The final test report has the following objectives:

- Inform the developers what works and what does not work.
- Provide information to the users of the software system so that they can determine whether the system is ready for production; and if so, to assess the potential consequences and initiate appropriate actions to minimize those consequences.
- After implementation, help the project trace problems in the event the application malfunctions in production. Knowing which functions have been correctly tested and which ones still contain defects can assist in taking corrective action.
- Use the test results to analyze the test process for the purpose of preventing similar defects from occurring in the future. Accumulating the results of many test reports to identify which components of the software development process are defect-prone provides this historical data, improves the developmental process and, if improved, could eliminate or minimize the occurrence of high-frequency defects.

### 9.3.2.1 Description of Test Reports

There is no generally accepted standard regarding the type, content and frequency of test reports. However, it is reasonable to assume that some type of report should be issued after the conclusion of each test activity. This would include reports at the conclusion of these test activities:

- Unit test
- Integration test
- System test

The individual who wrote the unit normally conducts unit testing. The objective is to assure all the functions in the unit perform correctly, and the unit structure performs correctly. The report should focus on what was tested, the test results, defects uncovered and, what defects have not been corrected, plus the unit tester’s recommendations as to what should be done prior to integration testing.

### 9.3.2.2 Integration Test Report

Integration testing tests the interfaces between individual projects or units. A good Test Plan will identify the interfaces and institute test conditions that will validate interfaces. Given this,
the integration report follows the same format as the Unit Test Report, except that the conditions tested are the interfaces.

9.3.2.3 System Test Report

Skill Category 5, Test Planning, presented a system test plan standard that identified the objectives of testing, what was to be tested, how it was to be tested, and when tests should occur. The System Test Report should present the results of executing that Test Plan. Figure 9-23 illustrates the test reporting standard that is based on the test plan standard.

1. General Information
   1.1 Summary. Summarize both the general functions of the software tested and the test analysis performed.
   1.2 Environment. Identify the software sponsor, developer, user organization, and the computer center where the software is to be installed. Assess the manner in which the test environment may be different from the operation environment, and the effects of this difference on the tests.
   1.3 References. List applicable references, such as:
      1.3.1 Project request (authorization).
      1.3.2 Previously published documents on the project.
      1.3.3 Documentation concerning related projects.
      1.3.4 FIPS publications and other reference documents.

2. Test Results and Findings
   Identify and present the results and findings of each test separately in paragraphs 2.1 through 2.n.
   2.1 Test (identifier).
      2.1.1 Validation tests. Compare the data input and output results, including the output of internally generated data, with this test with the data input and output requirements. State the findings.
      2.1.2 Verification tests. Compare what is shown on the document to what should be shown.
   2.n Test (identifier). Present the results and findings of the second and succeeding tests in a manner similar to that of paragraph 2.1.

3. Software Function Findings
   Identify and describe the findings on each function separately in paragraphs 3.1 through 3.n.
   3.1 Software Function.
      3.1.1 Performance. Describe briefly the function. Describe the software capabilities designed to satisfy this function. State the findings as to the demonstrated capabilities from one or more tests.
      3.1.2 Limits. Describe the range of data values tested. Identify the deficiencies, limitations, and constraints detected in the software during the testing with respect to this function.
   3.n. Function (identifier). Present the findings on the second and succeeding functions in a manner similar to that of paragraph 3.1.

Figure 9-23 System Test Report Standard Example
9.3.3 Guidelines for Report Writing

The following two guidelines are provided for writing and using the report information:

1. Develop a baseline.
   The data extracted from individual project reports can be used to develop a baseline for
   the enterprise based on mean scores of the reporting criteria. Rather than comparing
   quality, productivity, budget, defects, or other categories of metrics to external
   organizations, valuable management information can be made available. From this
   baseline, individual projects can be compared. Information from projects consistently
   scoring above the enterprise baseline can be used to improve those projects that are
   marginal or fall below the enterprise baseline.

2. Use good report writing practices. The following are examples of good report writing:
   • Allow project team members to review the draft and make comments before the report
     is finalized.
   • Don’t include names or assign blame.
   • Stress quality.
   • Limit the report to two or three pages stressing important items; include other
     information in appendices and schedules.
   • Eliminate small problems from the report and give these directly to the project people.
   • Hand-carry the report to the project leader.
   • Offer to have the testers work with the project team to explain their findings and
     recommendations.
Testing Specialized Technologies

The skill sets required by today’s software test professional in many ways mirror Moore’s Law. Paraphrasing just a bit, Moore's law states that advances in technology will double approximately every 18 to 24 months. While it is true that some testers on legacy projects still test applications where the origin of the COBOL code may, in fact, stretch back to 1965 when Gordon Moore, co-founder of INTEL, coined Moore’s Law, the reality is the skill sets needed today are advancing rapidly and have become more and more specialized.

To be clear, calling something specialized does not mean that the discussions about life cycles, test preparation, planning, test techniques, measurement, managing the project or leading the team are different. Quite the contrary, regardless of the technology the majority of the skills and tasks performed are applicable with the customization that any project might require. What this section deals with is the added nuances that certain technologies require for testing. Discussed here will be such things as testing web and mobile applications, testing cloud based applications, Agile, security, and Dev Ops. For these the nature of the technology and its impact on testing will be described.

<table>
<thead>
<tr>
<th>New Technology</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web-Based Applications</td>
<td>10-7</td>
</tr>
<tr>
<td>Mobile Application Testing</td>
<td>10-15</td>
</tr>
<tr>
<td>Cloud Computing</td>
<td>10-23</td>
</tr>
<tr>
<td>Testing in an Agile Environment</td>
<td>10-29</td>
</tr>
<tr>
<td>DevOps</td>
<td>10-32</td>
</tr>
<tr>
<td>The Internet of Things</td>
<td>10-34</td>
</tr>
</tbody>
</table>
10.1 New Technology

Identifying something as a specialized technology does not presuppose that it is a new technology. However, it is true that when we think of specialized technologies we often think of “newer” types of application development methodologies or newer hardware and software platforms. With that in mind we will begin this skill category with a discussion around the impact of new technologies on the organization and the test professional.

As organizations acquire new technologies, new skills are required because test plans need to be based on the types of technology used. Also technologies new to the organization and the testers pose technological risks which must be addressed in test planning and test execution. It is important to keep in mind that any technology new to the testers or the organization, whether it is “technologically new” or not should be considered a new technology for the purpose of risk analysis and subsequent test planning.

10.1.1 Risks Associated with New Technology

Testers need to answer the following questions when testing a new project:

- Is new technology utilized on the project being tested?
- If so, what are the concerns and risks associated with using that technology?
- If significant risks exist how will the testing process address those risks?

The following are the more common risks associated with the use of technology new to an IT organization. Note that this list is not meant to be comprehensive but rather representative of the types of risks frequently associated with using new technology:

- Unproven technology
  The technology is available, but there is not enough experience with the use of that technology to determine whether or not the stated benefits for using that technology can actually be received.

- Technology incompatible with other implemented technologies
  The technologies currently in place in the IT organization are usually incompatible with the new technology acquired. Therefore, the new technology may meet all of its stated benefits but the technology cannot be used because of incompatibility with currently implemented technologies.

- New technology obsoletes existing implemented technologies
  Many times when vendors develop new technologies, such as a new version of software, they discontinue support of the existing software version. Thus, the acquisition of new technology involves deleting the existing technologies and replacing it with the new. Sometimes vendors do not declare the current technologies obsolete until there has been general acceptance of the new technology. If testers do not assume that older technologies will become obsolete they may fail to address the significant new technology risk.
• Variance between documentation and technology execution

The documentation (e.g., manuals, instructions) associated with using new technologies may differ from the actual performance of the technologies. Thus, when organizations attempt to use the new technologies with the documented procedures the new technologies will fail to perform as specified.

• Staff not competent to use new technology

Training and deployment processes may be needed to assure the organization has adequate competency to use the new technology effectively and efficiently. If the organization’s staff does not possess the necessary skill sets, they will not gain the benefits attributable to the new technology.

• Lack of understanding how to optimize the new technology

Studies show that most organizations use only limited aspects of new technology. They do not take the time and effort to learn the technology well enough to optimize the use of the technology. Therefore, while some benefits may be received, the organization may miss some of the major benefits associated with using a new technology.

• Technology not incorporated into the organization’s work processes

This is typical implementation of new technologies at technology maturity Level 1. At this level, management cannot control how the new technology will be used in the IT organization. Because staff has the decision over whether or not to use technology and how to use it, some significant benefits associated with that technology may be lost.

• Obsolete testing tools

The implementation of new technology may obsolete the use of existing testing tools. New technologies may require new testing methods and tools.

• Inadequate vendor support

The IT staff may need assistance in using and testing the technology, but are unable to obtain that assistance from the vendor.

10.1.2 Testing the Effectiveness of Integrating New Technology

The mission assigned to software testers will determine whether or not testers need to assess the impact of new technologies on their software testing roles and responsibilities. That responsibility can be assigned to software testers, software developers, or process engineering and quality assurance groups. If the responsibility is assigned to testers they need to develop the competencies to fulfill that responsibility.

This skill category has been included in the STBOK because it is a testing activity performed by many software testing groups. It also represents an opportunity for software testers to provide more value to their IT organizations.
10.1.2.1 Test the Controls over Implementing the New Technology

Testing the adequacy of the controls over the new technology to evaluate the effectiveness of the implementation involves three tasks:

- Testing actual performance versus stated performance
  Does the technology accomplish its stated benefits?
- Testing the adequacy of the current processes to control the technology
  Do implemented work processes assure the technology will be used in an effective and efficient manner?
- Assessing whether staff skills are adequate to effectively use the technology
  Are the users of the technology sufficiently trained to use the technology effectively?

10.1.2.1.1 Test Actual Performance versus Stated Performance

Software systems are built based on assumptions. For example, some assumptions are that:

- hardware technology can perform at predefined speeds
- the staff assigned to a project can do the defined tasks
- if a specific work process is followed it will produce a predefined desired result

With currently implemented technology there is experience to support these assumptions. For example, if a specific individual is to use a specific development or testing tool, and those who have graduated from a specified training class can perform tasks using those tools, it is reasonable to assume that if project personnel had that training they could perform the defined work tasks.

When new technologies are introduced there are no experiences to support project assumptions. For example, if a software package was claimed to be able to search a database and retrieve desired information within X% of a second, that assumption would have to be tested. If the software failed to meet the performance specifications, a software system objective might not be met. For example, if a system objective was that a customer using the organization’s web portal to access records could determine the availability of a specific product within two seconds, and the hardware and software used to make that determination took 15 seconds the customer might leave the Web site and the sale would be lost.

The following is a list of some of the tests that testers may want to perform to evaluate whether or not a new technology can or cannot meet the specified performance criteria:

- Documentation represents actual technology execution.
- Training courses transfer the needed knowledge to use the technology.
- New technology is compatible with existing technology.
- Stated performance criteria represent actual performance criteria.
- Promised vendor support equals actual vendor support.
- Expected test processes and tools are effective in testing new technologies.
10.1.2.1.2 Test the Adequacy of the Current Processes to Control the Technology

The underlying premise of process engineering is that the method the IT organization performs to do work will be controlled. Work processes are the means for controlling the use of new technologies. Work processes include these three process attributes:

- **Standard**
  
  This is the objective to be achieved using new technology.

- **Procedure**
  
  The step-by-step methods that will be performed to utilize the new technology.

- **Quality Control**
  
  The means that will be used to determine that the procedures to do work are performed in the manner specified in those procedures.

If new technologies are to be used effectively and efficiently the work processes governing the use of those technologies must include the standard, the procedures and the quality control procedures. Without a standard (objective), the workers do not know the purpose for using the technology, without procedures they do not know how to use the technology, without quality control they do not know if they used the procedures correctly.

Any new technology that is deemed “out of control” requires additional attention. Out of control means that any or all of the key process engineering components are not present in the use of that technology. Out of control from a test perspective means that testers cannot make assumptions that work will be performed as specified, and the software will execute as specified.

If the testers determine that processes are not adequate to control new technology, they can take any one of the following three actions:

- Identify the new technology risk and report that risk to the appropriate software system stakeholder.
- Identify the potentially ineffective parts of the work process as related to that new technology. This might include determining the technology maturity level and/or any of the missing key process engineering components.
- Conduct tests to identify specific problems associated with the new technology and assess the potential impact of those problems on the operational system.

10.1.3 Test the Adequacy of Staff Skills to Use the Technology

There are two components to any professional process. The components are the process itself, and the competency of the process user to use that process effectively. The two components must be properly coordinated for the new technology governed by that process to be used effectively.
Relationship of Competency to Process shows the relationship of the competency of the user technology to the process controlling the use of that technology. The figure shows the five levels of technology process maturity, and the competency of the user to use the technology from high to low.

The key concept of Relationship of Competency to Process is: do you rely more on the competency of the user or the effectiveness of the process when using that technology. When the technology process maturity is low there is greater reliance on the competency of the user. When the technology process maturity is high there is more reliance on the process effectiveness and less reliance on the user competency.

![Figure 10-1 Relationship of Competency to Process](image)

Figure 10-1  Relationship of Competency to Process

The reason for the changing reliance between user and process is transfer of knowledge. At technological process maturity Level 1 most of the ability to use new technology effectively resides with the user. As the user(s) learns to use the technology more effectively there is a knowledge transfer from the user to the process. For example, if the user of an agile software development process learned that agile technology was only effective with small software systems, that knowledge could be incorporated into the agile software development process. Prior to using the process the user would assess the size of the system and, if it was large, use another software development methodology to build the software.

In assessing the adequacy of the staff competency to use new technology the tester should evaluate:

- Technological process maturity level
  - At lower levels more user competency is needed than at higher levels.
- Training available in the new technology
Testers need to determine the type of training available, the effectiveness of that training, and whether or not the users of the new technology have gone through that training.

- The performance evaluation of individuals to use the new technology effectively
  This looks at the individual’s past experience while working with that technology on projects and then assessing the effectiveness of the projects in using that technology.

## 10.2 Web-Based Applications

Web-based applications are those applications that use the Internet, intranets, and extranets. The Internet is a worldwide collection of interconnected networks. An intranet is a private network inside a company using web-based applications, but for use only within an organization. An extranet is an intranet that can be partially accessed by authorized outside users, enabling businesses to exchange information over the Internet securely.

### 10.2.1 Understand the Basic Architecture

One of the first tasks for the test professional when working on a web-based application project is to understand the basic architecture. Starting with the client’s access point, the web browsers reside on the client’s system and are networked to a web server, either through a remote connection or through a network such as a local area network (LAN) or wide area network (WAN). As the web server receives and processes requests from the user’s browser, requests may be sent to the application server to perform actions such as database queries or electronic commerce transactions. The application server may then send requests to the database servers or back-end processing systems. See Figure 10-2.
There are many variations within the web system architecture, but for illustration purposes the above diagram is representative.

### 10.2.1.1 Thin-Client versus Thick-Client Applications

The notion of thin-client and thick-client processing has been around since ancient times, say 20 to 30 years ago. In the olden days thin-client typically referred to a “dumb” terminal where a CRT and keyboard served as the user interface and all program execution happened on a remote system (e.g., mainframe computer). More recently, the term thin-client is used to refer to the relationship between the browser executing code with the majority of execution taking place on a web server. When the majority of processing is executed on the server-side, a system is considered to be a thin-client system. When the majority of processing is executed on the client-side, a system is considered to be a thick-client system.

In a thin-client system, the user interface runs on the client host while all other components run on the server host(s). The server is responsible for all services. After retrieving and processing data, only a plain HTML page is sent back to the client.

By contrast, in a thick-client system, most processing is done on the client-side; the client application handles data processing and applies logic rules to data. The server is responsible only for providing data access features and data storage. Components such as ActiveX controls and Java applets, which are required for the client to process data, are hosted and executed on the client machine.

Each of these systems calls for a different testing strategy. In thick-client systems, testing should focus on performance and compatibility. If Java applets are used, the applets will be sent to the browser with each request, unless the same applet is used within the same instance of the browser.
Compatibility issues in thin-client systems are less of a concern. Performance issues do, however, need to be considered on the server-side, where requests are processed, and on the network where data transfer takes place (for example, sending bitmaps to the browser). The thin-client system is designed to solve incompatibility problems as well as processing-power limitations on the client-side. Additionally, thin-client systems ensure that updates happen immediately, because the updates are applied at that server side only.

### 10.2.2 Test Related Concerns

Testers should have the following concerns when conducting web-based testing:

- **Browser compatibility** – Testers should validate consistent application performance on a variety of browser types and configurations.

- **Functional correctness** – Testers should validate that the application functions correctly. This includes validating links, calculations, displays of information and navigation. See section 10.2.2 for additional details.

- **Integration** – Testers should validate the integration between browsers and servers, applications and data, and hardware and software.

- **Usability** – Testers should validate the overall usability of a web page or a web application, including appearance, clarity, and navigation.

- **Accessibility** – Testers should validate that people with disabilities can perceive, understand, navigate, and interact with the web-based application under test. (Section 508 of the United States Rehabilitation Act requires that all United States Federal Departments and Agencies ensure that all Web site content be equally accessible to people with disabilities. This applies to Web applications, Web pages, and all attached files. It applies to intranet as well as public-facing Web pages.)

- **Security** – Testers should validate the adequacy and correctness of security controls, including access control and authorizations.

- **Stress/Load/Performance** – Testers should validate the performance of the web applications under load.

### 10.2.3 Planning for Web-based Application Testing

Planning for web-based application testing should follow the same basic processes as described in Skill Category 5. In this section certain unique issues regarding web test planning will be discussed.
10.2.3.1 Risk Based Approach

The very nature of web-based applications and the architecture involved create higher risks to the organization deploying the application. The simple “risk” calculation is Risk = Likelihood of failure times Cost of Failure. Web applications often impact both the multiplicand and the multiplier in the equation.

Web applications often see a higher level of use than traditional applications. Web apps are often customer facing and can see traffic in the thousands if not millions of hits per day. With the higher level of use comes a higher likelihood of failure.

Whenever products or services are customer facing, the cost of failure grows exponentially. Many companies have been defined by the failures of their web applications. The roll-out in 2013 of the US Government’s Affordable Care Act website was so catastrophic that it tainted the entire presidential administration.

It is critical when planning for testing that detailed analysis of the application system be done so test resources can be portioned appropriately to minimize the risks inherent in the deployment of a web application.

10.2.3.2 Client Behavior

For testing an existing website, test planning should include the use of:

Web Analytics – A great tool for planning web testing is the use of analytics. Web analytics can help the testers understand the patterns of use on the application site. Analytics can provide measures and metrics such as page views, exit rates, time on site, and visits.

Browser Usage – One of the challenges of web testing is “uncontrolled user interfaces” also known as browsers. Web site tools allow the tester to understand what browsers and what versions of the browsers are being used to access the web application and what types of mobile user platforms are accessing the site. By analyzing the current patterns the tester can set up the test environment to test on the different browsers and mobile devices identified. Note this would now be testing the web application on a mobile device. Mobile application testing is discussed in section 10.3.

Behavior Map – Understanding the behavior of visitors to a website helps the tester prioritize test cases to exercise both the most frequently accessed portions of an application and also test the most usual user movement. Heat maps are tools that essentially overlay a website and track the user’s interaction. Heat maps track interactions such as clicks, scrolls, and mouse movements. These tools provide a detailed picture on how the user moves around a site which can help the testers better plan the test procedures.

10.2.3.3 Identify Page Objects

During the planning phase and more specifically when planning for test automation the tester should identify all the various page objects and response items such as:
• Entry fields, buttons, radio buttons, checkboxes, dropdown list boxes, images, links, etc.
• Responses that may be in tables, spans, divs, lis, etc.

10.2.4 Identify Test Conditions

In Skill Category 7, section 7.1, considerable attention was given to identifying testable conditions as a precursor to writing test cases. Web application testing would follow that same process for identifying testable conditions. Shown here are some additional conditions applicable to web testing:

10.2.4.1 Browser Differences

Browser differences can make a web application appear and/or act differently to different people. The list given here is not intended to be exhaustive but rather is a sample.

Visual page display – Web applications do not display the same way across all browsers. Test as many browsers as possible to ensure that the application under test (AUT) works as intended.

Print handling – To make printing faster and easier, some pages add a link or button to print a browser-friendly version of the page being viewed.

Reload – Some browser configurations will not automatically display updated pages if a version of the page still exists in the cache. Some pages indicate if the user should reload the page.

Navigation – Browsers vary in the ease of navigation, especially when it comes to visiting pages previously visited during a session.

Graphics filters – Browsers may handle images differently, depending on the graphic files supported by the browser. In fact, some browsers may not show an image at all.

Caching – How the cache is configured will have an impact on the performance of a browser to display information.

Scripts – Browsers may handle scripts (e.g., Flash or Ajax page loads) differently. It is important to understand the script compatibility across browsers and as necessary measure the load times to help optimize performance. If scripts are only compatible with certain browsers, test to ensure that they degrade gracefully on others so that all users get the best possible experience.

