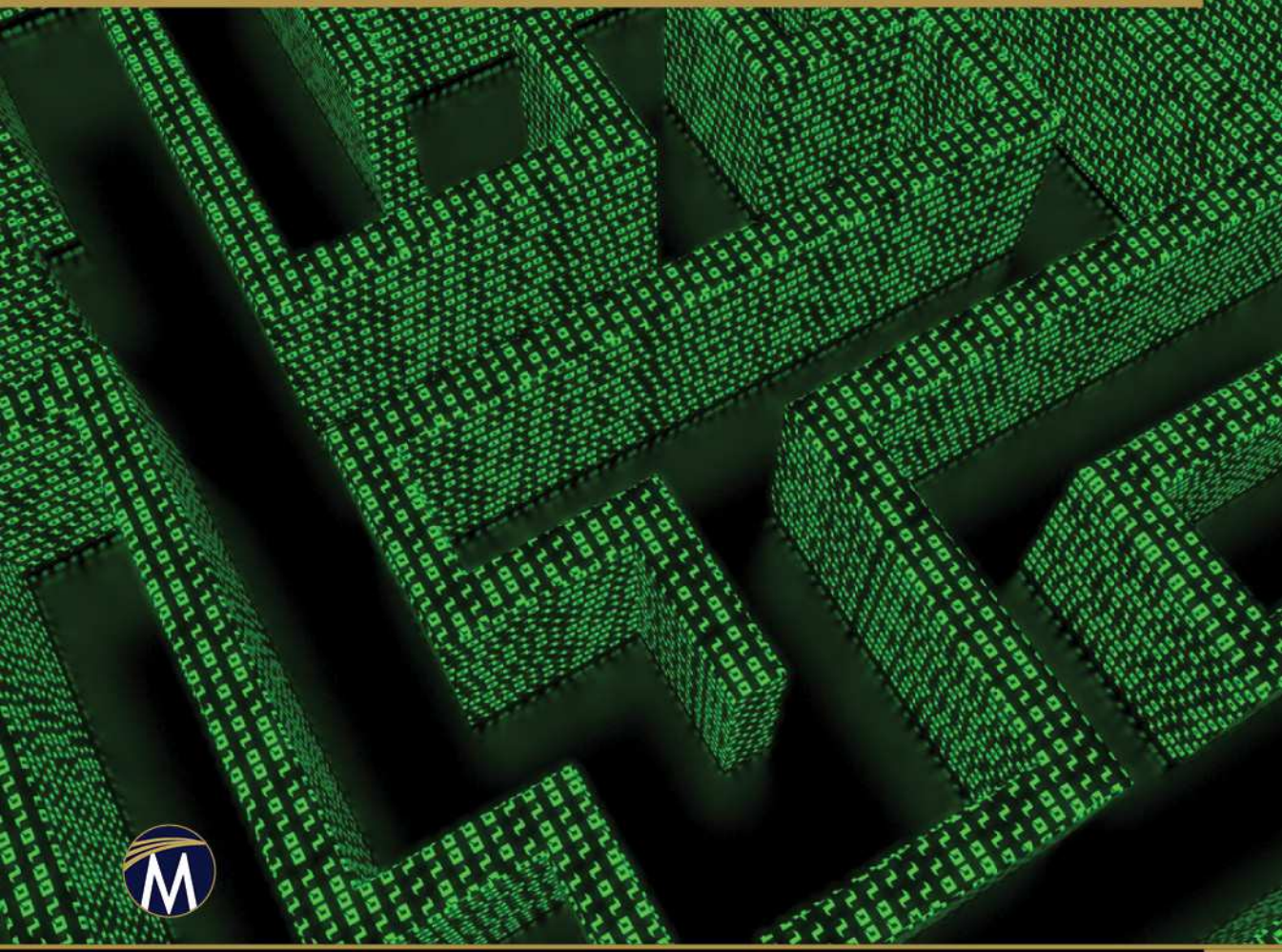
Software Quality Assurance

A Self-Teaching Introduction



R. CHOPRA

Software Quality Assurance

LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY

By purchasing or using this book (the "Work"), you agree that this license grants permission to use the contents contained herein, but does not give you the right of ownership to any of the textual content in the book or ownership to any of the information or products contained in it. *This license does not permit uploading of the Work onto the Internet or on a network (of any kind) without the written consent of the Publisher*. Duplication or dissemination of any text, code, simulations, images, etc. contained herein is limited to and subject to licensing terms for the respective products, and permission must be obtained from the Publisher or the owner of the content, etc., in order to reproduce or network any portion of the textual material (in any media) that is contained in the Work.