Dynamic application content – Browsers react differently to web generated external applications data feeds to applications like MS-Excel, MS-Word or Adobe PDFs.
**Dynamic page generation** – This includes how a user receives information from pages that change based on input. Examples of dynamic page generation include:
- Shopping card applications
- Data search applications
- Calculated forms

**File uploads and downloads** – Movement of data to and from remote data storage

**Email functions** – Dynamic calls to email functionality will differ from browser to browser and between native email programs.

### 10.2.4.2 Functionality Conditions to Test

Web applications, as with any application, need to work accurately, quickly, and consistently. The web application tester must ensure that the product will deliver the results the user intends. Some of the functional elements unique to web application testing are detailed in the following sections:

#### 10.2.4.2.1 Forms

The submission of forms is a key function on many websites. Whether the form is request for information or a feedback survey, the testers must ensure that all field inputs are validated and connections to back-end database systems store data correctly.

#### 10.2.4.2.2 Media Components

Test to ensure that audio and video playback, animations and interactive media work correctly. These components should function as expected and not break or slow down the rest of the app while loading or running.

#### 10.2.4.2.3 Search

A common function in web applications is the ability to search through content, files or documentation. Tests are needed to ensure that the search engine comprehensively indexes this information, updates itself regularly and are quick to look up and display relevant results.

#### 10.2.4.2.4 Role Based Access

Web applications frequently have different functionality available to different access groups. Test to ensure that each group accesses the correct functions.

#### 10.2.4.3 Usability Tests

Web applications are integral parts to most people’s daily activities. Whether it is checking banking information or ordering a book online, ensuring that web apps are easy to use is critical. The web application should provide a quality front-end experience to all users. Some
of the conditions for consideration in usability testing include those listed in the following sections.

10.2.4.3.1 Navigation

All links to and from the homepage should be prominent and point to the right destination pages. A standard approach for testing forward and backward and other links should be defined and used.

10.2.4.3.2 Accessibility

As discussed in section 10.2.2, testers must ensure that the application under test is easy to use even for those with disabilities or impairments of vision or motor functions.

10.2.4.3.3 Error Messages and Warnings

Like any application, the web app will invariably respond incorrectly at some point. As with any error routines, the AUT should trap the error, display a descriptive and helpful message to the user, and then return control to the user in such a fashion that the application continues to operate and preserves the user’s data.

10.2.4.3.4 Help and Documentation

Central to usability is the ability of user to use the system. Not all users will be equally comfortable using a web application and may need assistance the first few times. Even experienced users will question what to do at times and require assistance on specific items. Testers should test the documentation and/or support channels to ensure they are easily found and accessible from any module or page.

10.2.4.3.5 Adherence to De facto Standards

Standards for web design have had mixed success. Regardless of the lack of a recognized standard there certainly are some de facto standards which should be followed. This allows the web user the ability to move from website to website without re-learning the page layout styles. An easy example is the underlining or color change of a word(s) indicating a hyperlink to another page. Other de facto standards include: site log in, site log out, contact us, and help links/buttons being located in the top right corner of a web pages. Such standards help to alleviate the “hunting” necessary to find a commonly used option on various websites.

10.2.4.3.6 Layouts

Consistency across the web application for such items as central workspace, menu location and functionality, animations, interactions (such as drag-and-drop features and modal windows), fonts and colors is important.
10.2.4.4 Security Tests

Many web applications take input from users and store that data on a remote system (e.g., database server). It is critically important to validate that the application and data are protected from outside intrusion or unauthorized access. Testers must validate that these vulnerabilities do not exist in the AUT. Some of the common security issues are described here.

10.2.4.4.1 SQL Injection

A common attack pattern is for a hacker, through a user input vulnerability, to execute a SQL command on the app’s database, leading to damage or theft of user data. These generally occur due to the improper neutralization of special elements used in SQL commands or OS commands.

10.2.4.4.2 Cross-Site Scripting (XSS)

XSS is a type of computer security vulnerability which enables attackers to inject client-side script into web pages viewed by other users. This allows an attacker to send malicious content from an end-user and collect some type of data from a victim.

10.2.4.4.3 Secure Socket Layer (SSL)

SSL Certificates are small data files that digitally bind a cryptographic key to an organization’s details. When installed on a web server, it activates a padlock and the https protocol allowing secure connections from a web server to a browser. Testers must ensure that HTTPS is used where required for secure transaction control.

10.2.5 Web Testing Tools

In Skill Category 2, section 2.4.1, the notion of generalized tools, tools that can work across the life cycle and are technology independent, and specialized tools, tools that have a very specific function focused on a specialized niche were discussed. There are many specialized tools available for testing in the web-based app sphere. Shown below are lists of tools by different categories. The International Software Certifications Board (ISCB) does not endorse any tools. There are simply several examples.

10.2.5.1 Browser Add-ons

- Fire Fox browser add-ons: Firebug and FireCookie, FormSaver, Xpather
- IE browser add-ons: IECookiesView, Fiddler, Webcollect, Web Accessibility Toolbar, QF-Test, and Mathon
10.2.5.2 Automate Tools

Examples of automated test tools for web testing include Selenium, Sahi, HP QA Inspect, Test Studio, SOAtest, and WATiR (Web App Testing in Ruby).

10.3 Mobile Application Testing

Seven (7) billion is the estimated number of mobile subscriptions worldwide in 2014 with 4.5 billion unique mobile users (some users have multiple subscriptions). Approximately 85 percent of phone connectivity is over mobile devices (as compared to land lines). And the numbers grow daily!

Simply stated, mobile apps are the computer programs designed to run on those 7 billion smartphones, tablet computers and other mobile devices. Many mobile apps are pre-installed by the mobile device manufacturer while over a million apps are delivered through application distribution platforms such as the Apple App Store (1.3 million apps as of June 2014), Google Play Store (1.3 million apps as of September 2014), Windows Phone Store (300,000 apps as of August 2014), BlackBerry App World (236,601 apps as of October 2013) and Amazon App Store (250,908 apps as of July 2014). Mobile apps are also written and distributed within the closed environments of organizations.

Mobile applications can be classified into various domain areas which include: information, social media, games, retail, banking, travel, e-readers, telephony, and professional (e.g., financial, medical, and engineering) to name a few. The risks associated with the different domains vary and the resources expended on testing should correlate directly.
10.3.1 Characteristics and Challenges of the Mobile Platform

The Mobile platform presents the tester with some very unique circumstances which in turn creates some interesting test challenges. While testing an application has many similarities regardless of the delivery platform, the characteristics of the mobile platform requires more specialized skill sets.

10.3.1.1 Unique Characteristics of the Mobile Platform

Some of the unique characteristics of the mobile platform are:

- Multiple platforms
  - Operating system: iOS, Android, BlackBerry, Windows Mobile
  - Hardware: Apple iPhone and iPad, HTC, Motorola, Samsung, and so on
- Limited resources
  - Weaker CPU
  - Smaller memory
- Variety of connectivity modes
  - Cellular networks: G3, G4, CDMA, GPRS, etc.
  - Wi-Fi
- Multi-platform business processes
  - Mobile app – web – Database – Back-end system

10.3.1.2 Unique Characteristics create Unique Challenges

The characteristics of the mobile application environment create unique challenges for application development and subsequent conditions that will require testing. Some of the challenges presented by the mobile platform are:

- Compatibility:
  - CPU
  - Memory
  - Display size / resolution
  - Keyboard (soft / hard / external)
  - Touch functionality
- Multiple operating systems:
  - iOS versions
  - Android versions
  - Windows Mobile
  - Windows Phone
  - BlackBerry
- Data entry:
Typing on a smartphone keyboard, hard or soft, takes 2-5 times longer than on a computer keyboard.

- External keyboards (Bluetooth)
- Voice

- Unique functionality:
  - Location awareness (GPS + DB(s))
  - Orientation (Portrait / Landscape)
  - Gestures & shake-up (HW and SW-dependent)

### 10.3.2 Identifying Test Conditions on Mobile Apps

As with any application under test, testing a mobile application must first start with identifying the conditions that must be tested. Once the test conditions have been identified the rather academic process of creating test cases to test the conditions follows typical test processes. The tester must be aware of the “Human Interface Guidelines” that may have been part of the application requirement and that conditions should be identified to ensure the human interface guidelines have been followed. Test conditions for mobile apps would include areas such as functionality, performance, usability, security, interoperability, and survivability and recovery.

#### 10.3.2.1 Functionality

Testers must validate that the Mobile App can perform the functionality for which it was designed. Functionality testing should consider:

**Happy, Sad and Alternate Paths** – The tester must validate that the happy path, sad path and the alternate paths through the application execute and return the expected result.

**Installation Processes** – Unlike most other types of applications, mobile apps are likely to be installed by an end user on their own device. The tester must ensure that the installation process works correctly and is easy to understand for the target audience.

**Special Functionality** – The mobile application, more so than most other application delivery models, sports some interesting special functions. Functions such as:

- **Orientation** – Tester must validate, based on the design requirements, that the application changes orientation with the movement of the mobile device and that the display format presents the information in a fashion consistent with the design specification.

- **Location** – If an application requirement is location awareness, the tester must validate that the mobile device’s GPS or tower triangulation provides accurate location coordinates and that those coordinates are utilized by the application as required.

- **Gestures** – Testing, as applicable, that gestures such as “push to refresh” techniques have been implemented correctly.
• **Internationalization** – Ensure that language, weights, and measures work correctly for the location of the device.

• **Barcode scanning** – For mobile devices and applications that support barcodes or QR codes, testers must validate correct operation.

• **Hardware Add-ons** – Tester must validate that any potential hardware add-ons (like a credit card reader) function correctly with the application under test.

**Regulatory Compliance** – The tester must be aware of and validate any regulatory compliance issues. Examples include section 508 of the Americans with Disabilities Act, Health Information Portability and Accountability Act (HIPPA) regulations, and data use Privacy Laws.

### 10.3.2.2 Performance

Testers must validate that the Mobile App’s performance is consistent with stated design specifications. The tester should integrate performance testing with functional testing to measure the impact of load on the user experience. Tests should include:

• Load and stress tests when:
  - Many applications are open
  - System is on for long periods of time
  - Application experiences 2-3 times the expected capacity
  - When data storage spaced is exceeded

• Validate application opening time when:
  - Application is not in memory
  - Application has been pre-loaded

• Performance behavior when:
  - Low battery
  - Bad network coverage
  - Low available memory
  - Simultaneous access to multiple application servers

### 10.3.2.3 Usability

Providing a good user experience for ALL users is critical. The testers must ensure that the application is clear, intuitive and easy to navigate. Considerations when testing the usability of a mobile application are:

• Validate that the system status is visible.

• Test the consistency between what a user’s real-world experience would be and how the application functions. An example might be how the natural and logical order of things is handled.

• Test to validate that user control and freedom such as an emergency exit, undo, redo, or rollback function has been implemented.
• Test to ensure consistency and standards that are followed as dictated by the specific platform.
• Test to validate that error prevention has been done by eliminating error-prone conditions and presenting users with a confirmation option.
• Test to validate that the application was designed in such a fashion that objects and options are visible as necessary reducing the need for users to remember specific functionality.
• Real estate is valuable on the mobile device screen. The tester should ensure that good aesthetics and a minimalist design has been done. Examples might be ensuring that the screen is clear of information which is irrelevant or rarely needed.
• As with all applications, mobile or otherwise, the way the application handles error conditions is critically important to the user’s experience. The tester should validate that the application traps errors, helps the user recognize the error condition by providing easily understood verbiage, and then helps diagnose the problem and recover from error condition.
• Finally, the tester should ensure that help and documentation is available for the user as necessary.

10.3.2.4 Interrupt Testing

Mobile applications will likely experience interruptions during execution. An application will face interruptions such as incoming calls, text messages, or network coverage outage. The tester should create tests to validate the application functions under these various types of interruptions:
• Incoming and Outgoing SMS and MMS
• Incoming and Outgoing calls
• Incoming Notifications
• Cable Insertion and Removal for data transfer
• Network outage and recovery
• Media Player on/off
• Device Power cycle

10.3.2.5 Security

Mobile devices, by definition, are “mobile”. The mobility of the device and by extension the applications and data greatly increases risk. Testers must check how the device is accessed and the functions where system or data can be compromised. Areas of concern that must be tested include:
• Unauthorized access
• Data leakage
• Re-authentication for sensitive activities
• “Remember me” functionality
• Authentication of back-end security protocols
• Searching device’s file system (if possible)

10.3.2.6 Interoperability

Mobile devices frequently connect to other devices to upload, download or share information. The testers must validate that the system securely connects to and interfaces with other systems appropriately. Areas that should be tested include:

• Data Exchange
  ○ Exchanging data with the app server, or DB server
  ○ Data upload after the device is off-line

• Invoking functionality
  ○ Notifications: tray, pop-up, update
  ○ Real-time messaging
  ○ Video streaming

• Application updates

10.3.2.7 Survivability and Recovery

The testers must validate that system protects the system and data when failures occur. Test conditions must be identified that validate system reaction and response to:

• Battery low or failure
• Poor reception and loss of reception
• GPS loss
• Data transfer interruption

10.3.3 Mobile Test Labs

In Skill Category 2, section 2.3.3.2, Test Labs in the general sense were defined. The reasoning and issues surrounding that discussion ring true for the Mobile Test Lab as well if not more so.

The number of models and types of mobile devices on the market today is already large and growing fast. Each device tends to have differences, and deciding which devices to test can be a daunting task. Testing the mobile app in a test lab can provide an appropriate environment necessary to test multiple devices.
10.3.3.1 Devices vs. Emulators

Testing a mobile app is not simply installing the app on various devices and executing some cases. First, a development environment must be set up on the device which will provide testing and debugging capabilities. To install such an environment in the potential devices and operating environments (iOS, Android, etc.) would be challenging. While this does provide the real world experience the question to ask is, at what price?

An emulator is a piece of software running on a computer which simulates a mobile device. An emulator can simulate many different devices without the expense of device specific testing. However, emulators can’t give pixel-perfect resolution that might be needed for some testing, or allow to the tester to see how the app functions in conjunction with the quirks of real-life device. They do allow cost-efficient testing at scale and are a powerful tool to have in your mobile testing lab.

10.3.3.2 Components

When considering a Mobile Test Lab it is important to look at the various components that will be necessary to create a lab environment. Components to consider are:

- Hardware components would include:
  - Devices
  - Wi-Fi routers
  - SIM cards
  - Faraday cage(s)
  - Chargers
  - HW add-ons

- Software components would include:
  - Mobile OS versions
  - Patches
  - Emulators
  - Remote control applications
  - Configuration files
  - Automated scripts
  - Defect management
  - Quality Status dashboard

The investment in hardware and software will be significant so ensure that the devices are physically secured according to your company’s policies and consider insuring expensive devices.
10.3.3.3 Mobile App Testing Lab Procedures

In addition to the lab considerations detailed in section 2.3.3.2, specific procedures for the mobile test lab will include:

A first consideration would be test lab access. Things to consider:

- Who can do what in the lab
- Onboarding new devices
- Renting devices from a remote lab
- Installations
- Testing
- Automation

Second would be device security. Things to consider:

- Are devices accessible only in the lab
- Are devices allowed out of the lab

Third consideration is connectivity management. Things to consider:

- WiFi / USB / other
- Connection procedures for carriers

Fourth would be device management which would include:

- Inventory
- Labeling

Additionally it would be necessary to consider procedures governing the procurement and rental process.

10.3.3.4 Mobile Test Lab Services

Mobile Test Lab third-party services can provide a complete or partial outsourced mobile application testing solution. Organizations can use mobile test services to supplement internal teams on-site as well as extend teams off-site. Mobile testing can be outsourced to specialized test services with some of following potential benefits:

**Save time** – Delays as a result of waiting for the acquisition new devices can be reduced.

**Save money** – A third-party service could dramatically reduce capital investment in the necessary infrastructure as well as providing access to the latest mobile app testing tools without the need to purchase the tool.

**More robust** – By utilizing the most up-to-date mobile test tools the test process can be more robust and precise

**Cloud-based service** – Many services will be cloud-based providing innovative approaches to collaborate with third-party developers, other business partners, and customers. The cloud
approach supports a real-world environment using multiple devices, different operating systems, screen configurations, and carriers.

10.3.4 Mobile App Test Automation

In Skill Category 2, section 2.4, test tools as part of the Software Testing Ecosystem, were described in detail. The advantages and disadvantages would in most cases apply to mobile app automation as well.

There are some specific tools dedicated to automating tests for mobile platforms. Some of the different approaches include:

**Emulation** – As discussed earlier.

**Instrumentation** - Adding special hookups to the application under test to control it by signals from a computer (generally hard-wire connected).

**Remote control** – A variety of mobile test tools provide easy and secure remote control access to the devices under test using a browser.

**Add-ons** – Many software automation tools provide mobile testing add-ons to the existing tools suite.

10.4 Cloud Computing

“Comes from the early days of the Internet where we drew the network as a cloud… we didn’t care where the messages went… the cloud hid it from us”

– Kevin Marks, Google

10.4.1 Defining the Cloud

The National Institute of Standards and Technology define Cloud Computing as “a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.” To put it more simply, in the cloud model, data and computation are operated somewhere in a “cloud” which is some collection of data centers owned and maintained by a third party.
Regarding cloud testing, two perspectives need to be understood:

**Testing Cloud-based Applications** – Testing applications that are deployed in the cloud for such cloud specific nuances as cloud performance, security of cloud applications, and availability and continuity within the cloud.

**The Cloud as a Testing Platform** – Using the cloud environment to generate massive distributed load tests, simulate a large number of mobile devices, or run functional and performance monitors from all over the world.

### 10.4.1.1 Cloud Service Models

Within the cloud computing definition are three service models available to cloud customers:

**Infrastructure-as-a-Service (IaaS)** – Consumer does not manage or control underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and limited control of select networking components.

**Platform-as-a-Service (PaaS)** – Consumer can deploy onto cloud infrastructure consumer-created or acquired applications; created using programming languages, libraries, services, and tools supported by provider. Consumer does not manage or control the underlying cloud infrastructure.

**Software-as-a-Service (SaaS)** – Consumer uses provider’s applications running on cloud infrastructure; applications are accessible from various client devices through a web browser or program interface. The consumer does not manage or control the underlying cloud infrastructure.
10.4.1.2 Basic Cloud Architectures

There are four deployment models that describe how the computing infrastructure that delivers the services can be shared. The four models and basic characteristics are:

- **Private Cloud**
  - Operated solely for an organization
  - Managed by the organization or third party
  - Exists on-premises or off-premises
- **Community Cloud**
  - Shared by specific community of several organizations
  - Managed by the organizations or third party
  - Exists on-premises or off-premises
- **Public Cloud**
  - Available to general public
  - Managed by the organization or third party
  - Exists off-premises
- **Hybrid Cloud**
  - Composition of two or more clouds; remain unique entities
  - Bound together by standardized or proprietary technology
  - Exists on-premises or off-premises

10.4.1.3 Cloud Services Characteristics

There are five essential characteristics that all cloud services exhibit. They are discussed in the following sections.

10.4.1.3.1 On-demand Self-Service

The cloud can be used as and when required without prior subscription. It does not require pre-booking or “phased-delivery” for the consumer; hence, there is no need for under or over subscription in the cloud.

10.4.1.3.2 Ubiquitous Network Access

The cloud offers almost infinite network access to vast infrastructure and computing resources, such as storage facility, processor, memory, hosting and countless applications.

10.4.1.3.3 Location Independence and Homogeneity

The cloud uses a shared pool of resources, platforms and infrastructure residing on the Internet, which is located at various parts of the world, making the cloud location-independent. The services offered in the cloud are homogeneous. The same service is provided exactly in the same way to all users.
10.4.1.3.4 Elastically Scalable

Cloud computing capabilities, such as storage, computing power, processing and hosting are elastic; resources are pooled together to provide vast amounts of computing power.

10.4.1.3.5 Measured Service

Cloud computing services are measured; each service purchased or utilized by a consumer is measured and billed accordingly.

10.4.2 Testing in the Cloud

As with all specialized technologies discussed, testing cloud based applications is first and foremost a testing process which entails many of the generalized testing processes discussed earlier in this book. However, there are certain peculiarities to testing cloud applications that should be understood. The titles of these different test condition categories are similar to those we have seen in web, mobile, etc.; however, the specific nature of the tests is different.

10.4.2.1 Performance Testing

Cloud based applications by their very definition often run on hardware over which the application owner has little or no control. Further to that concern, cloud apps may well be sharing the hardware and operating environments with other applications. This characteristic of cloud based apps intensifies the need for performance and scalability testing.

10.4.2.1.1 Response Time and Latency

When testing cloud based apps, the tester must test response time in normal state as well as the expected peaks. It is also important to approximate a work case scenario so testing at those unexpected peaks can happen as well.

10.4.2.1.2 Elasticity

Response time and latency tests how quickly the cloud based application returns the data requested. Elasticity tests the ability of the application to handle sudden, unanticipated, and extraordinary loads. An example might be “Cyber Monday” (Monday after the Thanksgiving Holiday weekend in the United States).

10.4.2.1.3 Scalability

As with any application system, the ability to test the system’s performance using a planned level of capacity growth is necessary. While many Cloud environments are specifically designed to provide robust growth, testing to ensure reasonable scalability is a requirement.
10.4.2.1.4 Load and Stress Testing

The test suite should contain test conditions to test normal user, peak, simultaneous logins, steadily increasing and peak loads.

10.4.2.2 Security Testing

Cloud based applications usually share resources and infrastructure with others. The tester must give extra consideration to ensuring that data privacy and access control issues are working correctly. Some of the conditions to test include the items discussed in the following sections.

10.4.2.2.1 Infrastructure Leveling

Infrastructure leveling is testing to validate firewalls, Intrusion Prevention Systems (IPS), and VPN gateways.

10.4.2.2.2 Encryption

Cloud based architecture increases the need for message encryption. The testers should test messages both with and without encryption.

10.4.2.2.3 Authentication

Authentication is always a non-functional test which should be performed on an application system. Cloud only increases the need for testing to ensure that the policies defining which devices and users are allowed on the network are adhered to and the control is working.

10.4.2.2.4 Authorization

Similar to authentication, the tester must test the sufficiency of authorized roles, functions, and restrictions on accessing others’ data.

10.4.2.3 Availability and Continuity Testing

The cloud based application tester develops tests which both reactively and proactively test that IT services can recover and continue even after an incident occurs. Some of the conditions to test include the following.

10.4.2.3.1 Resilience

These are tests to ensure that disruptions will not materially affect critical business functions and supporting infrastructure. This requires testing of redundancy, automated failover, dual-live “hot sites,” mirroring/clustering and similar high availability techniques.

- **Fail-over Testing** – Test for component failure based on anticipated load.
• **Active-active Systems** – Test for capability of a network of independent processing nodes to access a replicated database and application.

### 10.4.2.3 Disaster Recovery Planning

Tester should validate that appropriate arrangements are in place to recover critical IT services to operation after a serious incident.

### 10.4.2.3.3 Contingency Planning

Validate that a process is in place to plan for uncertain future situations.

### 10.4.2.4 Third-party Dependencies

Cloud applications are likely to consume external APIs and services for providing some of their functionality.

#### 10.4.2.4.1 Test

The tester should test the external APIs and services as part of the regular test processes. While they may not specifically be part of the organization’s application(s) system they are a part of the application solution. The tester must ensure they are working properly.

#### 10.4.2.4.2 Monitor

On an ongoing basis, the tester should monitor these external APIs and services to ensure they continue to work. Unlike a more traditional environment where changes and releases are controlled by the development team, Cloud application issues may arise without proper notification.

### 10.4.3 Testing as a Service (TaaS)

Section 10.4.2 described the testing of Cloud based applications. In this section, testing using a Cloud based environment is described. In that sense, cloud testing is defined as “Testing as Service.” Test teams can extend their test environment capabilities by making use of a Cloud based model for application testing. The Cloud testing approach can test functional and non-functional conditions.

#### 10.4.3.1 SaaS Model (Software-as-a-Service)

In the SaaS model, testing products and services are recognized as on demand testing applications and are deployed for black box testing techniques.
10.4.3.2 PaaS Model (Platform-as-a-Service)

In the PaaS model, both build (development) and automation environments are available on demand. Testers can make use of the on demand environments to build and test the automation scripts.

10.4.3.3 IaaS Model (Infrastructure-as-a-Service)

In the IaaS model, organizations provide their storage servers or network and other computing infrastructure on demand which are used for validation testing in production-like environments.

10.5 Testing in an Agile Environment

In earlier skill categories, Agile as a development framework was been discussed. In Skill Category 7, User Stories as a source for test conditions and how those conditions become test cases was explained. In this section an overview of some specific testing issues relative to the Agile environment will be described.

10.5.1 Agile as an Iterative Process

The first principle in agile testing is that testing is iterative, just like gathering and eliciting requirements, understanding and refinement of requirements, definition and design of the software, development and building of software to specification, and even planning. It is critically important that a new mindset be adopted such that the testing processes also follow this principle and are iteratively performed.

10.5.2 Testers Cannot Rely on having Complete Specifications

The second principle reiterates and confirms that testers in an agile project cannot rely on having complete specifications. It is important to recognize that detailed SRS documents with all defined positive, negative, and all the alternative flows is not the convention of Agile Projects. Rather, testers should align to the fact that the requirements are identified, explored, and implemented throughout the life cycle of the project. Unlike a conventional project there is no single phase where comprehensive documentation is produced. So the focus of the test strategy should rely on ensuring quality without actually having any specifications in place.
10.5.3 Agile Testers must be Flexible

The third principle emphasizes on the necessity to maintain a flexible approach to testing. Agility requires flexibility as a core construct so it is imperative for the testers to work to the best of their ability with the information provided and gain a thorough understanding about the project through working in collaboration with others on the team. So the core of the entire principle is that testers, as members of the Agile team, are flexible, cooperative, and willing to work with other team members to ensure that quality is maintained.

10.5.4 Key Concepts for Agile Testing

There are a number of concepts that should be considered when working on Agile projects.

• Planning is the key to successful testing in Agile projects
• Testing is no longer a separate phase in Agile projects
• Requires a trained team to execute continuous testing
• Testers commit to tasks as part of Agile team in sprint planning
• Collaboration is key to understanding and testing the requirements
• Plan for current iteration and also any testing for existing features dependencies
• Along with user stories, need to understand acceptance criteria clearly
• Have a clear understanding of DONE criteria

10.5.5 Traditional vs. Agile Testing

Table 10-1 shows a comparison of traditional testing practices with agile testing practices.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Traditional Testing</th>
<th>Agile Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understanding Requirements</td>
<td>Upfront</td>
<td>Constant Interaction with team and Product Owner</td>
</tr>
<tr>
<td>Test Requirements</td>
<td>Complete and Baseline (freeze)</td>
<td>Incremental Req. as stories</td>
</tr>
<tr>
<td></td>
<td>Separate Change Request/ Enhancement</td>
<td>To accommodate change/dynamic Req.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Req. prioritized based on: business values of customers, early realization and feedback</td>
</tr>
<tr>
<td>Criteria</td>
<td>Traditional Testing</td>
<td>Agile Testing</td>
</tr>
<tr>
<td>--------------------------</td>
<td>----------------------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Planning</td>
<td>Plan one time delivery which will happen much later</td>
<td>Continuous sprint by sprint planning, deliver important features first</td>
</tr>
<tr>
<td>Designing of Test Cases</td>
<td>All done upfront and frozen before testing</td>
<td>Evolving</td>
</tr>
<tr>
<td>Managing Change</td>
<td>No Change. Change can come only as a CR</td>
<td>Adapt and adjust at every release and iteration boundary</td>
</tr>
<tr>
<td>Test Team Structure</td>
<td>Independent Test Team</td>
<td>Collaborative Team</td>
</tr>
<tr>
<td>Progress Review</td>
<td>Deliverables and Milestones reviews</td>
<td>See a working set of software at the end of every iteration and every release</td>
</tr>
<tr>
<td>Quality Responsibility</td>
<td>Test Engineer</td>
<td>Entire Team</td>
</tr>
<tr>
<td>Test Automation</td>
<td>Automation behind Manual Testing Phase</td>
<td>Iteration based automation</td>
</tr>
</tbody>
</table>

*Table 10-1 Traditional vs. Agile Practices*

### 10.5.6 Agile Testing Success Factors

It is the responsibility of everyone on the Agile team to ensure the success of the Agile project. Shown here are testing success factors that must always be considered:

- Testers are part of the team
  - Collective Ownership
- Agile Testing Mindset
  - Drop the "Quality Police" mindset
  - Focus on team goals & customer value
- Automate tests
  - Automate tests wherever practical
  - Need rapid feedback
- Look at the big picture
  - Balance against developer focus on technical implementation
- Collaborate
  - Collaborate with customers
  - Collaborate with team
- Continually improve
  - Team Retrospectives
10.6 DevOps

“The emerging professional movement that advocates a collaborative working relationship between Development and IT Operations, resulting in the fast flow of planned work (i.e., high deploy rates), while simultaneously increasing the reliability, stability, resilience of the production environment.”

10.6.1 DevOps Continuous Integration

In the extremely fast paced world that today’s IT organizations perform in, it is not unusual for new builds to be measured in hundreds per day rather than several per week. Organizations like Netflix, Facebook, Amazon, Twitter, and Google in some cases can deploy thousands of code builds per day while delivering world-class stability, reliability, and security. Traditional development lifecycle approaches are not intended to deliver nor can they deliver at this pace. Enter, DevOps.

The DevOps model creates a seamless integrated system moving from the development team writing the code, automatically deploying the executable code into the automated test environment, having the test team execute the automated tests, and then deployment into the production environment. This process moves in one smooth integrated flow. Automation plays a pivotal role in the DevOps process. The use of Continuous Integration tools and test automation are the standard in the DevOps model.

Figure 10-4 DevOps Model
The DevOps model emphasizes communication, collaboration, and integration between software developers, test teams, and operations personnel. Pulling from the best practices of both Agile and Lean, the DevOps approach recognizes the importance of close collaboration between the development team, which includes the testers, and operations staff.

### 10.6.2 DevOps Lifecycle

The DevOps lifecycle has the following process:

- Check in code
- Pull code changes for build
- Use of continuous integrations tool to generate builds and arrange releases. Test automation includes:
  - unit tests
  - integration tests
  - functional tests
  - acceptance tests
- Store artifacts and build repository (configuration management for storing artifacts, results, and releases)
- Use of release automation tool to deploy apps into production environment
- Configure environment
- Update databases
- Update apps
- Push to users

Within DevOps, every action in the chain is automated. This approach allows the application development team to focus on designing, coding, and testing a high quality deliverable.

### 10.6.3 Testers Role in DevOps

Unquestionably, the traditional role of the software testers is different within a DevOps lifecycle. Some of the changes are:

- The Software Test Team is required to align their efforts in the DevOps cycle.
- Testers have to ensure that all test cases are automated and achieve near 100% code coverage.
- Testers must ensure that their environments are standardized and deployment into the test environment is automated.
- All their pre-testing tasks, cleanups, post-testing tasks, etc. are automated and aligned with the Continuous Integration cycle.
Similar to the impact of Agile on the role of all individuals in the development cycle, DevOps encourages everyone to contribute across the chain. Ultimately, the quality and timeliness of the application system is the responsibility of everyone within the chain.

10.6.4 DevOps and Test Automation

Test automation is a requirement in the DevOps cycle. Test processes must run automatically when the deployment is completed in the test environment. Specialized automation testing tools and continuous integration tools are used to achieve this level of integration. A mature automation testing framework must exist so scripting new test cases can happen quickly.

10.7 The Internet of Things

The Internet of Things (IoT) as defined in the book, From Machine-to-Machine to the Internet of Things: Introduction to a New Age of Intelligence, is the interconnection of uniquely identifiable embedded computing devices within the existing Internet infrastructure. Typically, IoT is expected to offer advanced connectivity of devices, systems, and services that goes beyond machine-to-machine communications (M2M) and covers a variety of protocols, domains, and applications.

10.7.1 What is a Thing?

A “thing”, in the Internet of Things, can be a person with a heart monitor implant, an automobile that has built-in sensors to alert the driver when tire pressure is low, or any other natural or man-made object that can be assigned an IP address and provided with the ability to transfer data over a network. The Internet of Things has been around for some time but primarily in the machine-to-machine (M2M) communication in manufacturing and power, and oil and gas utilities. Devices like a “smart meters” for electric and water monitoring are examples of M2M communication. Other IoT devices currently in use are smart thermostat systems and washer/dryers that utilize wifi for remote monitoring.

10.7.2 IPv6 as a Critical Piece

The implementation of IPv6 was a critical piece needed for the IoT. According to Steve Leibson, who identifies himself as “occasional docent at the Computer History Museum,” the address space expansion means that we could “assign an IPV6 address to every atom on the

surface of the earth, and still have enough addresses left to do another 100+ earths.” In other words, humans could easily assign an IP address to every “thing” on the planet.

10.7.3 Impact of IoT

According to McKinsey Global Institute, the Internet of Things has the potential to create an economic impact of $2.7 trillion to $6.2 trillion annually by 2025 (McKinsey Global Institute, Disruptive technologies: Advances that will transform life, business, and the global economy, May 2013). Further, Gartner’s prognosticators say there will be nearly 26 billion devices on the Internet of Things by 2020 and ABI Research estimates that more than 30 billion devices will be wirelessly connected to the Internet of Things by 2020. In the diagram shown as Figure 10-5, SRI Consulting Business Intelligence created this Technology Roadmap which highlights the timing, features, and applications of significant technology milestones in the IoT as it continues to develop through 2025.

![IoT Roadmap](image)

*Figure 10-5  IoT Roadmap*

10.7.4 Testing and the Internet of Things

Implementation of the IoT technologies will require organizations to rethink the testing process. The IoT is about reporting data in real time, allowing users to make quicker, more informed decisions. One of the greatest challenges for the tester is how to simulate the real life...
scenarios that connected devices undergo in operation. Testing will become even more holistic as the development and test teams work to ensure that the devices and apps stand up to real world situations and maintain their high level of quality no matter what they are put through. Invariably testing will move out of the lab and into the wild where poor connectivity and imperfect conditions abound.
Vocabulary

Acceptance Criteria
A key prerequisite for test planning is a clear understanding of what must be accomplished for the test project to be deemed successful.

Those things a user will be able to do with the product after a story is implemented. (Agile)

Acceptance Testing
The objective of acceptance testing is to determine throughout the development cycle that all aspects of the development process meet the user’s needs.

Act
If your checkup reveals that the work is not being performed according to plan or that results are not as anticipated, devise measures for appropriate action. (Plan-Do-Check-Act)

Access Modeling
Used to verify that data requirements (represented in the form of an entity-relationship diagram) support the data demands of process requirements (represented in data flow diagrams and process specifications.)

Active Risk
Risk that is deliberately taken on. For example, the choice to develop a new product that may not be successful in the marketplace.

Actors
Interfaces in a system boundary diagram. (Use Cases)

Alternate Path
Additional testable conditions are derived from the exceptions and alternative course of the Use Case.

Affinity Diagram
A group process that takes large amounts of language data, such as a list developed by brainstorming, and divides it into categories.

Analogous
The analogy model is a nonalgorithmic costing model that estimates the size, effort, or cost of a project by relating it to
another similar completed project. Analogous estimating takes the actual time and/or cost of a historical project as a basis for the current project.

<table>
<thead>
<tr>
<th>Analogous Percentage Method</th>
<th>A common method for estimating test effort is to calculate the test estimate as a percentage of previous test efforts using a predicted size factor (SF) (e.g., SLOC or FPA).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>A single software product that may or may not fully support a business function.</td>
</tr>
<tr>
<td>Appraisal Costs</td>
<td>Resources spent to ensure a high level of quality in all development life cycle stages which includes conformance to quality standards and delivery of products that meet the user’s requirements/needs. Appraisal costs include the cost of in-process reviews, dynamic testing, and final inspections.</td>
</tr>
<tr>
<td>Appreciative or Enjoyment Listening</td>
<td>One automatically switches to this type of listening when it is perceived as a funny situation or an explanatory example will be given of a situation. This listening type helps understand real-world situations.</td>
</tr>
<tr>
<td>Assumptions</td>
<td>A thing that is accepted as true.</td>
</tr>
<tr>
<td>Audit</td>
<td>This is an inspection/assessment activity that verifies compliance with plans, policies, and procedures, and ensures that resources are conserved. Audit is a staff function; it serves as the “eyes and ears” of management.</td>
</tr>
<tr>
<td>Backlog</td>
<td>Work waiting to be done; for IT this includes new systems to be developed and enhancements to existing systems. To be included in the development backlog, the work must have been cost-justified and approved for development. A product backlog in Scrum is a prioritized featured list containing short descriptions of all functionality desired in the product.</td>
</tr>
<tr>
<td>Baseline</td>
<td>A quantitative measure of the current level of performance.</td>
</tr>
<tr>
<td>Benchmarking</td>
<td>Comparing your company’s products, services, or processes against best practices, or competitive practices, to help define superior performance of a product, service, or support process.</td>
</tr>
<tr>
<td>Benefits Realization Test</td>
<td>A test or analysis conducted after an application is moved into production to determine whether it is likely to meet the originating business case.</td>
</tr>
<tr>
<td>Black-Box Testing</td>
<td>A test technique that focuses on testing the functionality of the program, component, or application against its specifications without knowledge of how the system is constructed; usually data or business process driven.</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Bottom-Up</strong></td>
<td>Begin testing from the bottom of the hierarchy and work up to the top. Modules are added in ascending hierarchical order. Bottom-up testing requires the development of driver modules, which provide the test input, that call the module or program being tested, and display test output.</td>
</tr>
<tr>
<td><strong>Bottom-Up Estimation</strong></td>
<td>In this technique, the cost of each single activity is determined with the greatest level of detail at the bottom level and then rolls up to calculate the total project cost.</td>
</tr>
<tr>
<td><strong>Boundary Value Analysis</strong></td>
<td>A data selection technique in which test data is chosen from the “boundaries” of the input or output domain classes, data structures, and procedure parameters. Choices often include the actual minimum and maximum boundary values, the maximum value plus or minus one, and the minimum value plus or minus one.</td>
</tr>
<tr>
<td><strong>Brainstorming</strong></td>
<td>A group process for generating creative and diverse ideas.</td>
</tr>
<tr>
<td><strong>Branch Combination Coverage</strong></td>
<td>Branch Condition Combination Coverage is a very thorough structural testing technique, requiring $2^n$ test cases to achieve 100% coverage of a condition containing $n$ Boolean operands.</td>
</tr>
<tr>
<td><strong>Branch/Decision Testing</strong></td>
<td>A test method that requires that each possible branch on each decision point be executed at least once.</td>
</tr>
<tr>
<td><strong>Bug</strong></td>
<td>A general term for all software defects or errors.</td>
</tr>
<tr>
<td><strong>Calibration</strong></td>
<td>This indicates the movement of a measure so it becomes more valid, for example, changing a customer survey so it better reflects the true opinions of the customer.</td>
</tr>
<tr>
<td><strong>Candidate</strong></td>
<td>An individual who has met eligibility requirements for a credential awarded through a certification program, but who has not yet earned that certification through participation in the required skill and knowledge assessment instruments.</td>
</tr>
<tr>
<td><strong>Causal Analysis</strong></td>
<td>The purpose of causal analysis is to prevent problems by determining the problem’s root cause. This shows the relation between an effect and its possible causes to eventually find the root cause of the issue.</td>
</tr>
<tr>
<td><strong>Cause and Effect Diagrams</strong></td>
<td>A cause and effect diagram visualizes results of brainstorming and affinity grouping through major causes of a significant process problem.</td>
</tr>
<tr>
<td><strong>Cause-Effect Graphing</strong></td>
<td>Cause-effect graphing is a technique which focuses on modeling the dependency relationships between a program’s input conditions (causes) and output conditions (effects). CEG is considered a Requirements-Based test technique and is often referred to as Dependency modeling.</td>
</tr>
</tbody>
</table>
Certificant
An individual who has earned a credential awarded through a certification program.

Certification
A voluntary process instituted by a nongovernmental agency by which individual applicants are recognized for having achieved a measurable level of skill or knowledge. Measurement of the skill or knowledge makes certification more restrictive than simple registration, but much less restrictive than formal licensure.

Change Management
Managing software change is a process. The process is the primary responsibility of the software development staff. They must assure that the change requests are documented, that they are tracked through approval or rejection, and then incorporated into the development process.

Check
Check to determine whether work is progressing according to the plan and whether the expected results are obtained. Check for performance of the set procedures, changes in conditions, or abnormalities that may appear. As often as possible, compare the results of the work with the objectives.

Check Sheets
A check sheet is a technique or tool to record the number of occurrences over a specified interval of time; a data sample to determine the frequency of an event.

Checklists
A series of probing questions about the completeness and attributes of an application system. Well-constructed checklists cause evaluation of areas, which are prone to problems. It both limits the scope of the test and directs the tester to the areas in which there is a high probability of a problem.

Checkpoint Review
Held at predefined points in the development process to evaluate whether certain quality factors (critical success factors) are being adequately addressed in the system being built. Independent experts for the purpose of identifying problems conduct the reviews as early as possible.

Client
The customer that pays for the product received and receives the benefit from the use of the product.

CMMI-Dev
A process improvement model for software development. Specifically, CMMI for Development is designed to compare an organization’s existing development processes to proven best practices developed by members of industry, government, and academia.

Coaching
Providing advice and encouragement to an individual or individuals to promote a desired behavior.