Mercury Learning and Information ("MLI" or "the Publisher") and anyone involved in the creation, writing, or production of the companion disc, accompanying algorithms, code, or computer programs ("the software"), and any accompanying Web site or software of the Work, cannot and do not warrant the performance or results that might be obtained by using the contents of the Work. The author, developers, and the Publisher have used their best efforts to insure the accuracy and functionality of the textual material and/or programs contained in this package; we, however, make no warranty of any kind, express or implied, regarding the performance of these contents or programs. The Work is sold "as is" without warranty (except for defective materials used in manufacturing the book or due to faulty workmanship).

The author, developers, and the publisher of any accompanying content, and anyone involved in the composition, production, and manufacturing of this work will not be liable for damages of any kind arising out of the use of (or the inability to use) the algorithms, source code, computer programs, or textual material contained in this publication. This includes, but is not limited to, loss of revenue or profit, or other incidental, physical, or consequential damages arising out of the use of this Work.

The sole remedy in the event of a claim of any kind is expressly limited to replacement of the book, and only at the discretion of the Publisher. The use of "implied warranty" and certain "exclusions" vary from state to state, and might not apply to the purchaser of this product.

Software Quality Assurance

A Self-Teaching Introduction

RAJIV CHOPRA, PhD

Mercury Learning and Information
Dulles, Virginia
Boston, Massachusetts
New Delhi

Copyright ©2018 by Mercury Learning and Information LLC. All rights reserved.

Original Title and Copyright: *Software Testing and Quality Assurance: A Practical Approach, 5/e*.
© 2016 by S.K. Kataria & Sons.

This publication, portions of it, or any accompanying software may not be reproduced in any way, stored in a retrieval system of any type, or transmitted by any means, media, electronic display or mechanical display, including, but not limited to, photocopy, recording, Internet postings, or scanning, without prior permission in writing from the publisher.

Publisher: David Pallai
Mercury Learning and Information
22841 Quicksilver Drive
Dulles, VA 20166
info@merclearning.com
www.merclearning.com
(800) 232-0223

R. Chopra. *Software Quality Assurance: A Self-Teaching Introduction*.
ISBN: 978-1-683921-68-4

The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products. All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks, etc. is not an attempt to infringe on the property of others.

Library of Congress Control Number: 2017960720

181920321 This book is printed on acid-free paper in the United States of America

Our titles are available for adoption, license, or bulk purchase by institutions, corporations, etc. For additional information, please contact the Customer Service Dept. at 800-232-0223(toll free).

All of our titles are available in digital format at *authorcloudware.com* and other digital vendors. The sole obligation of Mercury Learning and Information to the purchaser is to replace the book, based on defective materials or faulty workmanship, but not based on the operation or functionality of the product.

CONTENTS

Chapter 1: Introduction to Software Testing **1**

1.0. Introduction 1
1.1. The Testing Process 2
1.2. What is Software Testing? 2
1.3. Why Should We Test? What is the Purpose? 6
1.4. Who Should do Testing? 9
1.5. How Much Should We Test? 9
1.6. Selection of Good Test Cases 9
1.7. Measurement of Testing 10
1.8. Incremental Testing Approach 10
1.9. Basic Terminology Related to Software Testing 11
1.10. Testing Life Cycle 17
1.11. When to Stop Testing? 18
1.12. Principles of Testing 18
1.13. Limitations of Testing 19
1.14. Available Testing Tools, Techniques, and Metrics 20
Summary 20
Multiple Choice Questions 21
Answers 22
Conceptual Short Questions With Answers 22
Review Questions 26

Chapter 2: Software Verification and Validation **29**
2.0. Introduction 29
2.1. Differences between Verification and Validation 30
2.2. Differences between QA and QC? 31
2.3. Evolving Nature of Area 31
2.4. V&V Limitations 32
2.5. Categorizing V&V Techniques 33
2.6. Role of V&V in SDLC—Tabular Form [IEEE std. 1012] 33
2.7. Proof of Correctness (Formal Verification) 37
2.8. Simulation and Prototyping 38
2.9. Requirements Tracing 38
2.10. Software V&V Planning (SVVP) 39
2.11. Software Technical Reviews (STRs) 43
2.11.1. Rationale for STRs 43
2.11.2. Types of STRs 45
2.11.3. Review Methodologies 46
2.12. Independent V&V Contractor (IV&V) 47
2.13. Positive and Negative Effects of Software V&V on Projects 48
2.14. Standard for Software Test Documentation (IEEE829) 50
Summary 57
Multiple Choice Questions 58
Answers 59
Conceptual Short Questions With Answers 59
Review Questions 62

Chapter 3: Software Quality **65**
3.0. Introduction 65
3.1. Role of Process in Software Quality 66
3.2. Software Control 66
3.3. Quality Assurance (QA) 67
3.4. QA Analyst 68
3.5. Quality Factor(s) 68
3.6. Quality Management 69

3.7. Methods of Quality Management 70
3.8. Core Components of Quality 71
3.9. Core Aspects of Quality 72
Summary 72
Multiple Choice Questions 73
Answers 74
Conceptual Short Questions With Answers 74
Chapter 4: Black-Box (or Functional) Testing Techniques 77
4.0. Introduction to Black-Box (or Functional Testing) 77
4.1. Boundary Value Analysis (BVA) 78
4.1.1. What is BVA? 78
4.1.2. Limitations of BVA 79
4.1.3. Robustness Testing 79
4.1.4. Worst-Case Testing 80
4.1.5. Examples with Their Problem Domain 81
4.1.5.1. Test cases for the triangle problem 81
4.1.5.2. Test cases for next date function 83
4.1.5.3. Test cases for the commission problem 84
4.1.6. Guidelines for BVA 86
4.2. Equivalence Class Testing 86
4.2.1. Weak Normal Equivalence Class Testing 87
4.2.2. Strong Normal Equivalence Class Testing 88
4.2.3. Weak Robust Equivalence Class Testing 88
4.2.4. Strong Robust Equivalence Class Testing 89
4.2.5. Solved Examples 90
4.2.5.1. Equivalence class test cases for the triangle problem 90
4.2.5.2. Equivalence class test cases for next date function 91
4.2.5.3. Equivalence class test cases for the commission problem 96
4.2.6. Guidelines for Equivalence Class Testing 97

4.3. Decision Table Based Testing 98
4.3.1. What are Decision Tables? 98
4.3.2. Advantages, Disadvantage, and Applications of Decision Tables 99
4.3.3. Examples 102
4.3.3.1. Test cases for the triangle problem using decision table based testing technique 102
4.3.3.2. Test cases for next date function 103
4.3.3.3. Test cases for the commission problem 108
4.3.4. Guidelines for Decision Table Based Testing 108
4.4. Cause-Effect Graphing Technique 109
4.4.1. Causes and Effects 109
4.4.2. Test Cases for the Triangle Problem 110
4.4.3. Test Cases for Payroll Problem 112
4.4.4. Guidelines for the Cause-Effect Functional Testing Technique 113
4.5. Comparison on Black-Box (or Functional) Testing Techniques 114
4.5.1. Testing Effort 114
4.5.2. Testing Efficiency 116
4.5.3. Testing Effectiveness 116
4.5.4. Guidelines for Functional Testing 117
4.6. Kiviat Charts 117
4.6.1. The Concept of Balance 119
Summary 127
Multiple Choice Questions 127
Answers 129
Conceptual Short Questions With Answers 129
Review Questions 139
Chapter 5: White-Box (or Structural) Testing Techniques 145
5.0. Introduction to White-Box Testing or Structural Testing or Clear-Box or Glass-Box or Open-Box Testing 145

5.1. Static Versus Dynamic White-Box Testing 146
5.2. Dynamic White-Box Testing Techniques 147
5.2.1. Unit/Code Functional Testing 147
5.2.2. Code Coverage Testing 148
5.2.2.1. Statement coverage 148
5.2.2.2. Path coverage 150
5.2.2.3. Condition coverage 151
5.2.2.4. Function coverage 152
5.2.3. Code Complexity Testing 153
5.2.3.1. Cyclomatic complexity, its properties and its meaning in tabular form 153
5.2.3.2. Cyclomatic complexity, its properties and its meaning in tabular form 159
5.2.3.3. Basis path testing with solved examples 161
5.2.3.4. DD path testing with solved examples 177
5.2.3.5. Graph matrices technique with solved examples 181
5.2.3.6. Data flow testing with examples 183
5.3. Mutation Testing Versus Error Seeding — Differences in Tabular Form 198
5.4. Comparison of Black-Box and White-Box Testing in Tabular Form 200
5.5. Practical Challenges in White-Box Testing 202
5.6. Comparison on Various White-Box Testing Techniques 202
5.7. Advantages of White-Box Testing 203
Summary 204
Multiple Choice Questions 204
Answers 206
Conceptual Short Questions With Answers 206
Review Questions 212

Chapter 6: Gray-Box Testing 219
6.0. Introduction to Gray-Box Testing 219
6.1. What is Gray-Box Testing? 220

6.2. Various Other Definitions of Gray-Box Testing 220
6.3. Comparison of White-Box, Black-Box, and Gray-Box Testing Approaches in Tabular Form 221
Summary 223
Multiple Choice Questions 223
Answers 224
Conceptual Short Questions With Answers 224
Review Questions 225