COCOMO II
The best recognized software development cost model is the Constructive Cost Model II.COCOMO®II is an enhancement over
the original COCOMO® model. COCOMO®II extends the capability of the model to include a wider collection of techniques and technologies. It provides support for object-oriented software, business software, software created via spiral or evolutionary development models and software using COTS application utilities.

**Code Comparison**

One version of source or object code is compared to a second version. The objective is to identify those portions of computer programs that have been changed. The technique is used to identify those segments of an application program that have been altered as a result of a program change.

**Common Causes of Variation**

Common causes of variation are typically due to a large number of small random sources of variation. The sum of these sources of variation determines the magnitude of the process’s inherent variation due to common causes; the process’s control limits and current process capability can then be determined.

**Compiler-Based Analysis**

Most compilers for programming languages include diagnostics that identify potential program structure flaws. Many of these diagnostics are warning messages requiring the programmer to conduct additional investigation to determine whether or not the problem is real. Problems may include syntax problems, command violations, or variable/data reference problems. These diagnostic messages are a useful means of detecting program problems, and should be used by the programmer.

**Complete Test Set**

A test set containing data that causes each element of pre-specified set of Boolean conditions to be true. In addition, each element of the test set causes at least one condition to be true.

**Completeness**

The property that all necessary parts of an entity are included. Often, a product is said to be complete if it has met all requirements.

**Complexity-Based Analysis**

Based upon applying mathematical graph theory to programs and preliminary design language specification (PDLs) to determine a unit's complexity. This analysis can be used to measure and control complexity when maintainability is a desired attribute. It can also be used to estimate test effort required and identify paths that must be tested.

**Compliance Checkers**

A parse program looking for violations of company standards. Statements that contain violations are flagged. Company standards are rules that can be added, changed, and deleted as needed.

**Comprehensive Listening**

Designed to get a complete message with minimal distortion. This type of listening requires a lot of feedback and summarization to fully understand what the speaker is communicating.

**Compromise**

An intermediate approach – Partial satisfaction is sought for both
parties through a “middle ground” position that reflects mutual
sacrifice. Compromise evokes thoughts of giving up something,
therefore earning the name “lose-lose.”

**Condition Coverage**

A white-box testing technique that measures the number of, or
percentage of, decision outcomes covered by the test cases
designed. 100% condition coverage would indicate that every
possible outcome of each decision had been executed at least
once during testing.

**Condition Testing**

A structural test technique where each clause in every condition is
forced to take on each of its possible values in combination with
those of other clauses.

**Configuration Management**

Software Configuration Management (CM) is a process for tracking
and controlling changes in the software. The ability to maintain
control over changes made to all project artifacts is critical to the
success of a project. The more complex an application is, the more
important it is to control change to both the application and its
supporting artifacts.

**Configuration Management Tools**

Tools that are used to keep track of changes made to systems and
all related artifacts. These are also known as version control tools.

**Configuration Testing**

Testing of an application on all supported hardware and software
platforms. This may include various combinations of hardware
types, configuration settings, and software versions.

**Consistency**

The property of logical coherence among constituent parts. Consistency can also be expressed as adherence to a given set of rules.

**Consistent Condition Set**

A set of Boolean conditions such that complete test sets for the
conditions uncover the same errors.

**Constraints**

A limitation or restriction. Constraints are those items that will likely
force a dose of “reality” on a test project. The obvious constraints
are test staff size, test schedule, and budget.

**Constructive Criticism**

A process of offering valid and well-reasoned opinions about the
work of others, usually involving both positive and negative
comments, in a friendly manner rather than an oppositional one.

**Control**

Control is anything that tends to cause the reduction of risk. Control
can accomplish this by reducing harmful effects or by reducing the
frequency of occurrence.

**Control Charts**

A statistical technique to assess, monitor and maintain the stability
of a process. The objective is to monitor a continuous repeatable
process and the process variation from specifications. The intent of
a control chart is to monitor the variation of a statistically stable process where activities are repetitive.

**Control Flow Analysis**

Based upon graphical representation of the program process. In control flow analysis, the program graph has nodes, which represent a statement or segment possibly ending in an unresolved branch. The graph illustrates the flow of program control from one segment to another as illustrated through branches. The objective of control flow analysis is to determine potential problems in logic branches that might result in a loop condition or improper processing.

**Conversion Testing**

Validates the effectiveness of data conversion processes, including field-to-field mapping, and data translation.

**Corrective Controls**

Corrective controls assist individuals in the investigation and correction of causes of risk exposures that have been detected.

**Correctness**

The extent to which software is free from design and coding defects (i.e., fault-free). It is also the extent to which software meets its specified requirements and user objectives.

**Cost of Quality (COQ)**

Money spent beyond expected production costs (labor, materials, equipment) to ensure that the product the customer receives is a quality (defect free) product. The Cost of Quality includes prevention, appraisal, and failure costs.

**COTS**

Commercial Off the Shelf (COTS) software products that are ready-made and available for sale in the marketplace.

**Coverage**

A measure used to describe the degree to which the application under test (AUT) is tested by a particular test suite.

**Coverage-Based Analysis**

A metric used to show the logic covered during a test session, providing insight to the extent of testing. The simplest metric for coverage would be the number of computer statements executed during the test compared to the total number of statements in the program. To completely test the program structure, the test data chosen should cause the execution of all paths. Since this is not generally possible outside of unit test, general metrics have been developed which give a measure of the quality of test data based on the proximity to this ideal coverage. The metrics should take into consideration the existence of infeasible paths, which are those paths in the program that have been designed so that no data will cause the execution of those paths.

**Critical Listening**

The listener is performing an analysis of what the speaker said. This is most important when it is felt that the speaker is not in complete control of the situation, or does not know the complete facts of a situation.
<table>
<thead>
<tr>
<th><strong>Critical Success Factors</strong></th>
<th>Critical Success Factors (CSFs) are those criteria or factors that must be present in a software application for it to be successful.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Customer</strong></td>
<td>The individual or organization, internal or external to the producing organization that receives the product.</td>
</tr>
<tr>
<td><strong>Customer’s/User’s of Software View of Quality</strong></td>
<td>Fit for use.</td>
</tr>
<tr>
<td><strong>Cyclomatic Complexity</strong></td>
<td>The number of decision statements, plus one.</td>
</tr>
<tr>
<td><strong>Damaging Event</strong></td>
<td>Damaging Event is the materialization of a risk to an organization’s assets.</td>
</tr>
<tr>
<td><strong>Data Dictionary</strong></td>
<td>Provides the capability to create test data to test validation for the defined data elements. The test data generated is based upon the attributes defined for each data element. The test data will check both the normal variables for each data element as well as abnormal or error conditions for each data element.</td>
</tr>
<tr>
<td><strong>Data Flow Analysis</strong></td>
<td>In data flow analysis, we are interested in tracing the behavior of program variables as they are initialized and modified while the program executes.</td>
</tr>
<tr>
<td><strong>DD (Decision-to-Decision) Path</strong></td>
<td>A path of logical code sequence that begins at a decision statement or an entry and ends at a decision statement or an exit.</td>
</tr>
<tr>
<td><strong>Debugging</strong></td>
<td>The process of analyzing and correcting syntactic, logic, and other errors identified during testing.</td>
</tr>
<tr>
<td><strong>Decision Analysis</strong></td>
<td>This technique is used to structure decisions and to represent real-world problems by models that can be analyzed to gain insight and understanding. The elements of a decision model are the decisions, uncertain events, and values of outcomes.</td>
</tr>
<tr>
<td><strong>Decision Coverage</strong></td>
<td>A white-box testing technique that measures the number of, or percentage of, decision directions executed by the test case designed. 100% decision coverage would indicate that all decision directions had been executed at least once during testing. Alternatively, each logical path through the program can be tested. Often, paths through the program are grouped into a finite set of classes, and one path from each class is tested.</td>
</tr>
<tr>
<td><strong>Decision Table</strong></td>
<td>A tool for documenting the unique combinations of conditions and associated results in order to derive unique test cases for validation testing.</td>
</tr>
<tr>
<td><strong>Decision Trees</strong></td>
<td>This provides a graphical representation of the elements of a decision model.</td>
</tr>
</tbody>
</table>
Defect
Operationally, it is useful to work with two definitions of a defect:
• From the producer’s viewpoint a defect is a product requirement that has not been met or a product attribute possessed by a product or a function performed by a product that is not in the statement of requirements that define the product;
• From the customer’s viewpoint a defect is anything that causes customer dissatisfaction, whether in the statement of requirements or not.

A defect is an undesirable state. There are two types of defects: process and product.

Defect Management
Process to identify and record defect information whose primary goal is to prevent future defects.

Defect Tracking Tools
Tools for documenting defects as they are found during testing and for tracking their status through to resolution.

Deliverables
Any product or service produced by a process. Deliverables can be interim or external. Interim deliverables are produced within the process but never passed on to another process. External deliverables may be used by one or more processes. Deliverables serve as both inputs to and outputs from a process.

Design Level
The design decomposition of the software item (e.g., system, subsystem, program, or module).

Desk Checking
The most traditional means for analyzing a system or a program. Desk checking is conducted by the developer of a system or program. The process involves reviewing the complete product to ensure that it is structurally sound and that the standards and requirements have been met. This tool can also be used on artifacts created during analysis and design.

Detective Controls
Detective controls alert individuals involved in a process so that they are aware of a problem.

Discriminative Listening
Directed at selecting specific pieces of information and not the entire communication.

Do
Create the conditions and perform the necessary teaching and training to ensure everyone understands the objectives and the plan. (Plan-Do-Check-Act)

The procedures to be executed in a process. (Process Engineering)

Driver
Code that sets up an environment and calls a module for test.
A driver causes the component under test to exercise the interfaces. As you move up the drivers are replaced with the actual components.

**Dynamic Analysis**

Analysis performed by executing the program code. Dynamic analysis executes or simulates a development phase product, and it detects errors by analyzing the response of a product to sets of input data.

**Dynamic Assertion**

A dynamic analysis technique that inserts into the program code assertions about the relationship between program variables. The truth of the assertions is determined as the program executes.

**Ease of Use and Simplicity**

These are functions of how easy it is to capture and use the measurement data.

**Effectiveness**

Effectiveness means that the testers completed their assigned responsibilities.

**Efficiency**

Efficiency is the amount of resources and time required to complete test responsibilities.

**Empowerment**

Giving people the knowledge, skills, and authority to act within their area of expertise to do the work and improve the process.

**Entrance Criteria**

Required conditions and standards for work product quality that must be present or met for entry into the next stage of the software development process.

**Environmental Controls**

Environmental controls are the means which management uses to manage the organization.

**Equivalence Partitioning**

The input domain of a system is partitioned into classes of representative values so that the number of test cases can be limited to one-per-class, which represents the minimum number of test cases that must be executed.

**Error or Defect**

A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.

**Error Guessing**

Test data selection technique for picking values that seem likely to cause defects. This technique is based upon the theory that test cases and test data can be developed based on the intuition and experience of the tester.

**Exhaustive Testing**

Executing the program through all possible combinations of values for program variables.

**Exit Criteria**

Standards for work product quality, which block the promotion of incomplete or defective work products to subsequent stages of the software development process.
Exploratory Testing  
The term “Exploratory Testing” was coined in 1983 by Dr. Cem Kaner. Dr. Kaner defines exploratory testing as “a style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the quality of his/her work by treating test-related learning, test design, test execution, and test result interpretation as mutually supportive activities that run in parallel throughout the project.”

Failure Costs  
All costs associated with defective products that have been delivered to the user and/or moved into production. Failure costs can be classified as either “internal” failure costs or “external” failure costs.

File Comparison  
Useful in identifying regression errors. A snapshot of the correct expected results must be saved so it can be used for later comparison.

Fitness for Use  
Meets the needs of the customer/user.

Flowchart  
Pictorial representations of data flow and computer logic. It is frequently easier to understand and assess the structure and logic of an application system by developing a flowchart than to attempt to understand narrative descriptions or verbal explanations. The flowcharts for systems are normally developed manually, while flowcharts of programs can be produced.

Force Field Analysis  
A group technique used to identify both driving and restraining forces that influence a current situation.

Formal Analysis  
Technique that uses rigorous mathematical techniques to analyze the algorithms of a solution for numerical properties, efficiency, and correctness.

FPA  
Function Point Analysis a sizing method in which the program’s functionality is measured by the number of ways it must interact with the users.

Functional System Testing  
Functional system testing ensures that the system requirements and specifications are achieved. The process involves creating test conditions for use in evaluating the correctness of the application.

Functional Testing  
Application of test data derived from the specified functional requirements without regard to the final program structure.

Gap Analysis  
This technique determines the difference between two variables. A gap analysis may show the difference between perceptions of importance and performance of risk management practices. The gap analysis may show discrepancies between what is and what needs to be done. Gap analysis shows how large the gap is and how far the leap is to cross it. It identifies the resources available to
deal with the gap.

**Happy Path** Generally used within the discussion of Use Cases, the happy path follows a single flow uninterrupted by errors or exceptions from beginning to end.

**Heuristics** Experience-based techniques for problem solving, learning, and discovery.

**Histogram** A graphical description of individually measured values in a data set that is organized according to the frequency or relative frequency of occurrence. A histogram illustrates the shape of the distribution of individual values in a data set along with information regarding the average and variation.

Pareto charts are a special use of a histogram.

**Incremental Model** The incremental model approach subdivides the requirements specifications into smaller buildable projects (or modules). Within each of those smaller requirements subsets, a development life cycle exists which includes the phases described in the Waterfall approach.

**Incremental Testing** Incremental testing is a disciplined method of testing the interfaces between unit-tested programs as well as between system components. It involves adding unit-tested programs to a given module or component one by one, and testing each resultant combination.

**Infeasible Path** A sequence of program statements that can never be executed.

**Influence Diagrams** Provides a graphical representation of the elements of a decision model.

**Inherent Risk** Inherent Risk is the risk to an organization in the absence of any actions management might take to alter either the risk’s likelihood or impact.

**Inputs** Materials, services, or information needed from suppliers to make a process work, or build a product.

**Inspection** A formal assessment of a work product conducted by one or more qualified independent reviewers to detect defects, violations of development standards, and other problems. Inspections involve authors only when specific questions concerning deliverables exist. An inspection identifies defects, but does not attempt to correct them. Authors take corrective actions and arrange follow-up reviews as needed.

**Instrumentation** The insertion of additional code into a program to collect information about program behavior during program execution.
<p>| <strong>Integration Testing</strong> | This test begins after two or more programs or application components have been successfully unit tested. It is conducted by the development team to validate the technical quality or design of the application. It is the first level of testing which formally integrates a set of programs that communicate among themselves via messages or files (a client and its server(s), a string of batch programs, or a set of online modules within a dialogue or conversation.) |
| <strong>Invalid Input</strong> | Test data that lays outside the domain of the function the program represents. |
| <strong>ISO 29119</strong> | ISO 29119 is a set of standards for software testing that can be used within any software development life cycle or organization. |
| <strong>Iterative Model</strong> | The project is divided into small parts allowing the development team to demonstrate results earlier on in the process and obtain valuable feedback from system users. |
| <strong>Judgment</strong> | A decision made by individuals that is based on three criteria which are: fact, standards, and experience. |
| <strong>Keyword-Driven Testing</strong> | Keyword-driven testing, also known as table-driven testing or action word based testing, is a testing methodology whereby tests are driven wholly by data. Keyword-driven testing uses a table format, usually a spreadsheet, to define keywords or action words for each function that will be executed. |
| <strong>Leadership</strong> | The ability to lead, including inspiring others in a shared vision of what can be, taking risks, serving as a role model, reinforcing and rewarding the accomplishments of others, and helping others to act. |
| <strong>Life Cycle Testing</strong> | The process of verifying the consistency, completeness, and correctness of software at each stage of the development life cycle. |
| <strong>Management</strong> | A team or individuals who manage(s) resources at any level of the organization. |
| <strong>Mapping</strong> | Provides a picture of the use of instructions during the execution of a program. Specifically, it provides a frequency listing of source code statements showing both the number of times an instruction was executed and which instructions were not executed. Mapping can be used to optimize source code by identifying the frequently used instructions. It can also be used to determine unused code, which can demonstrate code, which has not been tested, code that is infrequently used, or code that is non-entrant. |
| <strong>Mean</strong> | A value derived by adding several quantities and dividing the sum by the number of these quantities. |
| <strong>Measures</strong> | A unit to determine the dimensions, quantity, or capacity (e.g., lines of code are a measure of software size). |
| <strong>Mentoring</strong> | Mentoring is helping or supporting an individual in a non-supervisory capacity. Mentors can be peers, subordinates, or superiors. What is important is that the mentor does not have a managerial relationship to the mentored individual when performing the task of mentoring. |
| <strong>Metric</strong> | A software metric is a mathematical number that shows a relationship between two measures. |
| <strong>Metric-Based Test Data Generation</strong> | The process of generating test sets for structural testing based on use of complexity or coverage metrics. |
| <strong>Mission</strong> | A customer-oriented statement of purpose for a unit or a team. |
| <strong>Model Animation</strong> | Model animation verifies that early models can handle the various types of events found in production data. This is verified by “running” actual production transactions through the models as if they were operational systems. |
| <strong>Model Balancing</strong> | Model balancing relies on the complementary relationships between the various models used in structured analysis (event, process, data) to ensure that modeling rules/standards have been followed; this ensures that these complementary views are consistent and complete. |
| <strong>Model-Based Testing</strong> | Test cases are based on a simple model of the application. Generally, models are used to represent the desired behavior of the application being tested. The behavioral model of the application is derived from the application requirements and specification. |
| <strong>Moderator</strong> | Manages the inspection process, is accountable for the effectiveness of the inspection, and must be impartial. |
| <strong>Modified Condition Decision Coverage</strong> | A compromise which requires fewer test cases than Branch Condition Combination Coverage. |
| <strong>Motivation</strong> | Getting individuals to do work tasks they do not want to do or to perform those work tasks in a more efficient or effective manner. |
| <strong>Mutation Analysis</strong> | A method to determine test set thoroughness by measuring the extent to which a test set can discriminate the program from slight variants (i.e., mutants) of it. |
| <strong>Network Analyzers</strong> | A tool used to assist in detecting and diagnosing network problems. |</p>
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-functional Testing</td>
<td>Non-functional testing validates that the system quality attributes and characteristics have been considered during the development process. Non-functional testing is the testing of a software application for its non-functional requirements.</td>
</tr>
<tr>
<td>Objective Measures</td>
<td>An objective measure is a measure that can be obtained by counting.</td>
</tr>
<tr>
<td>Open Source</td>
<td>Open Source: “pertaining to or denoting software whose source code is available free of charge to the public to use, copy, modify, sublicense, or distribute.”</td>
</tr>
<tr>
<td>Optimum Point of Testing</td>
<td>The point where the value received from testing no longer exceeds the cost of testing.</td>
</tr>
<tr>
<td>Oracle</td>
<td>A (typically automated) mechanism or principle by which a problem in the software can be recognized. For example, automated test oracles have value in load testing software (by signing on to an application with hundreds or thousands of instances simultaneously), or in checking for intermittent errors in software.</td>
</tr>
<tr>
<td>Outputs</td>
<td>Products, services, or information supplied to meet customer needs.</td>
</tr>
<tr>
<td>Pair-Wise</td>
<td>Pair-wise testing (also known as all-pairs testing) is a combinatorial method used to generate the least number of test cases necessary to test each pair of input parameters to a system.</td>
</tr>
<tr>
<td>Parametric Modeling</td>
<td>A mathematical model based on known parameters to predict cost/schedule of a test project. The parameters in the model can vary based on the type of project.</td>
</tr>
<tr>
<td>Pareto Analysis</td>
<td>The Pareto Principle states that only a “vital few” factors are responsible for producing most of the problems. This principle can be applied to risk analysis to the extent that a great majority of problems (80%) are produced by a few causes (20%). If we correct these few key causes, we will have a greater probability of success.</td>
</tr>
<tr>
<td>Pareto Charts</td>
<td>A special type of bar chart to view the causes of a problem in order of severity: largest to smallest based on the 80/20 premise.</td>
</tr>
<tr>
<td>Pass/Fail Criteria</td>
<td>Decision rules used to determine whether a software item or feature passes or fails a test.</td>
</tr>
<tr>
<td>Passive Risk</td>
<td>Passive Risk is that which is inherent in inaction. For example, the choice not to update an existing product to compete with others in the marketplace.</td>
</tr>
</tbody>
</table>
Path Expressions A sequence of edges from the program graph that represents a path through the program.

Path Testing A test method satisfying the coverage criteria that each logical path through the program be tested. Often, paths through the program are grouped into a finite set of classes and one path from each class is tested.

Performance Test Validates that both the online response time and batch run times meet the defined performance requirements.

Performance/Timing Analyzer A tool to measure system performance.

Phase (or Stage) Containment A method of control put in place within each stage of the development process to promote error identification and resolution so that defects are not propagated downstream to subsequent stages of the development process.