Chapter 7: Levels of Testing 227
7.0. Introduction 227
7.1. Unit, Integration, System, and Acceptance Testing Relationship 228
7.2. Integration Testing 229
7.2.1. Classification of Integration Testing 230
7.2.2. Decomposition-Based Integration 230
7.2.2.1. Types of decomposition-based techniques top-down integration approach 230
7.2.2.2. Bottom-up integration approach 231
7.2.2.3. Sandwich integration approach 231
7.2.2.4. Big-bang integration 232
7.2.2.5. Pros and cons of decomposition-based techniques 232
7.2.2.6. Guidelines to choose integration method and conclusions 233
7.2.3. Call Graph-Based Integration 233
7.2.3.1. Pairwise integration 233
7.2.3.2. Neighborhood integration 234
7.2.3.3. Pros and cons 234
7.2.4. Path-Based Integration with its Pros and Cons 235
7.2.5. System Testing 238
7.2.5.1. What is system testing? 238
7.2.5.2. Why is system testing done? 239

7.2.5.3. Functional versus non functional system testing (in tabular form) 240
7.2.5.4. Functional system testing techniques 242
7.2.5.5. Non functional testing techniques 250
7.2.5.6. Acceptance testing 260
7.2.5.7. Performance testing 265
Summary 283
Multiple Choice Questions 284
Answers 285
Conceptual Short Questions With Answers 285
Review Questions 290

Chapter 8: Quality Assurance 293
8.0. Introduction 293
8.1. Quality Planning 294
8.2. Quality Plan Objectives 295
8.3. Planning Process Overview 295
8.4. Business Plan and Quality Plan 296
8.5. Total Quality Management (TQM) 297
8.6. TQM Concepts 300
8.7. Zero Defect Movement 302
Summary 302
Multiple Choice Questions 302
Answers 304
Conceptual Short Questions With Answers 304
Review Questions 307

Chapter 9: Quality Standards 309
9.0. Introduction 309
9.1. Quality Models/Standards/Guidelines 309
9.2. Types of Models 310
9.3. ISO Standards 317
9.4. CMM and CMMI 319

9.5. Six Sigma Concepts 325
9.6. Quality Challenge 342
9.7. National Quality Awards 343
Summary 345
Multiple Choice Questions 345
Answers 347
Conceptual Short Questions With Answers 347
Review Questions 349

Chapter 10: Reducing the Number of Test Cases 351

10.0. Prioritization Guidelines 351
10.1. Priority Category Scheme 352
10.2. Risk Analysis 353
10.3. Regression Testing—Overview 356
10.3.1. Differences between Regression and Normal Testing 356
10.3.2. Types of Regression Testing 357
10.3.2.1. Corrective regression testing 357
10.3.2.2. Progressive regression testing 357
10.3.2.3. The retest-all strategy 357
10.3.2.4. The selective strategy 357
10.3.2.5. Regression testing at unit level 358
10.3.2.6. Regression testing at integration level 358
10.3.2.7. Regression testing at system level 359
10.3.2.8. Regression testing of global variables 359
10.3.2.9. Comparison of various regression testing techniques 359
10.3.2.10. Regression testing in object-oriented software 360
10.4. Prioritization of Test Cases for Regression Testing 360
10.5. Regression Testing Technique—A Case Study 361
10.6. Slice-Based Testing 362
Summary 364

Multiple Choice Questions 364
Answers 366
Conceptual Short Questions With Answers 367
Review Questions 369

Chapter 11: Test Management and Automation 371
11.0. Automated Testing 371
11.1. Consideration During Automated Testing 372
11.2. Static and Dynamic Analysis Tools 374
11.3. Problems with Manual Testing 375
11.4. Scope of Automated Testing 376
11.5. Disadvantages of Automated Testing 377
11.6. Testing Tools 378
11.7. Test Automation: "No Silver Bullet" 379
11.8. Testing and Debugging 379
11.9. Criteria for the Selection of Test Tools 384
11.10. Design and Architecture for Automation 386
11.11. Characteristics of Modern Testing Tools 387
11.12. Case Study on Automated Tools, Namely, Rational Robot, WinRunner, Silk Test, and Load Runner 387
Summary 390
Multiple Choice Questions 391
Answers 392
Conceptual Short Questions With Answers 392
Review Questions 393

Chapter 12: A Case Study on Testing of E-Learning Management Systems 397
1 Introduction 398
2 Software Requirement Specifications 399
2.1. Introduction 399
2.1.1. Purpose 399
2.1.2. Scope 399
2.1.3. Definitions, Acronyms, and Abbreviations 400

2.1.4. References Books 400
2.1.5. Overview 401
2.2. Overall Descriptions 401
2.2.1. Product Perspective 401
2.2.1.1. System interfaces 401
2.2.1.2. User interfaces 401
2.2.1.3. Hardware interfaces 402
2.2.1.4. Software interfaces 402
2.2.1.5. Communication interfaces 402
2.2.1.6. Memory constraints 402
2.2.1.7. Site adaptation requirements 402
2.2.2. Product Functions 402
2.2.3. User Characteristics 403
2.2.4. Constraints 403
2.2.5. Assumptions and Dependencies 403
2.2.6. Apportioning of Requirements 403
2.3. Specific Requirements 404
2.3.1. User Interfaces and Validations 404
2.3.2. Functions 413
2.3.3. Modules 413
2.3.4. Performance Requirements 414
2.3.5. Logical Database Requirements 414
2.3.6. Design Constraints 414
2.3.7. Software System Attributes 414
2.4. Change Management Process 415
2.5. Document Approval 415
2.6. Supporting Information 415
3 System Design 416
4 Reports and Testing 417
4.1. Test Report 417
4.2. Testing 417
4.2.1. Types of Testing 417
4.2.2. Levels of Testing 418

5 Test Cases 420
5.1. Return Filed Report 420
5.2. Monthly/Quarterly Tax Paid Form 436
5.3. Monthly/Quarterly Tax Paid Form 438
5.4. Monthly /Quarterly Tax Paid Form 442
5.5. Service Wise Report (Admin Report) 444
5.6. STRPs Wise Report (Admin Report) 448
Conclusion 449

Chapter 13: Object-Oriented Testing 451
13.0. Basic Unit for Testing, Inheritance, and Testing 452
13.1. Basic Concepts of State Machines 460
13.2. Testing Object-Oriented Systems 483
13.2.1. Implementation-Based Class Testing/White-Box or Structural Testing 483
13.2.2. Responsibility-Based Class Testing/Black-Box/ Functional Specification-Based Testing of Classes 495
13.3. Heuristics for Class Testing 506
13.4. Levels of Object-Oriented Testing 513
13.5. Unit Testing a Class 514
13.6. Integration Testing of Classes 517
13.7. System Testing (With Case Study) 521
13.8. Regression and Acceptance Testing 531
13.9. Managing the Test Process 533
13.10. Design For Testability (DFT) 537
13.11. GUI Testing 540
13.12. Comparison of Conventional and Object-Oriented Testing 540
13.13. Testing Using Orthogonal Arrays 542
13.14. Test Execution Issues 544
13.15. Case Study—Currency Converter Application 544
Summary 553
Multiple Choice Questions 554
Answers 555

Conceptual Short Questions With Answers 556
Review Questions 558

Chapter 14: The Game Testing Process **559**
14.0. "Black-Box" Testing 560
14.1. "White-Box" Testing 562
14.2. The Life Cycle of a Build 563
14.3. On Writing Bugs Well 573
Exercises 580

Chapter 15: Basic Test Plan Template **583**

Chapter 16: Game Testing By the Numbers **591**
16.0. Testing progress 591
16.1. Testing Effectiveness 595
16.2. Tester Performance 598
Exercises 600

Appendix A: Quality Assurance and Testing Tools 603
Appendix B: Sample Project Description 611
Appendix C: Glossary 617
Appendix D: Bibliography 653

Index 655

CHAPTER 1

INTRODUCTION TO SOFTWARE TESTING

Inside this Chapter:

1.0. Introduction
1.1. The Testing Process
1.2. What is Software Testing?
1.3. Why Should We Test? What is the Purpose?
1.4. Who Should Do Testing?
1.5. What Should We Test?
1.6. Selection of Good Test Cases
1.7. Measurement of the Progress of Testing
1.8. Incremental Testing Approach
1.9. Basic Terminology Related to Software Testing
1.10. Testing Life Cycle
1.11. When to Stop Testing?
1.12. Principles of Testing
1.13. Limitations of Testing
1.14. Available Testing Tools, Techniques, and Metrics

1.0. INTRODUCTION

Testing is the process of executing the program with the intent of finding faults. Who should do this testing and when should it start are very important questions that are answered in this text. As we know software testing is the

fourth phase of the software development life cycle (SDLC). About 70% of development time is spent on testing. We explore this and many other interesting concepts in this chapter.