Plan Define your objective and determine the conditions and methods required to achieve your objective. Clearly describe the goals and policies needed to achieve the objective at this stage. (Plan-Do-Check-Act)

Plan-Do-Check-Act model One of the best known process improvement models is the Plan-Do-Check-Act model for continuous process improvement.

Planning Poker In Agile Development, Planning Poker is a consensus-based technique designed to remove the cognitive bias of anchoring.

Policy Managerial desires and intents concerning either process (intended objectives) or products (desired attributes).

Population Analysis Analyzes production data to identify, independent from the specifications, the types and frequency of data that the system will have to process/produce. This verifies that the specs can handle types and frequency of actual data and can be used to create validation tests.

Post Conditions A list of conditions, if any, which will be true after the Use Case finished successfully.

Pre-Conditions A list of conditions, if any, which must be met before the Use Case can be properly executed.

Prevention Costs Resources required to prevent defects and to do the job right the first time. These normally require up-front expenditures for benefits that will be derived later. This category includes money spent on establishing methods and procedures, training workers, acquiring tools, and planning for quality. Prevention resources are spent before the product is actually built.
Preventive Controls
Preventive controls will stop incorrect processing from occurring.

Problem-Solving
Cooperative mode – Attempts to satisfy the interests of both parties. In terms of process, this is generally accomplished through identification of “interests” and freeing the process from initial positions. Once interests are identified, the process moves into a phase of generating creative alternatives designed to satisfy identified interests (criteria).

Procedure
Describe how work must be done and how methods, tools, techniques, and people are applied to perform a process. There are Do procedures and Check procedures. Procedures indicate the “best way” to meet standards.

Process
The process or set of processes used by an organization or project to plan, manage, execute, monitor, control, and improve its software related activities. A set of activities and tasks. A statement of purpose and an essential set of practices (activities) that address that purpose.

Process Improvement
To change a process to make the process produce a given product faster, more economically, or of higher quality.

Process Risk
Process risk is the activities such as planning, resourcing, tracking, quality assurance, and configuration management.

Producer/Author
Gathers and distributes materials, provides product overview, is available for clarification, should contribute as an inspector, and must not be defensive.

Producer’s View of Quality
Meeting requirements.

Product
The output of a process: the work product. There are three useful classes of products: Manufactured Products (standard and custom), Administrative/Information Products (invoices, letters, etc.), and Service Products (physical, intellectual, physiological, and psychological). A statement of requirements defines products; one or more people working in a process produce them.

Production Costs
The cost of producing a product. Production costs, as currently reported, consist of (at least) two parts; actual production or right-the-first time costs (RFT) plus the Cost of Quality (COQ). RFT costs include labor, materials, and equipment needed to provide the product correctly the first time.

Productivity
The ratio of the output of a process to the input, usually measured in the same units. It is frequently useful to compare the value added to a product by a process, to the value of the input
<table>
<thead>
<tr>
<th><strong>Proof of Correctness</strong></th>
<th>The use of mathematical logic techniques to show that a relationship between program variables assumed true at program entry implies that another relationship between program variables holds at program exit.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Quality</strong></td>
<td>A product is a quality product if it is defect free. To the producer, a product is a quality product if it meets or conforms to the statement of requirements that defines the product. This statement is usually shortened to: quality means meets requirements. From a customer’s perspective, quality means “fit for use.”</td>
</tr>
<tr>
<td><strong>Quality Assurance (QA)</strong></td>
<td>The set of support activities (including facilitation, training, measurement, and analysis) needed to provide adequate confidence that processes are established and continuously improved to produce products that meet specifications and are fit for use.</td>
</tr>
<tr>
<td><strong>Quality Control (QC)</strong></td>
<td>The process by which product quality is compared with applicable standards, and the action taken when nonconformance is detected. Its focus is defect detection and removal. This is a line function; that is, the performance of these tasks is the responsibility of the people working within the process.</td>
</tr>
<tr>
<td><strong>Quality Improvement</strong></td>
<td>To change a production process so that the rate at which defective products (defects) are produced is reduced.</td>
</tr>
<tr>
<td><strong>RAD Model</strong></td>
<td>A variant of prototyping, is another form of iterative development. The RAD model is designed to build and deliver application prototypes to the client while in the iterative process.</td>
</tr>
</tbody>
</table>
Reader (Inspections)  Must understand the material, paraphrases the material during the inspection, and sets the inspection pace.

Recorder (Inspections)  Must understand error classification, is not the meeting stenographer (captures enough detail for the project team to go forward to resolve errors), classifies errors as detected, and reviews the error list at the end of the meeting.

Recovery Test  Evaluates the contingency features built into the application for handling interruptions and for returning to specific points in the application processing cycle, including checkpoints, backups, restores, and restarts. This test also assures that disaster recovery is possible.

Regression Analysis  A means of showing the relationship between two variables. Regression analysis will provide two pieces of information. The first is a graphic showing the relationship between two variables. Second, it will show the correlation, or how closely related the two variables are.

Regression Testing  Testing of a previously verified program or application following program modification for extension or correction to ensure no new defects have been introduced.

Reliability  This refers to the consistency of measurement. Two different individuals take the same measurement and get the same result. The measure is reliable.

Requirement  A formal statement of:
1. An attribute to be possessed by the product or a function to be performed by the product
2. The performance standard for the attribute or function; and/or
3. The measuring process to be used in verifying that the standard has been met.

Requirements-Based Testing  RBT focuses on the quality of the Requirements Specification and requires testing throughout the development life cycle. Specifically, RBT performs static tests with the purpose of verifying that the requirements meet acceptable standards defined as: complete, correct, precise, unambiguous, and clear, consistent, relevant, testable, and traceable.

Residual Risk  Residual Risk is the risk that remains after management responds to the identified risks.

Reuse Model  The premise behind the Reuse Model is that systems should be built using existing components, as opposed to custom-building new components. The Reuse Model is clearly suited to Object-
Oriented computing environments, which have become one of the
premiere technologies in today's system development industry.

**Risk**
Risk can be measured by performing risk analysis.

**Risk Acceptance**
Risk Acceptance is the amount of risk exposure that is acceptable
to the project and the company and can be either active or passive.

**Risk Analysis**
Risk Analysis is an analysis of an organization's information
resources, its existing controls, and its organization and computer
system or application system vulnerabilities. It combines the loss
potential for each resource or combination of resources with an
estimated rate of occurrence to establish a potential level of
damage in dollars or other assets.

**Risk Appetite**
Risk Appetite defines the amount of loss management is willing to
accept for a given risk.

**Risk Assessment**
Risk Assessment is an examination of a project to identify areas of
potential risk. The assessment can be broken down into analysis,
identification, and prioritization.

**Risk Avoidance**
Risk avoidance is a strategy for risk resolution to eliminate the risk
altogether. Avoidance is a strategy to use when a lose-lose
situation is likely.

**Risk Event**
Risk Event is a future occurrence that may affect the project for
better or worse. The positive aspect is that these events will help
you identify opportunities for improvement while the negative
aspect will be the realization of threats and losses.

**Risk Exposure**
Risk Exposure is the measure that determines the probability of
likelihood of the event times the loss that could occur.

**Risk Identification**
Risk Identification is a method used to find risks before they
become problems. The risk identification process transforms
issues and concerns about a project into tangible risks, which can
be described and measured.

**Risk Leverage**
Risk leverage is a measure of the relative cost-benefit of
performing various candidate risk resolution activities.

**Risk Management**
Risk Management is the process required to identify, quantify,
respond to, and control project, process, and product risk.

**Risk Mitigation**
Risk Mitigation is the action taken to reduce threats and/or
vulnerabilities.

**Risk Protection**
Risk protection is a strategy to employ redundancy to mitigate
(reduce the probability and/or consequence of) a risk.
<p>| <strong>Risk Reduction</strong> | Risk reduction is a strategy to decrease risk through mitigation, prevention, or anticipation. Decreasing either the probability of the risk occurrence or the consequence when the risk is realized reduces risk. Reduction is a strategy to use when risk leverage exists. |
| <strong>Risk Reserves</strong> | A risk reserve is a strategy to use contingency funds and built-in schedule slack when uncertainty exists in cost or time. |
| <strong>Risk Transfer</strong> | Risk transfer is a strategy to shift the risk to another person, group, or organization and is used when another group has control. |
| <strong>Risk-Based Testing</strong> | Risk-based testing prioritizes the features and functions to be tested based on the likelihood of failure and the impact of a failure should it occur. |
| <strong>Run Chart</strong> | A run chart is a graph of data (observation) in chronological order displaying shifts or trends in the central tendency (average). The data represents measures, counts or percentages of outputs from a process (products or services). |
| <strong>Sad Path</strong> | A path through the application which does not arrive at the desired result. |
| <strong>Scatter Plot Diagram</strong> | A graph designed to show whether there is a relationship between two changing variables. |
| <strong>Scenario Testing</strong> | Testing based on a real-world scenario of how the system is supposed to act. |
| <strong>Scope of Testing</strong> | The scope of testing is the extensiveness of the test process. A narrow scope may be limited to determining whether or not the software specifications were correctly implemented. The scope broadens as more responsibilities are assigned to software testers. |
| <strong>Selective Regression Testing</strong> | The process of testing only those sections of a program where the tester’s analysis indicates programming changes have taken place and the related components. |
| <strong>Self-Validating Code</strong> | Code that makes an explicit attempt to determine its own correctness and to proceed accordingly. |
| <strong>SLOC</strong> | Source Lines of Code |
| <strong>Smoothing</strong> | An unassertive approach – Both parties neglect the concerns involved by sidestepping the issue, postponing the conflict, or choosing not to deal with it. |
| <strong>Soft Skills</strong> | Soft skills are defined as the personal attributes which enable an individual to interact effectively and harmoniously with other people. |</p>
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Software Feature</strong></td>
<td>A distinguishing characteristic of a software item (e.g., performance, portability, or functionality).</td>
</tr>
<tr>
<td><strong>Software Item</strong></td>
<td>Source code, object code, job control code, control data, or a collection of these.</td>
</tr>
<tr>
<td><strong>Software Quality Criteria</strong></td>
<td>An attribute of a quality factor that is related to software development.</td>
</tr>
<tr>
<td><strong>Software Quality Factors</strong></td>
<td>Software quality factors are attributes of the software that, if they are wanted and not present, pose a risk to the success of the software and thus constitute a business risk.</td>
</tr>
<tr>
<td><strong>Software Quality Gaps</strong></td>
<td>The first gap is the producer gap. It is the gap between what was specified to be delivered, meaning the documented requirements and internal IT standards, and what was actually delivered. The second gap is between what the producer actually delivered compared to what the customer expected.</td>
</tr>
<tr>
<td><strong>Special Causes of Variation</strong></td>
<td>Variation not typically present in the process. They occur because of special or unique circumstances.</td>
</tr>
<tr>
<td><strong>Special Test Data</strong></td>
<td>Test data based on input values that are likely to require special handling by the program.</td>
</tr>
<tr>
<td><strong>Spiral Model</strong></td>
<td>Model designed to include the best features from the Waterfall and Prototyping, and introduces a new component risk-assessment.</td>
</tr>
<tr>
<td><strong>Standardize</strong></td>
<td>Procedures that are implemented to ensure that the output of a process is maintained at a desired level.</td>
</tr>
<tr>
<td><strong>Standardizer</strong></td>
<td>Must know IT standards &amp; procedures, ensures standards are met and procedures are followed, meets with project leader/manager, and ensures entrance criteria are met (product is ready for review).</td>
</tr>
<tr>
<td><strong>Standards</strong></td>
<td>The measure used to evaluate products and identify nonconformance. The basis upon which adherence to policies is measured.</td>
</tr>
<tr>
<td><strong>Statement of Requirements</strong></td>
<td>The exhaustive list of requirements that define a product. Note that the statement of requirements should document requirements proposed and rejected (including the reason for the rejection) during the requirement determination process.</td>
</tr>
<tr>
<td><strong>Statement Testing</strong></td>
<td>A test method that executes each statement in a program at least once during program testing.</td>
</tr>
<tr>
<td><strong>Static Analysis</strong></td>
<td>Analysis of a program that is performed without executing the program. It may be applied to the requirements, design, or code.</td>
</tr>
<tr>
<td><strong>Statistical Process Control</strong></td>
<td>The use of statistical techniques and tools to measure an ongoing process for change or stability.</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Story Points</strong></td>
<td>Measurement of a feature’s size relative to other features. Story Points are an analogous method in that the objective is to compare the sizes of features to other stories and reference stories.</td>
</tr>
<tr>
<td><strong>Stress Testing</strong></td>
<td>This test subjects a system, or components of a system, to varying environmental conditions that defy normal expectations. For example, high transaction volume, large database size or restart/recovery circumstances. The intention of stress testing is to identify constraints and to ensure that there are no performance problems.</td>
</tr>
<tr>
<td><strong>Structural Analysis</strong></td>
<td>Structural analysis is a technique used by developers to define unit test cases. Structural analysis usually involves path and condition coverage.</td>
</tr>
<tr>
<td><strong>Structural System Testing</strong></td>
<td>Structural System Testing is designed to verify that the developed system and programs work. The objective is to ensure that the product designed is structurally sound and will function correctly.</td>
</tr>
<tr>
<td><strong>Structural Testing</strong></td>
<td>A testing method in which the test data is derived solely from the program structure.</td>
</tr>
<tr>
<td><strong>Stub</strong></td>
<td>Special code segments that when invoked by a code segment under testing, simulate the behavior of designed and specified modules not yet constructed.</td>
</tr>
<tr>
<td><strong>Subjective Measures</strong></td>
<td>A person’s perception of a product or activity.</td>
</tr>
<tr>
<td><strong>Supplier</strong></td>
<td>An individual or organization that supplies inputs needed to generate a product, service, or information to a customer.</td>
</tr>
<tr>
<td><strong>System Boundary Diagram</strong></td>
<td>A system boundary diagram depicts the interfaces between the software under test and the individuals, systems, and other interfaces. These interfaces or external agents are referred to as “actors.” The purpose of the system boundary diagram is to establish the scope of the system and to identify the actors (i.e., the interfaces) that need to be developed. (Use Cases)</td>
</tr>
<tr>
<td><strong>System Test</strong></td>
<td>The entire system is tested to verify that all functional, information, structural and quality requirements have been met. A predetermined combination of tests is designed that, when executed successfully, satisfy management that the system meets specifications. System testing verifies the functional quality of the system in addition to all external interfaces, manual procedures, restart and recovery, and human-computer interfaces. It also verifies that interfaces between the application and the open environment work correctly, that JCL functions correctly, and that the application functions appropriately with the Database</td>
</tr>
</tbody>
</table>
Management System, Operations environment, and any communications systems.

**Test**
1. A set of one or more test cases.
2. A set of one or more test cases and procedures.

**Test Case Generator**
A software tool that creates test cases from requirements specifications. Cases generated this way ensure that 100% of the functionality specified is tested.

**Test Case Specification**
An individual test condition, executed as part of a larger test that contributes to the test's objectives. Test cases document the input, expected results, and execution conditions of a given test item. Test cases are broken down into one or more detailed test scripts and test data conditions for execution.

**Test Cycle**
Test cases are grouped into manageable (and schedulable) units called test cycles. Grouping is according to the relation of objectives to one another, timing requirements, and on the best way to expedite defect detection during the testing event. Often test cycles are linked with execution of a batch process.

**Test Data**
Data points required to test most applications; one set of test data to confirm the expected results (data along the happy path), a second set to verify the software behaves correctly for invalid input data (alternate paths or sad path), and finally data intended to force incorrect processing (e.g., crash the application).

**Test Data Management**
A defined strategy for the development, use, maintenance, and ultimately destruction of test data.

**Test Data Set**
Set of input elements used in the testing process.

**Test Design Specification**
A document that specifies the details of the test approach for a software feature or a combination of features and identifies the associated tests.

**Test Driver**
A program that directs the execution of another program against a collection of test data sets. Usually, the test driver also records and organizes the output generated as the tests are run.

**Test Environment**
The Test Environment can be defined as a collection of hardware and software components configured in such a way as to closely mirror the production environment. The Test Environment must replicate or simulate the actual production environment as closely as possible.

**Test Harness**
A collection of test drivers and test stubs.

**Test Incident Report**
A document describing any event during the testing process that requires investigation.
<table>
<thead>
<tr>
<th><strong>Test Item</strong></th>
<th>A software item that is an object of testing.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Test Item</strong></td>
<td>A document that identifies test items and includes status and location information.</td>
</tr>
<tr>
<td><strong>Transmittal Report</strong></td>
<td>Test labs are another manifestation of the test environment which is more typically viewed as a brick and mortar environment (designated, separated, physical location).</td>
</tr>
<tr>
<td><strong>Test Labs</strong></td>
<td>A chronological record of relevant details about the execution of tests.</td>
</tr>
<tr>
<td><strong>Test Log</strong></td>
<td>A document describing the intended scope, approach, resources, and schedule of testing activities. It identifies test items, the features to be tested, the testing tasks, the personnel performing each task, and any risks requiring contingency planning.</td>
</tr>
<tr>
<td><strong>Test Point Analysis (TPA)</strong></td>
<td>Calculates test effort based on size (derived from FPA), strategy (as defined by system components and quality characteristics to be tested and the coverage of testing), and productivity (the amount of time needed to perform a given volume of testing work).</td>
</tr>
<tr>
<td><strong>Test Procedure Specification</strong></td>
<td>A document specifying a sequence of actions for the execution of a test.</td>
</tr>
<tr>
<td><strong>Test Scripts</strong></td>
<td>A specific order of actions that should be performed during a test session. The script also contains expected results. Test scripts may be manually prepared using paper forms, or may be automated using capture/playback tools or other kinds of automated scripting tools.</td>
</tr>
<tr>
<td><strong>Test Stubs</strong></td>
<td>Simulates a called routine so that the calling routine's functions can be tested. A test harness (or driver) simulates a calling component or external environment, providing input to the called routine, initiating the routine, and evaluating or displaying output returned.</td>
</tr>
<tr>
<td><strong>Test Suite Manager</strong></td>
<td>A tool that allows testers to organize test scripts by function or other grouping.</td>
</tr>
<tr>
<td><strong>Test Summary Report</strong></td>
<td>A document that describes testing activities and results and evaluates the corresponding test items.</td>
</tr>
<tr>
<td><strong>Testing</strong></td>
<td>1. The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.</td>
</tr>
<tr>
<td></td>
<td>2. The process of analyzing a software item to detect the differences between existing and required conditions, i.e. bugs, and to evaluate the features of the software items. See: dynamic analysis, static analysis, software engineering.</td>
</tr>
</tbody>
</table>
**Testing Process Assessment**

Thoughtful analysis of testing process results, and then taking corrective action on the identified weaknesses.

**Testing Schools of Thought**

A school of thought simply defined as “a belief (or system of beliefs) shared by a group.” Generally accepted test schools of that are: the Analytical School, the Factory School, the Quality (control) School, Context-driven School, and the Agile School.

**Therapeutic Listening**

The listener is sympathetic to the speaker's point of view. During this type of listening, the listener will show a lot of empathy for the speaker's situation.

**Thread Testing**

This test technique, which is often used during early integration testing, demonstrates key functional capabilities by testing a string of units that accomplish a specific function in the application.

**Threat**

Threat is something capable of exploiting a vulnerability in the security of a computer system or application. Threats include both hazards (any source of potential damage or harm) and events that can trigger vulnerabilities.

**Threshold Values**

Threshold values define the inception of risk occurrence. Predefined thresholds act as a warning level to indicate the need to execute the risk action plan.

**Timeliness**

This refers to whether the data was reported in sufficient time to impact the decisions needed to manage effectively.

**TMMi**

A process improvement model for software testing. The Test Maturity Model integration (TMMi) is a detailed model for test process improvement and is positioned as being complementary to the CMMI.

**Tools**

Any resources that are not consumed in converting the input into the deliverable.

**Top-Down**

Begin testing from the top of the module hierarchy and work down to the bottom using interim stubs to simulate lower interfacing modules or programs. Modules are added in descending hierarchical order.

**Top-Down Estimation**

Generate an overall estimate based on initial knowledge. It is used at the initial stages of the project and is based on similar projects. Past data plays an important role in this form of estimation.