1.1. THE TESTING PROCESS

Testing is different from debugging. Removing errors from your programs is known as *debugging* but testing aims to locate as yet undiscovered errors. We test our programs with both valid and invalid inputs and then compare our expected outputs as well as the observed outputs (after execution of software). Please note that testing starts from the requirements analysis phase only and goes until the last maintenance phase. During requirement analysis and designing we do *static testing* wherein the SRS is tested to check whether it is as per user requirements or not. We use techniques of code reviews, code inspections, walkthroughs, and software technical reviews (STRs) to do static testing. *Dynamic testing* starts when the code is ready or even a unit (or module) is ready. It is dynamic testing as now the code is tested. We use various techniques for dynamic testing like black-box, gray-box, and white-box testing. We will be studying these in the subsequent chapters.

1.2. WHAT IS SOFTWARE TESTING?

The concept of software testing has evolved from simple program "check-out" to a broad set of activities that cover the entire software life-cycle.

There are five distinct levels of testing that are given below:

a. **Debug:** It is defined as the successful correction of a failure.

b. **Demonstrate:** The process of showing that major features work with typical input.

c. **Verify:** The process of finding as many faults in the application under test (AUT) as possible.

d. **Validate:** The process of finding as many faults in requirements, design, and AUT.

e. **Prevent:** To avoid errors in development of requirements, design, and implementation by self-checking techniques, including "test before design."

There are various definitions of testing that are given below:

> *"Testing is the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements."*
>
> *[IEEE 83a]*

OR

> *"Software testing is the process of executing a program or system with the intent of finding errors."*
>
> *[Myers]*

OR

> *"It involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results."*
>
> *[Hetzel]*

Testing is NOT:

a. The process of demonstrating that errors are not present.

b. The process of showing that a program performs its intended functions correctly.

c. The process of establishing confidence that a program does what it is supposed to do.

So, all these definitions are incorrect. Because, with these as guidelines, one would tend to operate the system in a normal manner to see if it works. One would unconsciously choose such normal/correct test data as would prevent the system from failing. Besides, it is not possible to certify that a system has no errors—simply because it is almost impossible to detect all errors.

So, simply stated: *"Testing is basically a task of locating errors."*

It may be:

a. **Positive testing:** Operate the application as it should be operated. Does it behave normally? Use a proper variety of legal test data, including data values at the boundaries to test if it fails. Check actual test results with the expected. Are results correct? Does the application function correctly?

b. **Negative testing:** Test for abnormal operations. Does the system fail/crash? Test with illegal or abnormal data. Intentionally attempt to make things go wrong and to discover/detect—"Does the program do what it should not do? Does it fail to do what it should?"

c. **Positive view of negative testing:** The job of testing is to discover errors before the user does. A good tester is one who is successful in making the system fail. Mentality of the tester has to be destructive—opposite to that of the creator/author, which should be constructive.

One very popular equation of software testing is:

Software Testing = Software Verification + Software Validation

As per IEEE definition(s):

Software verification: *"It is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase."*

OR

"It is the process of evaluating, reviewing, inspecting and doing desk checks of work products such as requirement specifications, design specifications and code."

OR

"It is a human testing activity as it involves looking at the documents on paper."

Whereas **software validation:** *"It is defined as the process of evaluating a system or component during or at the end of development process to determine whether it satisfies the specified requirements. It involves executing the actual software. It is a computer based testing process."*

Both verification and validation (V&V) are complementary to each other.

As mentioned earlier, good testing expects more than just running a program. We consider a leap-year function working on MS SQL (server data base):

```
CREATE FUNCTION  f_is_leap_year (@ ai_year small int)
RETURNS small int
AS
BEGIN
   --if year is illegal (null or –ve), return –1
   IF (@ ai_year IS NULL) or
```

```
        (@ ai_year <= 0) RETURN -1
   IF ((((@ ai_year % 4) = 0) AND
        ((@ ai_year % 100)< > 0)) OR
        ((@ ai_year % 400) = 0)
        RETURN 1  — leap year
   RETURN 0  --Not a leap year
END
```

We execute above program with number of inputs:

TABLE 1.1 Database Table: Test_leap_year

Serial no.	Year (year to test)	Expected result	Observed result	Match
1.	–1	–1	–1	Yes
2.	–400	–1	–1	Yes
3.	100	0	0	Yes
4.	1000	0	0	Yes
5.	1800	0	0	Yes
6.	1900	0	0	Yes
7.	2010	0	0	Yes
8.	400	1	1	Yes
9.	1600	1	1	Yes
10.	2000	1	1	Yes
11.	2400	1	1	Yes
12.	4	1	1	Yes
13.	1204	1	1	Yes
14.	1996	1	1	Yes
15.	2004	1	1	Yes

In this database table given above there are 15 test cases. But these are not sufficient as we have not tried with all possible inputs. We have not considered the trouble spots like:

i. Removing statement (@ ai_year % 400 = 0) would result in Y2K problem.

ii. Entering year in float format like 2010.11.

iii. Entering year as a character or as a string.