**Tracing**

A process that follows the flow of computer logic at execution time. Tracing demonstrates the sequence of instructions or a path followed in accomplishing a given task. The two main types of trace are tracing instructions in computer programs as they are executed, or tracing the path through a database to locate predetermined pieces of information.
<table>
<thead>
<tr>
<th>Vocabulary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Triangulation</strong></td>
<td>Story Triangulation is a form of estimation by analogy. After the first few estimates have been made, they are verified by relating them to each other. (Agile methods)</td>
</tr>
<tr>
<td><strong>Triggers</strong></td>
<td>A device used to activate, deactivate, or suspend a risk action plan. Triggers can be set by the project tracking system.</td>
</tr>
<tr>
<td><strong>Use Case Points</strong></td>
<td>A derivative of the Use Cases method is the estimation technique known as Use Case Points. Use Case Points are similar to Function Points and are used to estimate the size of a project.</td>
</tr>
<tr>
<td><strong>Unit Test</strong></td>
<td>Testing individual programs, modules, or components to demonstrate that the work package executes per specification, and validate the design and technical quality of the application. The focus is on ensuring that the detailed logic within the component is accurate and reliable according to pre-determined specifications. Testing stubs or drivers may be used to simulate behavior of interfacing modules.</td>
</tr>
<tr>
<td><strong>Usability Test</strong></td>
<td>The purpose of this event is to review the application user interface and other human factors of the application with the people who will be using the application. This is to ensure that the design (layout and sequence, etc.) enables the business functions to be executed as easily and intuitively as possible. This review includes assuring that the user interface adheres to documented User Interface standards, and should be conducted early in the design stage of development. Ideally, an application prototype is used to walk the client group through various business scenarios, although paper copies of screens, windows, menus, and reports can be used.</td>
</tr>
<tr>
<td><strong>Use Case</strong></td>
<td>A Use Case is a technique for capturing the functional requirements of systems through the interaction between an Actor and the System.</td>
</tr>
<tr>
<td><strong>User</strong></td>
<td>The customer that actually uses the product received.</td>
</tr>
<tr>
<td><strong>User Acceptance Testing</strong></td>
<td>User Acceptance Testing (UAT) is conducted to ensure that the system meets the needs of the organization and the end user/customer. It validates that the system will work as intended by the user in the real world, and is based on real world business scenarios, not system requirements. Essentially, this test validates that the right system was built.</td>
</tr>
<tr>
<td></td>
<td>The testing of a computer system or parts of a computer system to make sure it will solve the customer's problem regardless of what the system requirements indicate.</td>
</tr>
<tr>
<td><strong>User Story</strong></td>
<td>A short description of something that a customer will do when they use an application (software system). The User Story is focused on the value or result a customer would receive from doing whatever it is the application does.</td>
</tr>
</tbody>
</table>
Validation
Validation physically ensures that the system operates according to the desired specifications by executing the system functions through a series of tests that can be observed and evaluated.

Validity
This indicates the degree to which a measure actually measures what it was intended to measure.

Values (Sociology)
The ideals, customs, instructions, etc., of a society toward which the people have an affective regard. These values may be positive, as cleanliness, freedom, or education, or negative, as cruelty, crime, or blasphemy. Any object or quality desired as a means or as an end in itself.

Verification
The process of determining whether the products of a given phase of the software development cycle fulfill the requirements established during the previous phase. The act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether items, processes, services, or documents conform to specified requirements.

Virtualization
The concept of virtualization (within the IT space) usually refers to running multiple operating systems on a single machine.

Vision
A vision is a statement that describes the desired future state of a unit.

V-Model
The V-Model is considered an extension of the Waterfall Model. The purpose of the “V” shape is to demonstrate the relationships between each phase of specification development and its associated dynamic testing phase.

Vulnerability
Vulnerability is a design, implementation, or operations flaw that may be exploited by a threat. The flaw causes the computer system or application to operate in a fashion different from its published specifications and results in destruction or misuse of equipment or data.

Walkthroughs
An informal review (static) testing process in which the author “walks through” the deliverable with the review team looking for defects.

Waterfall
A development model in which progress is seen as flowing steadily downwards through the phases of conception, initiation, requirements, design, construction, dynamic testing, production/implementation, and maintenance.

WBS
A Work Breakdown Structure (WBS) groups project components into deliverable and accountable pieces.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>White-Box Testing</td>
<td>A testing technique that assumes that the path of the logic in a program unit or component is known. White-box testing usually consists of testing paths, branch by branch, to produce predictable results. This technique is usually used during tests executed by the development team, such as Unit or Component testing.</td>
</tr>
<tr>
<td>Wideband Delphi</td>
<td>A method for the controlled exchange of information within a group. It provides a formal, structured procedure for the exchange of opinion, which means that it can be used for estimating.</td>
</tr>
<tr>
<td>Withdrawal</td>
<td>Conflict is resolved when one party attempts to satisfy the concerns of others by neglecting its own interests or goals. This is a lose-win approach.</td>
</tr>
<tr>
<td>Workbench</td>
<td>The objective of the workbench is to produce the defined output products (deliverables) in a defect-free manner. The procedures and standards established for each workbench are designed to assist in this objective.</td>
</tr>
</tbody>
</table>
Test Plan Example

I

HOPE MATE VERSION 3.5
SYSTEM TEST PLAN
CHICAGO HOPE
DEPARTMENT OF
INFORMATION
TECHNOLOGY (DOIT)
Abstract

This document describes the proposed approach to be taken by Chicago Hope DOIT staff to test the HopeMate 3.5 system update.
<table>
<thead>
<tr>
<th>Version</th>
<th>Modified By</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>M. Jones</td>
<td>04/11/xx</td>
<td>Draft</td>
</tr>
<tr>
<td>0.2</td>
<td>M. Jones</td>
<td>04/13/xx</td>
<td>Minor Corrections</td>
</tr>
<tr>
<td>0.3</td>
<td>M. Jones</td>
<td>04/22/xx</td>
<td>Post Review by Test Manager</td>
</tr>
<tr>
<td>0.4</td>
<td>M. Jones</td>
<td>05/04/xx</td>
<td>Post Review by Projects Office</td>
</tr>
</tbody>
</table>
# Table of Contents

## DOCUMENT CONTROL .................................................................2
- Control ..........................................................................................2
- Abstract .......................................................................................2
- Distribution ...................................................................................2
- History ..........................................................................................2

## TABLE OF CONTENTS .....................................................................3

## 1. GENERAL INFORMATION .........................................................5
1.1. DEFINITIONS ...............................................................................5
1.2 REFERENCES ...............................................................................6
   - 1.2.1 Document Map .......................................................................6
   - 1.2.2 Test Directory Location ..........................................................6
   - 1.2.3 Additional Reference Documents ...........................................6
1.3 KEY CONTRIBUTORS .......................................................................6
1.4 HUMAN RESOURCES REQUIRED ...............................................7
   - 1.4.1 Test Resource Persons ............................................................7
   - 1.4.2 Test Team .............................................................................7
1.5 TEST ENVIRONMENT RESOURCES .............................................7
   - 1.5.1 Test Environment ..................................................................7
   - 1.5.2 Hardware Components ...........................................................8
   - 1.5.3 Software Components .............................................................8
1.6 RESOURCE BUDGETING ............................................................8
1.7 SCHEDULE ..................................................................................8

## 2. TEST OBJECTIVES AND SCOPE ..............................................9
2.1 PURPOSE OF THIS DOCUMENT ................................................9
   - 2.1.1 Summary of document contents: ...........................................9
   - 2.1.2 Document Outputs .................................................................9
2.2 OBJECTIVES OF SYSTEM TEST ................................................9
   - 2.2.1 Method of achieving the Objectives ......................................9
2.3 SCOPE OF SYSTEM TEST ........................................................10
   - 2.3.1 Inclusions ...........................................................................10
   - 2.3.2 Exclusions ...........................................................................10
   - 2.3.3 Specific Exclusions ...............................................................10
2.4 DETAILED SYSTEM TEST SCOPE ..........................................11
   - 2.4.1 Access Management Upgrade ..............................................11
   - 2.4.2 Ambulatory ..........................................................................11
2.4.3 Clinical Documentation ................................................................. 11
2.4.4 Emergency Document ................................................................. 12
2.4.5 Infrastructure ............................................................................. 12
2.4.6 Portal/Mobility/SHM ................................................................. 12

3. TEST STRATEGY .............................................................................. 13
3.1. OVERALL STRATEGY ................................................................. 13
3.2. PROPOSED SOFTWARE “DROPS” SCHEDULE ......................... 13
3.3. TESTING PROCESS ................................................................. 14
3.4 INSPECTING RESULTS ............................................................. 14
3.5. SYSTEM TESTING ................................................................. 15
3.5.1 Test Environment Pre-Test Criteria ......................................... 15
3.5.2 Resumption Criteria .............................................................. 15
3.6. EXIT CRITERIA ................................................................. 15
3.7. DAILY TESTING STRATEGY .................................................... 15
3.8 TEST CASES OVERVIEW ......................................................... 16
3.9. SYSTEM TESTING CYCLES ...................................................... 16

4. MANAGEMENT PROCEDURES ................................................... 17
4.1. ERROR TRACKING & REPORTING ............................................. 17
4.2. ERROR MANAGEMENT ............................................................. 17
4.2.1. Error Status Flow ............................................................... 17
4.3. ERROR CLASSIFICATION ......................................................... 18
4.3.1. Explanation of Classifications ............................................. 18
4.3.2. Error Turnaround Time ....................................................... 18
4.3.3. Re-release of Software ....................................................... 19
4.3.4 Build Release Procedures .................................................... 19
4.4. ERROR TRACKING, REPORTING & UPDATING ...................... 19
4.4.1. Purpose of Error Review Team ........................................... 20
4.4.2. Error Review Team Meeting Agenda ................................. 20
4.5. PROGRESS REPORTING ......................................................... 20
4.6. RETEST AND REGRESSION TESTING ...................................... 21

5. TEST PREPARATION AND EXECUTION .................................... 22
5.1 PROCESSES ................................................................. 22
5.2 FORMAL REVIEWING ............................................................. 22
5.2.1 Formal Review Points ........................................................... 22

6. RISKS & DEPENDENCIES ......................................................... 23
6.1. RISKS ..................................................................................... 23
6.2. DEPENDENCIES ................................................................. 23
6.2.1 General Dependencies

7. SIGNOFF
1. General Information

1.1. Definitions:

The following definitions apply to this document.

**Testing Software** is operating the software under controlled conditions, to (1) verify that it behaves “as specified”; (2) to detect errors, and (3) to check that what has been specified is what the user actually wanted.

**System testing** – black-box type testing that is based on overall requirements specifications; covers all combined parts of a system (see ‘Black Box’ testing).

**Unit testing** – the most ‘micro’ scale of testing; to test particular functions or code modules. Typically done by the programmer and not by testers, as it requires detailed knowledge of the internal program design and code. (see ‘White Box’ testing)

**Regression testing** – re-testing after fixes or modifications of the software or its environment. It can be difficult to determine how much re-testing is needed, especially near the end of the development cycle. Automated testing tools can be especially useful for this type of testing.

**Load testing** – testing an application under heavy loads, such as testing of a web System under a range of loads to determine at what point the system’s response time degrades or fails.

**Stress testing** – term often used interchangeably with ‘load’ and ‘performance’ testing. Also used to describe such tests as system functional testing while under unusually heavy loads, heavy repetition of certain actions or inputs, input of large numerical values, large complex queries to a database system, etc.

**Performance testing** – term often used interchangeably with ‘stress’ and ‘load’ testing. Ideally ‘performance’ testing (and any other ‘type’ of testing) is defined in requirements documentation or QA or Test Plans.

**Security testing** – testing how well the system protects against unauthorized internal or external access, wilful damage, etc; may require sophisticated testing techniques.

**Black box testing** is testing the application without any specific knowledge of internal design or code. Tests are based on requirements and functionality, whereby a specific input produces an output (result), and the output is checked against a pre-specified expected result.
**White box testing** – based on knowledge of the internal logic of an application’s code. Tests are based on coverage of code statements, branches, paths, conditions. Unit Testing is generally of this nature, where the programmer knows what an individual piece of code is supposed to do and thus can check it step by step.

**Soak Testing** is testing the stability and ‘uptime’ of the integrated software application.

**Integration testing** – testing of combined parts of an application to determine if they function together correctly. The ‘parts’ can be code modules, individual applications, client and server applications on a network, etc. This type of testing is especially relevant to client/server and distributed systems.

**Installation Testing** is testing that the application can be installed correctly on various different types of browsers and operating systems.

**Functional Testing** – see ‘System Testing’

### 1.2 References

#### 1.2.1 Document Map:

This section contains the document Map of the HopeMate 3.5 Project System Test documentation.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Originator / Owner</th>
<th>Location / Link</th>
</tr>
</thead>
</table>

#### 1.2.2 Chicago Hope DOIT Test Directory Location:

\`\`\Server1\Data\QA\HopeMate35\`

#### 1.2.3 Additional Reference Documents

<table>
<thead>
<tr>
<th>Reference</th>
<th>Originator / Owner</th>
<th>Location / Link</th>
</tr>
</thead>
</table>
1.3 Key Contributions

List key contributors to this document. These may include clients, developers, QA, configuration managers, test analysts.

<table>
<thead>
<tr>
<th>Role</th>
<th>Name</th>
<th>Resource Type</th>
<th>Resource Title</th>
<th>No.</th>
<th>Date Required</th>
<th>Duration</th>
<th>Who</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Support Team</td>
<td>Operations Support</td>
<td>1</td>
<td>mm/dd/yy</td>
<td></td>
<td>Ongoing</td>
<td>Ongoing</td>
<td>Persons Name</td>
<td>Assigned</td>
</tr>
<tr>
<td>Development Team</td>
<td>Development Team Contact</td>
<td>1</td>
<td>mm/dd/yy</td>
<td></td>
<td>Ongoing</td>
<td>Ongoing</td>
<td>Persons Name</td>
<td>Assigned</td>
</tr>
<tr>
<td>Project Management</td>
<td>High level BMC Contact</td>
<td>1</td>
<td>Ongoing</td>
<td></td>
<td>Ongoing</td>
<td>Ongoing</td>
<td>Persons Name</td>
<td>Assigned</td>
</tr>
<tr>
<td>Project Management</td>
<td>System Test Manager</td>
<td>1</td>
<td>Ongoing</td>
<td></td>
<td>Ongoing</td>
<td>Ongoing</td>
<td>Persons Name</td>
<td>Assigned</td>
</tr>
</tbody>
</table>

1.4 Human Resources Required

1.4.1 Test Resource Persons
1.4.2 Test Team

<table>
<thead>
<tr>
<th>Technical Support</th>
<th>Middleware Support</th>
<th>1</th>
<th>mm/dd/yy</th>
<th>Ongoing</th>
<th>Persons Name</th>
<th>Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technical Support</td>
<td>Database Support [DBA]</td>
<td>1</td>
<td>mm/dd/yy</td>
<td>Ongoing</td>
<td>Persons Name</td>
<td>Assigned</td>
</tr>
<tr>
<td>Test Support Team</td>
<td>Build Engineer</td>
<td>1</td>
<td>mm/dd/yy</td>
<td>Ongoing</td>
<td>Persons Name</td>
<td>Assigned</td>
</tr>
<tr>
<td>Design Team</td>
<td>Design Representative</td>
<td>1</td>
<td>mm/dd/yy</td>
<td>Ongoing</td>
<td>Persons Name</td>
<td>Assigned</td>
</tr>
<tr>
<td>UAT Team</td>
<td>UAT Representative</td>
<td>1</td>
<td>mm/dd/yy</td>
<td>Ongoing</td>
<td>Persons Name</td>
<td>Assigned</td>
</tr>
</tbody>
</table>

1.5 Test Environment Resources

1.5.1 Test Environment

A dedicated, separate test environment is required for System Testing. This environment will consist of the hardware and software components.

The environment that will be used is the MAIN test environment - http://11.22.33.44/

The test environment is described in detail in the test specification documentation - \\Server\Folder\QA\HopeMate\Test_Specification\TESTSPEC.doc

1.5.2 Hardware Components:
- 8 test pc's
- Oracle 8 Test Database
- Solaris 2.6 Server
1.5.3 Software Components:

<table>
<thead>
<tr>
<th>Software</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows x</td>
<td>2nd Release &amp; JVM Update</td>
</tr>
<tr>
<td>Windows XX Server</td>
<td>Service Pack x</td>
</tr>
<tr>
<td>Internet Explorer</td>
<td>x.03</td>
</tr>
<tr>
<td>Chrome</td>
<td>x.01</td>
</tr>
<tr>
<td>Safari</td>
<td>x.7</td>
</tr>
<tr>
<td>FireFox</td>
<td>x.61</td>
</tr>
<tr>
<td>Outlook 20xx</td>
<td>20xx</td>
</tr>
<tr>
<td>Microsoft Office 20xx</td>
<td>20xx Professional</td>
</tr>
<tr>
<td>Ghost</td>
<td>-</td>
</tr>
<tr>
<td>System Commander</td>
<td>Deluxe</td>
</tr>
</tbody>
</table>

The configuration of the individual test machines will be described in:

\Server\Folder\QA\HopeMate\Test Specification\Test Machine Configurations.doc

1.6 Resource Budgeting

<table>
<thead>
<tr>
<th>Resource</th>
<th>Estimated $$</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training for test analyst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testing Tools</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1.7 Schedule

Project Milestone Chart here
2. Test Objectives and Scope

2.1 Purpose of this document

The purpose of the document is to describe the approach to be used by Chicago Hope DOIT for testing the HopeMate 3.5 upgrade, the exact scope of which will be specified in section 2.3.

This document is a draft description, and as such is required to be reviewed before being accepted.

2.1.1. Summary of document contents:

1. The applications that will, and will not, be tested.
2. The types of tests that will be performed.
3. The scope of the overall system testing
4. The overall strategy to be adopted and the activities to be completed
5. The resources required
6. The methods and processes to be used to test the release.
7. The activities, dependencies and effort required to conduct the System Test.
8. Where test logs, summaries and reports will be stored.
9. When this work will be done
10. Who will do the work.

2.1.2. Document Outputs

The Test Specification is a separate document which follows on from this System Test Plan. The Test Specification document contains detailed test cases and automated test scripts to be applied, the data to be processed, the automated testing coverage & scripts, and the expected results.

2.2 Objectives of System Test

The objective is:

1. The verification and validation of the HopeMate 3.5 System Functionality.
2. The identification, tracking and reporting of issues and discrepancies discovered before or during system test
3. That the software satisfies the release criteria, as specified by Chicago Hope xxx Department.

2.2.1. Method of achieving the Objectives
The objectives will be achieved by defining, reviewing and executing detailed test cases, in which specified steps will be taken to test the software, the results of which will be compared against pre-specified expected results, to evaluate whether the test was successful. This can only happen in working directly with the business groups to define their requirements from the system.

2.3 Scope of System Test

2.3.1 Inclusions

The scope of the Chicago Hope DOIT testing consists of System Testing, as defined in Section 1: Definitions, of the HopeMate Project components only.

The contents of this release, which this test plan will cover, are as follows:

<table>
<thead>
<tr>
<th>Item</th>
<th>Delivery Date to System Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>D Access Management</td>
<td>9th May</td>
</tr>
<tr>
<td>A Ambulatory</td>
<td>3rd June</td>
</tr>
<tr>
<td>A Clinical Documentation</td>
<td>13th June</td>
</tr>
<tr>
<td>A Emergency Department</td>
<td>8th July</td>
</tr>
<tr>
<td>A Infrastructure</td>
<td>18th July</td>
</tr>
<tr>
<td>A Portal/Mobility/SHM</td>
<td>3rd August</td>
</tr>
<tr>
<td>A Live Defect Fixes [Severity A Defect fixes]</td>
<td>Ongoing</td>
</tr>
</tbody>
</table>

Detailed descriptions of the scope of each function will be described in section 2.4.

2.3.2. Exclusions

When the scope of the Test Plan has been agreed and signed off, no further inclusions will be considered for inclusion in this release, except:

1) Where there is the express permission and agreement of the Chicago Hope DOIT System Test Manager and the Test Team Leader; and

2) Where the changes/inclusions will not require significant effort on behalf of the test team (i.e. requiring extra preparation - new test cases etc.) and will not adversely affect the test schedule; or

3) The Chicago Hope DOIT System Test Manager explicitly agrees to accept any impact on the test schedule resulting from the changes/inclusions.

4) System testing means black-box testing - this means that testing will not include PL/SQL validation.
2.3.3. Specific Exclusions

- Severity A Errors since the code cut was taken (4th April) until the start of system test (8th May).
- Usability Testing (UT), is the responsibility of Chicago Hope xx Department.
- There will be no testing other than specified in the Test Specification document.
- No Hardware testing will be performed.
- Testing of the application on Linux is outside the scope of this plan.
- Testing of the application on Unix, including Solaris is outside the scope of this plan.
- Testing of the application on Windows XP Pro is outside the scope of this plan.
- Security Testing is outside the scope of this plan.
- Any other test configuration other than those specified in the test configurations document will not be tested.

2.4 DOIT Test Responsibilities

Chicago Hope DOIT shall be responsible for:

- System Testing
- Management of User Acceptance Testing
- Error Reporting and tracking
- Progress Reporting
- Post Test Analysis

2.4 Detailed System Test Scope

2.4.1. Access Management Upgrade

The Access Management upgrade system testing comprises system testing of the four main application sections:

- Basic Registration
- Enterprise Registration
- Interface and Conversion Tools
- Location Management
The changes to the application components consist of:

1) Defect fixes applied by J.S. Smith as specified in J.S. Smith’s document "Access Management Final Status".
2) Enhancements as specified in XYZsys’ “HopeMate 3.5 Feature List”.