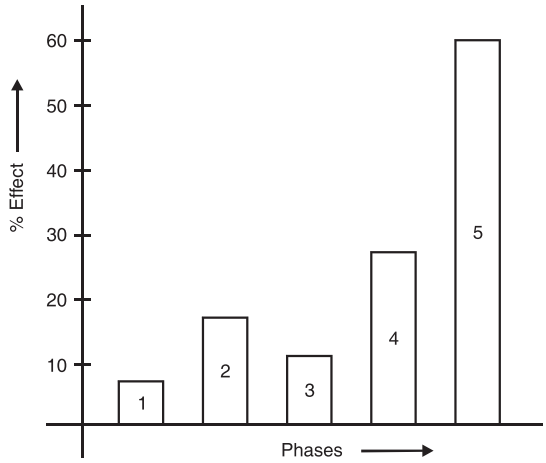
iv. Entering year as NULL or zero (0).

This list can also grow further. These are our *trouble spots* or *critical areas*. We wish to locate these areas and fix these problems before our customer does.

1.3. WHY SHOULD WE TEST? WHAT IS THE PURPOSE?

Testing is necessary. Why?

1. The Technical Case:
   - **a.** Competent developers are not inflallible.
   - **b.** The implications of requirements are not always forseeable.
   - **c.** The behavior of a system is not necessarily predictable from its components.
   - **d.** Languages, databases, user interfaces, and operating systems have bugs that can cause application failures.
   - **e.** Reusable classes and objects must be trustworthy.
2. The Business Case:
   - **a.** If you don't find bugs your customers or users will.
   - **b.** Post-release debugging is the most expensive form of development.
   - **c.** Buggy software hurts operations, sales, and reputation.
   - **d.** Buggy software can be hazardous to life and property.
3. The Professional Case:
   - **a.** Test case design is a challenging and rewarding task.
   - **b.** Good testing allows confidence in your work.
   - **c.** Systematic testing allows you to be most effective.
   - **d.** Your credibility is increased and you have pride in your efforts.
4. The Economics Case: Practically speaking, defects get introduced in every phase of SDLC. Pressman has described a defect amplification model wherein he says that errors get amplified by a certain factor if that error is not removed in that phase only. This may increase the cost of defect removal. This principle of detecting errors as close to their point of introduction as possible is known as *phase containment of errors.*



1. Requirements Analysis; 2. Design;
3. Implementation; 4. Testing; 5. Maintenance

FIGURE 1.1 Efforts During SDLC.

5. To Improve Quality: As computers and software are used in critical applications, the outcome of a bug can be severe. Bugs can cause huge losses. Bugs in critical systems have caused airplane crashes, allowed space shuttle systems to go awry, and halted trading on the stock market. Bugs can kill. Bugs can cause disasters.

   In a computerized embedded world, the quality and reliability of software is a matter of life and death. This can be achieved only if thorough testing is done.

6. For Verification and Validation (V&V): Testing can serve as metrics. It is heavily used as a tool in the V&V process. We can compare the quality among different products under the same specification based on results from the same test.

   Good testing can provide measures for all relevant quality factors.

7. For Reliability Estimation: Software reliability has important relationships with many aspects of software, including the structure and the amount of testing done to the software. Based on an operational profile (an estimate of the relative frequency of use) of various inputs to the program, testing can serve as a statistical sampling method to gain failure data for reliability estimation.

Recent Software Failures

a. May 31st, 2012, HT reports the failure of air traffic management software, Auto Trac-III, at Delhi Airport. The system is unreliable. This ATC software was installed in 2010 as Auto Trac-II (its older version). Since then it has faced many problems due to inadequate testing. Some of the snags were:

1. May 28, 2011, snag hits radar system of ATC.
2. Feb. 22, 2011, ATC goes blind for 10 minutes with no data about arriving or departing flights.
3. July 28, 2010, radar screens at ATC go blank for 25 minutes after system displaying flight data crashes.
4. Feb. 10, 2010, one of the radar scopes stops working at ATC.
5. Jan. 27, 2010, a screen goes blank at ATC due to a technical glitch.
6. Jan. 15, 2010, radar system collapses due to software glitch. ATC officials manually handle the aircraft.

b. The case of a 2010 Toyota Prius that had a software bug that caused braking problems on bumpy roads.

c. In another case of Therac-25, 6 cancer patients were given overdose.

d. A breach on play station network caused a loss of $170 million to Sony Corp.