2.4.2. Ambulatory

The Ambulatory system testing consists of:

1) Testing the changes made to xxx
2) Testing the changes made to yyy
3) Testing the changes made to zzz
4) Testing the changes made to the cgi-bin & perl scripts

Notes:
• Chicago Hope DOIT will initially test using sample files provided by XYZsys, and then liaise with XYZsys and Chicago Hope xx Department for further testing.
• Chicago Hope DOIT will check file formats against the Technical Specification v1.03.

2.4.3. Clinical Documentation

The Clinical Documentation consists of:

1) Testing the changes made aaa
2) Testing the changes made bbb
3) Testing the changes made ccc
4) Testing the changes made ddd

2.4.4. Emergency Department

The Emergency Department changes include:

• Updates to ED Status Board including enhancement xxx, yyy and zzz.
• Report updates

Note - Stress Testing & Performance criteria have not been specified for the Emergency
Department, thus performance/stress testing cannot, and will not, be performed by Chicago Hope DOIT as part of the HopeMate 3.5 System Test.

2.4.5. Infrastructure

The HopeMate 3.5 Infrastructure system testing comprises system testing of the security/application access. See section 2.4.6 for details:

2.4.6. Portal/Mobility/SHM

The HopeMate 3.5 Security system testing comprises testing of the two main internal applications:

- Secure Health Messages (SHM)
- Security Application Access (Biometric Authentication)

The changes to these two applications consists of:

1. Defect fixes applied by J.S. Smith as specified in J.S. Smith’s document "Security Final Status".
2. Performance Enhancements as specified in J.S. Smith’s documents "Biometric Enhancements Notes" and "Final Status" documents.
3. Test Strategy

3.1. Overall Strategy

The testing approach is largely determined by the nature of the delivery of the various software components that comprise the HopeMate 3.5 upgrade. This approach consists of incremental releases, “drops”, in which the later release contains additional functionality.

In order for the software to be verified by the Xth of month Y, the software is required to be of a good quality – and must be delivered on time, and must include the scheduled deliverables.

Additionally, in order for adequate regression testing to take place, all the functions must be fully fixed prior to next software “drop”.

To verify that each ‘drop’ contains the required deliverables, a series of pre-tests (per build tests) shall be performed upon each release. The aim of this is twofold – firstly to verify that the software is of “testable” quality, and secondly to verify that the application has not been adversely affected by the implementation of the additional functionality. [See Section 3.5 for details of the pre-tests.]

The main thrust of the approach is to intensively test the front end in the first releases, discovering the majority of errors in this period. With the majority of these errors fixed, end-to-end testing will be performed to prove total system processing in the later Releases.

System retest of outstanding errors and related regression testing will be performed on an ongoing basis.

When all serious errors are fixed, an additional set of test cases will be performed in the final release to ensure the system works in an integrated manner. It is intended that the testing in the later releases be the final proving of the system as a single application, and not testing of individual functions. There should be no class A or B errors outstanding prior to the start of final Release testing.

The testing will be performed in a stand alone test environment that will be as close as possible to the customer environment (live system). This means that the system will be running in as complete a manner as possible during all system testing activities.

3.2. Proposed Software "Drops" Schedule

>> See Schedule in Section 1.7

3.3. Testing Process

1) Preparing the test plan
2) Review USE CASE documentation
3) Preparing the test specification
4) Approval of the test specification
5) Setup of test environment
6) Preparing the test cases
7) Reviewing test plan
8) Approval of test plan
9) Reviewing test cases
10) Approve test cases
11) Test Execution – Integration Testing
12) Test Execution – System Test
13) Test Execution – User Acceptance Tests
14) Update test reports
15) Deliver final test reports

Chicago Hope DOIT plans to begin testing the HopeMate 3.5 application on Monday the 10th of month Y, 200x, and is scheduled to last until the 3rd of month Z, 200x.

All system test cases and automated test scripts will be developed from the functional specification, requirements definition, real world cases, and transaction type categories.

Most testing will take a black box approach, in that each test case will contain specified inputs, which will produce specified expected outputs.

Although the initial testing will be focused on individual objects (e.g. Emergency Department), the main bulk of the testing will be end-to-end testing, where the system will be tested in full.

3.4 Inspecting Results

Inspecting the results will at times comprise of:

- Inspecting on-screen displays
- Inspecting printouts of database tables
- Viewing of log files
- Cross application validation.

3.5. System Testing

3.5.1 Test Environment Pre-Test Criteria:

A series of simple tests will be performed for the test environment pre-test. In order for the software to be accepted into test these test cases should be completed successfully. Note - these tests are not intended to perform in depth testing of the software.

3.5.2. Resumption Criteria
In the event that system testing is suspended resumption criteria will be specified and testing will not re-commence until the software reaches these criteria.

3.6. Exit Criteria

The Exit Criteria detailed below must be achieved before the Phase 1 software can be recommended for promotion to User Acceptance status. Furthermore, it is recommend that there be a minimum 2 days effort Final Regression testing AFTER the final fix/change has been retested.

- All High Priority errors from System Test must be fixed and tested
- If any medium or low-priority errors are outstanding - the implementation risk must be signed off as acceptable by Design Team.

3.7. Daily Testing Strategy

The sequence of events will be as follows:

"Day 1"
Input data through the front end applications
Files will be verified to ensure they have correct test data
Run Day 1 cgi-bin & perl scripts
Check Outputs
Check Log files

"Day 2"
Validation & Check off via front-end applications
Files will be verified to ensure they have correct test data
Run Day 2 cgi-bin & perl scripts
Check Outputs
Check Log files

"Day 3"
Validation & Check off via front-end applications
Files will be verified to ensure they have correct test data
Unvalidated & Insecure Items Processing
Run cgi-bin & perl scripts Day 1 & 2
Check Outputs
Check Log files

3.8 Test Cases Overview
The software will be tested, and the test cases will be designed, under the following functional areas:

<table>
<thead>
<tr>
<th>SECTION</th>
<th>CATEGORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Build Tests</td>
</tr>
<tr>
<td>2.</td>
<td>System pre-tests</td>
</tr>
<tr>
<td>3.</td>
<td>System installation tests</td>
</tr>
<tr>
<td>4.</td>
<td>GUI Validation tests</td>
</tr>
<tr>
<td>5.</td>
<td>System Functionality tests</td>
</tr>
<tr>
<td>6.</td>
<td>Error Regression tests</td>
</tr>
<tr>
<td>7.</td>
<td>Full Regression Cycle</td>
</tr>
</tbody>
</table>

### 3.9. System Testing Cycles

The system testing will occur in distinct phases:

For each build of the software that the test team receives the following tests will be performed:

<table>
<thead>
<tr>
<th>Testing Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Build Tests</td>
</tr>
<tr>
<td>2. Installation Tests</td>
</tr>
<tr>
<td>3. System Pre-Tests</td>
</tr>
<tr>
<td>4. Testing of new functionality</td>
</tr>
<tr>
<td>5. Retest of fixed Defects</td>
</tr>
<tr>
<td>6. Regression testing of unchanged functionality</td>
</tr>
</tbody>
</table>
4. Management Procedures

4.1. Error Tracking & Reporting

All errors and suspected errors found during testing will be input by each tester to the Chicago Hope DOIT MS Access Error tracking system.

The Defects will be recorded against specific builds and products, and will include a one line summary description, a detailed description and a hyperlink to a screenshot (if appropriate).

The error tracking system will be managed (logged, updated and reported) by Chicago Hope DOIT.

Any critical problems likely to affect the end date for testing will be reported immediately to the Test Manager and hence the Quality Manager.

Daily metrics will be generated, detailing the testing progress that day. The report will include Defect classification, status and summary description.

4.2. Error Management

During System Test, discrepancies will be recorded as they are detected by the testers. These discrepancies will be input into the Chicago Hope DOIT Defect Database with the status “Open”.

Daily an “Error Review Team” will meet to review and prioritise the errors that were raised, and either assign, defer, or close them as appropriate. Error assignment shall be via email.

This team will consist of the following representatives:

<table>
<thead>
<tr>
<th>MANAGEMENT TEAM</th>
<th>DEFECT FIX TEAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Test Manager</td>
<td>Developer 1</td>
</tr>
<tr>
<td>Test Team Leader</td>
<td>Developer 2</td>
</tr>
<tr>
<td>Build &amp; Release Engineer</td>
<td>Design Person 1</td>
</tr>
<tr>
<td>Development Manager</td>
<td>Design Person 2</td>
</tr>
</tbody>
</table>

4.2.1. Error Status Flow

Overview of test status flow: (Note - Engineering status flow not reflected - Assigned & Work in Progress errors are with the development team, and the engineering status is kept separate. The DOIT Test Team will then revisit the Defect when it exits the engineering status and "re-
appears" as either "Open" or "Fixed to be Confirmed".)

Open => Assigned => Work in Progress => Fixed to be Confirmed => CLOSED + reason for closing

The Error status changes as follows: initially Open, Error Review Team sets Assigned, Tech Team leads sets the value for "Assigned To" (i.e. the developer who will fix it) and when Defect is being fixed the developer sets it to Work in Progress, then sets it to "Fixed to be Confirmed" when fix is ready. Note - Only the testers can set the error status to closed. These changes can only be made in the Error Management system according to your access rights.

Closed has a set of reason values - e.g. Fixed & Retested; Not an Error; Duplicate; Change Request; 3rd Party error.

4.3. Error Classification

Errors, which are agreed as valid, will be categorised as follows by the Error Review Team:

- Category A - Serious errors that prevent System test of a particular function continuing or serious data type error
- Category B - Serious or missing data related errors that will not prevent implementation.
- Category C - Minor errors that do not prevent or hinder functionality.

4.3.1. Explanation of Classifications

1) An "A" Defect is a either a showstopper or of such importance as to radically affect the functionality of the system i.e.:
   - Potential harm to patient
   - If, because of a consistent crash during processing of a new application, a user could not complete that application.
   - Incorrect data is passed to system resulting in corruption or system crashes

   Example of severally affected functionality:
   - Change of patient status incorrectly identifies …….
   - Incorrect dosage of medications for ………
2) Defects would be classified as "B" where a less important element of functionality is affected, e.g.:
   • a value is not defaulting correctly and it is necessary to input the correct value
   • data is affected which does not have a major impact, for example - where an element of a patient’s chart was not propagated to the database
   • there is an alternative method of completing a particular process - e.g. a problem might occur which has a work-around.
   • Serious cosmetic error on front-end.

3) "C" type Defects are mainly cosmetic Defects i.e.:
   • Incorrect / misspelling of text on screens
   • drop down lists missing or repeating an option

4.3.2. Error Turnaround Time

In the event of an error preventing System Test from continuing, the error should be turned around by Defect fix team in 24 hours, i.e. it should be given the highest priority by the fixing team, calling in additional resources if required.

4.3.3. Re-release of Software

The release of newer versions of the software will be co-ordinated with the Test Manager - new versions should only be released when agreed, and where there is a definite benefit (i.e. contains fixes to X or more numbers of Defects).

Retest of outstanding errors and related regression testing will be performed on an ongoing basis.

4.3.4 Build Release Procedures

During System Test the scheduled releases of the software will be co-ordinated between the Development Team leader, the Build Engineer and the System Test Manager, according the schedule in Section 7.

However, no new versions, other than the agreed drops, will be deployed under the control of the System Test Manager unless it concerns a fix to a very serious error or issue that prevents testing from continuing.
## 4.4. Error Tracking, Reporting & Updating

<table>
<thead>
<tr>
<th></th>
<th>The Test Lead will refer any major error/anomaly to either Development Team Leader or designated representative on the development team as well as raising a formal error record. J.S. Smith will also be immediately informed of any Defects that delay testing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>This has several advantages: § it prevents the testers trying to proceed beyond 'showstoppers' § it puts the developer on immediate notice of the problem § it allows the developer to put on any traces that might be necessary to track down the error.</td>
</tr>
<tr>
<td>2.</td>
<td>All Defects raised will be entered into the Defect Database, which will contain all relevant data.</td>
</tr>
<tr>
<td>3.</td>
<td>These errors will be logged on the day they occur with a status of &quot;OPEN&quot;</td>
</tr>
<tr>
<td>4.</td>
<td>A daily &quot;Error Review Team&quot; will meet to review and prioritise the discrepancies that were raised, and will assign the Defects to the appropriate parties for fixing. The assignments will be made automatically via email by the Error Management System.</td>
</tr>
<tr>
<td>5.</td>
<td>The database should be updated by the relevant person (i.e. developer, tester etc.) with the status of all errors should the status of the error change - e.g. Assigned, Work in Progress, Fixed to be confirmed, Closed.</td>
</tr>
<tr>
<td>6.</td>
<td>Errors will remain in an &quot;Open&quot; state until they are &quot;Assigned&quot; to the responsible parties; they will remain &quot;Assigned&quot; until the status of the error changes to &quot;Work In Progress&quot;, then set to &quot;Fixed to be Confirmed&quot; when a fix is ready or &quot;Open&quot; if not fixed.</td>
</tr>
</tbody>
</table>
4.4.1. Purpose of Error Review Team.

The purpose of the Error Review meeting is to ensure the maximum efficiency of the development and system testing teams for the release of the new software through close co-operation of all involved parties.

This will be achieved through daily meetings whose function will be to

- Agree status of each raised Error
- Prioritisation of valid Error's
- Ensure that enough documentation is available with Error's.
- Agree content and timescale for software releases into System test.
- Ensure one agreed source of Error reporting information.
- Identify any issues which may affect the performance of system testing.

4.4.2. Error Review Team Meeting Agenda.

- Review any actions from last meeting.
- Review new Error's raised for Duplicates etc.
- Determine adequacy of documentation associated with raised Error's.
- Classify and prioritise each new Error. (Note: 'A' Defects automatically are priority).
- Agree release content and timescale. (see note).
- Review of assigned actions from meeting.
Note: Release content and timescale must be co-ordinated with the Development Manager and System Test Manager, and must satisfy the agreed Build Handover procedures. This also applies to any production fixes implemented - the handover has to specify the content of the release, including any such fixes.

4.5. Progress Reporting

Progress reports will be distributed to the Project Meeting, Development Manager and Quality Manager. The progress report will detail:

- Test Plans
- Test Progress to date
- Test execution summary
- List of problems opened
- Plans of next test activities to be performed
- Any testing issues/risks which have been identified

For each set of test cases executed, the following results shall be maintained...

- Test Identifier, and Error identifier if the test failed
- Result - Passed, Failed, Not Executable, For Retest
- Test Log
- Date of execution
- Signature of tester

4.6. Retest & Regression Testing

Once a new build or software is delivered into Test, errors which have been resolved by the Development Engineers, will be validated. Any discrepancies will be entered/re-entered into the error tracking system and reported back to the Development Team.

Builds will be handed off to System Test, as described in point 4.3.4.

This procedure will be repeated until the exit criteria are satisfied.
5. Test Preparation and Execution

5.1 Processes

The test procedures and guidelines to be followed for this project will be detailed as follows:

- A specific test directory has been setup, which will contain the test plan, all test cases, the error tracking database and any other relevant documentation.

  The location of this directory is: \\Server\Folder\QA\SystemTesting

The following table lists the test documents which will be produced as part of the System testing of the software, along with a brief description of the contents of each document.

<table>
<thead>
<tr>
<th>DOCUMENTATION</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Plan</td>
<td>(this document) Describes the overall test approach for the project, phases, objectives etc. to be taken for testing the software, and Control documentation.</td>
</tr>
<tr>
<td>Test Specification</td>
<td>Describes the test environment - hardware &amp; Software; the test machine configurations to be used and provides a detailed specification of the tests to be carried out.</td>
</tr>
<tr>
<td>Test Logs</td>
<td>Results of test cases.</td>
</tr>
<tr>
<td>DOIT Defect Database</td>
<td>Contains the Defect database used for tracking the Defects reported during system test.</td>
</tr>
<tr>
<td>Test Report</td>
<td>Post testing review &amp; analysis.</td>
</tr>
<tr>
<td>Metrics</td>
<td>Daily, Weekly &amp; Summary totals of Errors raised during System Test.</td>
</tr>
</tbody>
</table>

*Table 5-1: System Test Documentation Suite*

5.2. Formal Reviewing

There will be several formal review points before and during system test, including the review of this document. This is a vital element in achieving a quality product.

5.2.1. Formal Review Points

- Design Documentation - Requirements Specification & Functional Specification
- System Test Plan
- System Test Conditions
- System Test Progress/Results
- Post System Test Review
6. **Risks & Dependencies**

The testing could be severely impacted, resulting in incomplete or inadequate testing of the software, or adversely affecting the release date, by the following risks and dependencies:

### 6.1. Risks

<table>
<thead>
<tr>
<th>DETAIL</th>
<th>RESPONSIBLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out of date/inaccurate requirements definition/functional specification(s)</td>
<td></td>
</tr>
<tr>
<td>Lack of unit testing</td>
<td></td>
</tr>
<tr>
<td>Problems relating to Code merge</td>
<td></td>
</tr>
<tr>
<td>Test Coverage limitations (OS/Browser matrix)</td>
<td></td>
</tr>
</tbody>
</table>

### 6.2. Dependencies

All functionality has to be fully tested prior to the Xth of month Y. That means that the close of business on the Xth of month Y is the latest time in which a test build can be accepted for final regression testing if the testing is to be complete on the Zrd of month Z.

This means that all Defects must have been detected and fixed by close of business on the Xth of month Y, leaving 3 days to do the final regression test.

#### 6.2.1 General Dependencies

- System will be delivered on time.
- System is of testable quality.
- Test environment is setup and available for test use on time.
- All "Show-Stopper" errors receive immediate attention from the development team.
- All Defects found in a version of the software will be fixed and unit tested by the development team before the next version is released.
- All documentation will be up to date and delivered to the system test team in time for creating/amending relevant tests.
- The design of the software must be final, and design documentation must be complete, informative prior to System Test proper commences.
7. Signoff

This document must be formally approved before System Test can commence. The following people will be required to sign off (see Document Control at beginning of Test Plan):

<table>
<thead>
<tr>
<th>GROUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chicago Hope DOIT Quality Manager</td>
</tr>
<tr>
<td>Chicago Hope DOIT System Test Manager</td>
</tr>
<tr>
<td>Test Team Leader</td>
</tr>
<tr>
<td>Project Manager1</td>
</tr>
<tr>
<td>Business Owner</td>
</tr>
</tbody>
</table>
The checklists found in appendix C are referenced as part of Skill Category 7, section 7.1.3, *Defining Test Conditions from Test Transaction Types.*

<table>
<thead>
<tr>
<th>Checklist</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Transaction Types Checklist</td>
<td>C-2</td>
</tr>
<tr>
<td>Records Testing Checklist</td>
<td>C-4</td>
</tr>
<tr>
<td>File Testing Checklist</td>
<td>C-5</td>
</tr>
<tr>
<td>Relationship Conditions Checklist</td>
<td>C-6</td>
</tr>
<tr>
<td>Error Conditions Testing Checklist</td>
<td>C-8</td>
</tr>
<tr>
<td>Use of Output Conditions Checklist</td>
<td>C-10</td>
</tr>
<tr>
<td>Search Conditions Checklist</td>
<td>C-11</td>
</tr>
<tr>
<td>Merging/Matching Conditions Checklist</td>
<td>C-12</td>
</tr>
<tr>
<td>Stress Conditions Checklist</td>
<td>C-14</td>
</tr>
<tr>
<td>Control Conditions Checklist</td>
<td>C-15</td>
</tr>
<tr>
<td>Attributes Conditions Checklist</td>
<td>C-18</td>
</tr>
<tr>
<td>Stress Conditions Checklist</td>
<td>C-20</td>
</tr>
<tr>
<td>Procedures Conditions Checklist</td>
<td>C-21</td>
</tr>
</tbody>
</table>
# C.1 Field Transaction Types Checklist

<table>
<thead>
<tr>
<th>#</th>
<th>Item</th>
<th>RESPONSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Have all codes been validated?</td>
<td>Yes</td>
</tr>
<tr>
<td>2.</td>
<td>Can fields be properly updated?</td>
<td>No</td>
</tr>
<tr>
<td>3.</td>
<td>Is there adequate size in the field for accumulation of totals?</td>
<td>N/A</td>
</tr>
<tr>
<td>4.</td>
<td>Can the field be properly initialized?</td>
<td>Comments</td>
</tr>
<tr>
<td>5.</td>
<td>If there are restrictions on the contents of the field, are those restrictions validated?</td>
<td>Remarks</td>
</tr>
<tr>
<td>6.</td>
<td>Are rules established for identifying and processing invalid field data?</td>
<td>Remarks</td>
</tr>
<tr>
<td></td>
<td>a. If no, develop this data for the error-handling transaction type.</td>
<td>Remarks</td>
</tr>
<tr>
<td></td>
<td>b. If yes, have test conditions been prepared to validate the specification processing for invalid field data?</td>
<td>Remarks</td>
</tr>
<tr>
<td>7.</td>
<td>Have a wide range of normal valid processing values been included in the test conditions?</td>
<td>Yes</td>
</tr>
<tr>
<td>8.</td>
<td>For numerical fields, have the upper and lower values been tested?</td>
<td>No</td>
</tr>
<tr>
<td>9.</td>
<td>For numerical fields, has a zero value been tested?</td>
<td>N/A</td>
</tr>
<tr>
<td>10.</td>
<td>For numerical fields, has a negative test condition been prepared?</td>
<td>Yes</td>
</tr>
<tr>
<td>11.</td>
<td>For alphabetical fields, has a blank condition been prepared?</td>
<td>No</td>
</tr>
<tr>
<td>12.</td>
<td>For an alphabetical/alphanumeric field, has a test condition longer than the field length been prepared? (The purpose is to check truncation processing.)</td>
<td>Yes</td>
</tr>
<tr>
<td>13.</td>
<td>Have you verified from the data dictionary printout that all valid conditions have been tested?</td>
<td>Remarks</td>
</tr>
<tr>
<td>14.</td>
<td>Have you reviewed systems specifications to determine that all valid conditions have been tested?</td>
<td>Yes</td>
</tr>
<tr>
<td>#</td>
<td>Item</td>
<td>RESPONSE</td>
</tr>
<tr>
<td>----</td>
<td>----------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td></td>
<td><strong>15. Have you reviewed requirements to determine all valid conditions have been tested?</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>16. Have you verified with the owner of the data element that all valid conditions have been tested?</strong></td>
<td></td>
</tr>
</tbody>
</table>
## C.2 Records Testing Checklist

<table>
<thead>
<tr>
<th>#</th>
<th>Item</th>
<th>RESPONSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Has a condition been prepared to test the processing of the first record?</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>Has a condition been determined to validate the processing of the last record?</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>If there are multiple records per transaction, are they all processed correctly?</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>If there are multiple records on a storage media (i.e., permanent or temporary file), are they all processed correctly?</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>If there are variations in size of record, are all those size variations tested (e.g., a header with variable length trailers)?</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>Can two records with the same identifier be processed (e.g., two payments for the same accounts receivables file)?</td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>Can the first record stored by retrieved?</td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>Can the last record stored be retrieved?</td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td>Will all the records entered be properly stored?</td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td>Can all of the records stored be retrieved?</td>
<td></td>
</tr>
<tr>
<td>11.</td>
<td>Do current record formats coincide with the formats used on files created by other systems?</td>
<td></td>
</tr>
</tbody>
</table>
## C.3 File Testing Checklist

<table>
<thead>
<tr>
<th>#</th>
<th>Item</th>
<th>RESPONSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Has a condition been prepared to test each file?</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>Has a condition been prepared to test each file’s interface with each module?</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>Have test conditions been prepared to validate access to each file in different environments (e.g., web, mobile, batch)?</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>Has a condition been prepared to validate that the correct version of each file will be used?</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>Have conditions been prepared that will validate that each file will be properly closed after the last record has been processed for that file?</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>Have conditions been prepared that will validate that each record type can be processed from beginning to end of the system intact?</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>Have conditions been prepared to validate that all of the records entered will be processed through the system?</td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td>Are test conditions prepared to create a file for which no prior records exist?</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>Has a condition been prepared to validate the correct closing of a file when all records on the file have been deleted?</td>
<td>Yes</td>
</tr>
</tbody>
</table>
## C.4 Relationship Conditions Checklist

<table>
<thead>
<tr>
<th>#</th>
<th>Item</th>
<th>RESPONSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Has a data element relationship test matrix been prepared?</td>
<td>Yes</td>
</tr>
<tr>
<td>2.</td>
<td>Has the relationship matrix been verified for accuracy and completeness with the end user/customer/BA of the system?</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Has the relationship matrix been verified for accuracy and completeness with the project leader of the system?</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>Does the test relationship matrix include the following relationships:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a. Value of one field related to the value in another field</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b. Range of values in one field related to a value in another field</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c. Including a value in one field requires the inclusion of a value in another field</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d. The absence of a value in one field causes the absence of a value in another field</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e. The presence of a value in one field causes the absence of certain values in another field (for example, the existence of a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>specific product in another field, such as a retail customer may not buy a commercial product)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f. The value in one field is inconsistent with past values for that field (for example, a customer who normally buys a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>in a quantity of two or three now has a purchase quantity of 600)</td>
<td></td>
</tr>
</tbody>
</table>
4. g. The value in one field is inconsistent with what would logically be expected for an area/activity (for example, it may be inconsistent for people in a particular department to work and be paid for overtime)

h. The value in one field is unrealistic for that field (for example, for hours worked overtime, 83 might be an unrealistic value for that field — this is a relationship between field and the value in the field)

i. Relationships between time periods and conditions (for example, bonuses might only be paid during the last week of a quarter)

j. Relationships between time of day and processing occurring (for example, a teller transaction occurring other than normal banking hours)

<table>
<thead>
<tr>
<th>#</th>
<th>Item</th>
<th>RESPONSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>g. The value in one field is inconsistent with what would logically be expected for an area/activity (for example, it may be inconsistent for people in a particular department to work and be paid for overtime)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>h. The value in one field is unrealistic for that field (for example, for hours worked overtime, 83 might be an unrealistic value for that field — this is a relationship between field and the value in the field)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i. Relationships between time periods and conditions (for example, bonuses might only be paid during the last week of a quarter)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>j. Relationships between time of day and processing occurring (for example, a teller transaction occurring other than normal banking hours)</td>
<td></td>
</tr>
</tbody>
</table>

5. Have conditions been prepared for all relationships that have a significant impact on the application?
## C.5 Error Conditions Testing Checklist

<table>
<thead>
<tr>
<th>#</th>
<th>Item</th>
<th>RESPONSE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>1.</td>
<td>Has a brainstorming session with end users/customers been performed to identify functional errors?</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>Has a brainstorming session been conducted with project personnel to identify structural error conditions?</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Have functional error conditions been identified for the following cases:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a. Rejection of invalid codes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b. Rejection of out-of-range values</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c. Rejection of improper data relationships</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d. Rejections of invalid dates</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e. Rejections of unauthorized transactions of the following types:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Not a valid value</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Not a valid customer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Not a valid product</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Not a valid transaction type</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Not a valid price</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f. Alphabetic data in numeric fields</td>
<td></td>
</tr>
<tr>
<td></td>
<td>g. Blanks in a numeric field</td>
<td></td>
</tr>
<tr>
<td></td>
<td>h. All blank conditions in a numerical field</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i. Negative values in a positive field</td>
<td></td>
</tr>
<tr>
<td></td>
<td>j. Positive values in a negative field</td>
<td></td>
</tr>
<tr>
<td></td>
<td>k. Negative balances in a financial account</td>
<td></td>
</tr>
<tr>
<td></td>
<td>l. Numbers in an alphabetic field</td>
<td></td>
</tr>
<tr>
<td></td>
<td>m. Blanks in an alphabetic field</td>
<td></td>
</tr>
<tr>
<td></td>
<td>n. Values longer than the field permits</td>
<td></td>
</tr>
<tr>
<td></td>
<td>o. Totals which exceed maximum size of total fields</td>
<td></td>
</tr>
<tr>
<td>#</td>
<td>Item</td>
<td>RESPONSE</td>
</tr>
<tr>
<td>---</td>
<td>------------------------------------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>3.</td>
<td>p. Proper accumulation of totals (at all levels for multiple level totals)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>q. Incomplete transactions (i.e., one or more fields missing)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r. Obsolete data in the field (e.g., a code which had been valid but is no longer valid)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>s. New value which will become acceptable but is not acceptable at the current time (e.g., a new district code for which the district has not yet been established)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>t. A postdated transaction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>u. Change of a value which affects a relationship (e.g., if the unit digit was used to control year, then the switching from nine in 89 to zero in 90 can be adequately processed)</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>Has the data dictionary list of field specifications been used to generate invalid specifications?</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>Have the following architectural error conditions been tested:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a. Page overflow</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b. Report format conformance to design layout</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c. Posting of data in correct portion of reports</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d. Printed error messages are representative of actual error condition</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e. All instructions are executed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f. All paths are executed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>g. All internal tables are tested</td>
<td></td>
</tr>
<tr>
<td></td>
<td>h. All loops are tested</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i. All “perform” type of routines have been adequately tested</td>
<td></td>
</tr>
<tr>
<td></td>
<td>j. All compiler warning messages have been adequately addressed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>k. The correct version of the program has been tested</td>
<td></td>
</tr>
<tr>
<td></td>
<td>l. Unchanged portions of the system will be revalidated after any part of the system has been changed</td>
<td></td>
</tr>
</tbody>
</table>
## C.6 Use of Output Conditions Checklist

<table>
<thead>
<tr>
<th>#</th>
<th>Item</th>
<th>RESPONSE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>1.</td>
<td>Have all of the end user actions been identified?</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>Have the actions been identified in enough detail that the contributions of information system outputs can be related to those actions?</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Has all of the information utilized in taking an action been identified and related to the action?</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>Have the outputs from the application under test been identified to the specific actions?</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>Does the end user correctly understand the output reports/screens?</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>Does the end user understand the type of logic/computation performed in producing those outputs?</td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>Is the end user able to identify the contribution those outputs make to the actions taken?</td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>Has the relationship between the system outputs and business actions been defined?</td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td>Does the interpretation of the matrix indicate that the end user does not have adequate information to take an action?</td>
<td></td>
</tr>
</tbody>
</table>
## C.7 Search Conditions Checklist

<table>
<thead>
<tr>
<th>#</th>
<th>Item</th>
<th>RESPONSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Have all the internal tables been identified?</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>Have all the internal lists of error messages been identified?</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>Has the search logic been identified?</td>
<td>N/A</td>
</tr>
<tr>
<td>4</td>
<td>Have all the authorization routines been identified?</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>Have all the password routines been identified?</td>
<td>No</td>
</tr>
<tr>
<td>6</td>
<td>Has all the business processing logic that requires a search been identified?</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>Have the data base search routines been identified?</td>
<td>N/A</td>
</tr>
<tr>
<td>8</td>
<td>Have subsystem searches been identified (for example, finding a tax rate in a sales tax subsystem)?</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>Has a complex search logic been identified (i.e., that requiring two or more conditions or two or more records such as searching for accounts over 90 days old and over $100)?</td>
<td>Yes</td>
</tr>
<tr>
<td>10</td>
<td>Have test conditions been graded for all of the above search conditions?</td>
<td>No</td>
</tr>
<tr>
<td>11</td>
<td>Has the end user been interviewed to determine the type of one-time searches that might be encountered in the future?</td>
<td>No</td>
</tr>
</tbody>
</table>
## C.8 Merging/Matching Conditions Checklist

<table>
<thead>
<tr>
<th>#</th>
<th>Item</th>
<th>RESPONSE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Have all the files associated with the application been identified? (Note that in this condition files include specialized files, data bases, and internal groupings of records used for matching and merging).</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Have the following merge/match conditions been addressed:</td>
<td>Comments</td>
</tr>
<tr>
<td>1.</td>
<td>a. Merge/match of records of two different identifiers (inserting a new item, such as a new employee on the payroll file)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b. A merge/match on which there are no records on the merged/matched file</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c. A merge/match in which the merged/matched record becomes the lowest value on the file</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d. A merge/match in which the merged/matched record becomes the highest value on the file</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e. A merge/match in which the merged/matched record has an equal value as an item on a file, for example, adding a new employee in which the new employee's payroll number equals an existing payroll number on the file</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f. A merge/match for which there is no input file/transactions being merged/matched</td>
<td></td>
</tr>
<tr>
<td></td>
<td>g. A merge/match in which the first item on the file is deleted</td>
<td></td>
</tr>
<tr>
<td></td>
<td>h. A merge/match is which the last item on the merged/matched file is deleted</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i. A merge/match in which two incoming records have the same value</td>
<td></td>
</tr>
</tbody>
</table>
2.  j. A merge/match in which two incoming records indicated a value on the merged/matched file is to be deleted  
    k. A merge/match condition when the last remaining record on the merged/matched file is deleted  
    l. A merge/match condition in which the incoming merged/matched file is out of the sequence, or has a single record out of sequence  
   
3. Have these test conditions been applied to the totality of merge/match conditions that can occur in the application under test?
## C.9 Stress Conditions Checklist

<table>
<thead>
<tr>
<th>#</th>
<th>Item</th>
<th>RESPONSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Have all the desired performance capabilities been identified?</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>Have all the system features that contribute to test been identified?</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Have the following system performance capabilities been identified:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a. Turnaround performance</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b. Availability/up-time performance</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c. Response time performance</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d. Error handling performance</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e. Report generation performance</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f. Internal computational performance</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>Have the following system features been identified which may adversely affect performance:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a. Internal computer processing speed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b. Efficiency of programming language</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c. Efficiency of data base management system</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d. File storage capabilities</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>Do the project personnel agree that the stress conditions are realistic to validate software performance?</td>
<td></td>
</tr>
</tbody>
</table>
## C.10 Control Conditions Checklist

<table>
<thead>
<tr>
<th>#</th>
<th>Item</th>
<th>RESPONSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Have the business transactions processed by the software been identified?</td>
<td>Yes</td>
</tr>
<tr>
<td>2.</td>
<td>Has a transaction flow analysis been made for each transaction</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Have the controls over the transaction flow been documented?</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>Do the data input controls address the following areas:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a. Do they ensure the accuracy of data input?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b. Do they ensure the completeness of data input?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c. Do they ensure the timeliness of data input?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d. Are record counts used where applicable?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e. Are predetermined control totals used where applicable?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f. Are control logs used where applicable?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>g. Is key verification used where applicable?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>h. Are the input data elements validated?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i. Are controls in place to monitor overrides and bypasses?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>j. Are overrides and bypasses restricted to supervisory personnel?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>k. Are overrides and bypasses automatically recorded and submitted to supervision for analysis?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>l. Are transaction errors recorded?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>m. Are rejected transactions monitored to assure that they are corrected on a timely basis?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>n. Are passwords required to enter business transactions?</td>
<td></td>
</tr>
<tr>
<td>#</td>
<td>Item</td>
<td>RESPONSE</td>
</tr>
<tr>
<td>----</td>
<td>----------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td></td>
<td>Are applications shut down (locked) after predefined periods of inactivity?</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>Do the data entry controls include the following controls:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a. Is originated data accurate?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b. Is originated data complete?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c. Is originated data recorded on a timely basis?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d. Are there procedures and methods for data origination?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e. Are cross-referenced fields checked?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f. Are pre-numbered documents used where applicable?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>g. Is there an effective method for authorizing transactions?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>h. Are systems overrides controlled? Are they applicable?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i. Are manual adjustments controlled?</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>Do the processing controls address the following areas:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a. Are controls over input maintained throughout processing?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b. Is all entered data validated?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c. Do overriding/bypass procedures need to be manually validated after processing?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d. Is a transaction history file maintained?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e. Do procedures exist to control errors?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f. Are rejected transactions controlled to assure correct and reentry?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>g. Have procedures been established to control the integrity of data files/data bases?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>h. Do controls exist over recording the correct dates for transactions?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i. Are there concurrent update protections procedures?</td>
<td></td>
</tr>
<tr>
<td>#</td>
<td>Item</td>
<td>RESPONSE</td>
</tr>
<tr>
<td>----</td>
<td>----------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>6.</td>
<td>j. Are easy-to-understand error messages printed out for each error condition?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>k. Are the procedures for processing corrected transactions the same as those for processing original transactions?</td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>Do the data output controls address the following items?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a. Are controls in place to assure the completeness of output?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b. Are output documents reviewed to ensure that they are generally acceptable and complete?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c. Are output documents reconciled to record counts/control totals?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d. Are controls in place to ensure that output products receive the appropriate security protection?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e. Are output error messages clearly identified?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f. Is a history maintained of output product errors?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>g. Are users informed of abnormal terminations?</td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>Has the level of risk for each control area been identified?</td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td>Has the level of risk been confirmed with the audit function?</td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td>Has the end user/customer been notified of the level of control risk?</td>
<td></td>
</tr>
</tbody>
</table>
## C.11 Attributes Conditions Checklist

<table>
<thead>
<tr>
<th>#</th>
<th>Item</th>
<th>RESPONSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Have the software attributes been identified?</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>Have the software attributes been ranked?</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Does the end user/customer agree with the attribute ranking?</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>Have test conditions been developed for at least the high importance attributes?</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>For the correctness attributes, are the functions validated as accurate and complete?</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>For the authorization attribute, have the authorization procedures for each transaction been validated?</td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>For the file integrity attribute, has the integrity of each file/table been validated?</td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>For the audit trail attribute, have test conditions validated that each business transaction can be reconstructed?</td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td>For the continuity of processing attribute, has it been validated that the system can be recovered within a reasonable time span and that transactions can be captured and/or processed during the recovery period?</td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td>For the service attribute, has it been validated that turnaround time/response time meets user needs?</td>
<td></td>
</tr>
<tr>
<td>11.</td>
<td>For the access control attribute, has it been validated that only authorized individuals can gain access to the system?</td>
<td></td>
</tr>
<tr>
<td>#</td>
<td>Item</td>
<td>RESPONSE</td>
</tr>
<tr>
<td>----</td>
<td>----------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>12</td>
<td>For the compliance attribute, has it been validated that IT standards are complied with, that the system development methodology is being followed, and that the appropriate policies, procedures, and regulations are complied with?</td>
<td>Yes</td>
</tr>
<tr>
<td>13</td>
<td>For the reliability attribute, has it been validated that incorrect, incomplete, or obsolete data will be processed properly?</td>
<td>No</td>
</tr>
<tr>
<td>14</td>
<td>For the ease of use attribute, has it been validated that people can use the system effectively, efficiently, and economically?</td>
<td>N/A</td>
</tr>
<tr>
<td>15</td>
<td>For the maintainable attribute, has it been validated that the system can be changed, enhanced with reasonable effort and on a timely basis?</td>
<td>Comments</td>
</tr>
<tr>
<td>16</td>
<td>For the portable attribute, has it been validated that the software can be moved efficiently to other platforms?</td>
<td>No</td>
</tr>
<tr>
<td>17</td>
<td>For the coupling attribute, has it been validated that this software system can properly integrate with other systems?</td>
<td>Yes</td>
</tr>
<tr>
<td>18</td>
<td>For the performance attribute, has it been validated that the processing performance/software performance is acceptable to the end user?</td>
<td>No</td>
</tr>
<tr>
<td>19</td>
<td>For the ease of use attribute, has it been validated that the operation personnel can effectively, economically, and efficiently operate the software?</td>
<td>N/A</td>
</tr>
</tbody>
</table>
## C.12 Stress Conditions Checklist

<table>
<thead>
<tr>
<th>#</th>
<th>Item</th>
<th>RESPONSE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>1.</td>
<td>Has the state of an empty master file been validated?</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>Has the state of an empty transaction file been validated?</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Has the state of an empty table been validated?</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>Has the state of an insufficient quantity been validated?</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>Has the state of negative balance been validated?</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>Has the state of duplicate input been validated?</td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>Has the state of entering the same transaction twice been validated (particularly from a web app)?</td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>Has the state of concurrent update been validated (i.e., two client systems calling on the same master record at the same time)?</td>
<td></td>
</tr>
</tbody>
</table>
## C.13 Procedures Conditions Checklist

<table>
<thead>
<tr>
<th>#</th>
<th>Item</th>
<th>RESPONSE</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Have the backup procedures been validated?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>comments</td>
</tr>
<tr>
<td>2.</td>
<td>Have the off-site storage procedures been validated?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>comments</td>
</tr>
<tr>
<td>3.</td>
<td>Have the recovery procedures been validated?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>comments</td>
</tr>
<tr>
<td>4.</td>
<td>Have the client side operating procedures been validated?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>comments</td>
</tr>
</tbody>
</table>
It is each candidate’s responsibility to stay current in the field and to be aware of published works and materials available for professional study and development. Software Certifications recommends that candidates for certification continually research and stay aware of current literature and trends in the field. There are many valuable references that have not been listed here. These references are for informational purposes only.

Ambler, Scott. *Web services programming tips and tricks: Documenting a Use Case*. October 2000

The American Heritage® Science Dictionary, Copyright © 2002. Published by Houghton Mifflin. All rights reserved.


Beck, Kent, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas. 2001, Per the above authors this declaration may be freely copied in any form, but only in its entirety through this notice.


Joiner, Brian. “Stable and Unstable Processes, Appropriate and Inappropriate Managerial Action.” From an address given at a Deming User’s Group Conference in Cincinnati, OH.


Spence, Linda, University of Sutherland. *Software Engineering*. Available at http://osiris.sunderland.ac.uk/rif/linda_spence/HTML/contents.html


William, Wake. Senior Consultant, Industrial Logic, Inc.