Why it happened?

As we know software testing constitutes about 40% of overall effort and 25% of the overall software budget. Software defects are introduced during SDLC due to poor quality requirements, design, and code. Sometimes due to the lack of time and inadequate testing, some of the defects are left behind, only to be found later by users. Software is a ubiquitous product; 90% of people use software in their everyday life. Software has high failure rates due to the poor qualify of the software.

Smaller companies that don't have deep pockets can get wiped out because they did not pay enough attention to software quality and conduct the right amount of testing.

1.4. WHO SHOULD DO TESTING?

As mentioned earlier, testing starts right from the very beginning. This implies that testing is everyone's responsibility. By "everyone," we mean all project team members. So, we cannot rely on one person only. Naturally, it is a *team effort.* We cannot only designate the tester responsible. Even the developers are responsible. They build the code but do not indicate any errors as they have written their own code.

1.5. HOW MUCH SHOULD WE TEST?

Consider that there is a `while_loop` that has three paths. If this loop is executed twice, we have (3×3) paths and so on. So, the total number of paths through such code will be:

$$= 1 + 3 + (3 \times 3) + (3 \times 3 \times 3) + \ldots$$
$$= 1 + \sum 3^n \qquad \text{(where } n > 0\text{)}$$

This means an infinite number of test cases. Thus, testing is not 100% exhaustive.

1.6. SELECTION OF GOOD TEST CASES

Designing a good test case is a complex art. It is complex because:

a. Different types of test cases are needed for different classes of information.

b. All test cases within a test suite will not be good. Test cases may be good in variety of ways.

c. People create test cases according to certain testing styles like domain testing or risk-based testing. And good domain tests are different from good risk-based tests.

Brian Marick coined a new term to a lightly documented test case—the test idea. According to Brian, *"A test idea is a brief statement of something that should be tested."* For example, if we are testing a square-root function, one test idea would be—"test a number less than zero." The idea here is again to check if the code handles an error case.

Cem Kaner said—*"The best test cases are the ones that find bugs."* Our efforts should be on the test cases that finds issues. Do broad or deep coverage testing on the trouble spots.

A test case is a question that you ask of the program. The point of running the test is to gain information like whether the program will pass or fail the test.

1.7. MEASUREMENT OF TESTING

There is no single scale that is available to measure the testing progress. A good project manager (PM) wants worse conditions to occur in the very beginning of the project instead of in the later phases. If errors are large in numbers, we can say either testing was not done thoroughly or it was done so thoroughly that all errors were covered. So there is no standard way to measure our testing process. But metrics can be computed at the organizational, process, project, and product levels. Each set of these measurements has its value in monitoring, planning, and control.

NOTE *Metrics is assisted by four core components—schedule quality, resources, and size.*

1.8. INCREMENTAL TESTING APPROACH

To be effective, a software tester should be knowledgeable in two key areas:

1. Software testing techniques
2. The application under test (AUT)

For each new testing assignment, a tester must invest time in learning about the application. A tester with no experience must also learn testing techniques, including general testing concepts and how to define test cases. Our goal is to define a suitable list of tests to perform within a tight deadline. There are 8 stages for this approach:

Stage 1: Exploration
Purpose: To gain familiarity with the application
Stage 2: Baseline test
Purpose: To devise and execute a simple test case
Stage 3: Trends analysis

Purpose: To evaluate whether the application performs as expected when actual output cannot be predetermined

Stage 4: Inventory

Purpose: To identify the different categories of data and create a test for each category item

Stage 5: Inventory combinations

Purpose: To combine different input data

Stage 6: Push the boundaries

Purpose: To evaluate application behavior at data boundaries

Stage 7: Devious data

Purpose: To evaluate system response when specifying bad data

Stage 8: Stress the environment

Purpose: To attempt to break the system

The schedule is tight, so we may not be able to perform all of the stages. The time permitted by the delivery schedule determines how many stages one person can perform. After executing the baseline test, later stages could be performed in parallel if more testers are available.

1.9. BASIC TERMINOLOGY RELATED TO SOFTWARE TESTING

We must define the following terminologies one by one:

1. Error (or mistake or bugs): People make errors. When people make mistakes while coding, we call these mistakes bugs. Errors tend to propagate. A requirements error may be magnified during design and still amplified during coding. So, an error is a mistake during SDLC.

2. Fault (or defect): A missing or incorrect statement in a program resulting from an error is a fault. So, a fault is the representation of an error. Representation here means the mode of expression, such as a narrative text, data flow diagrams, hierarchy charts, etc. Defect is a good synonym for fault. Faults can be elusive. They requires fixes.

3. Failure: A failure occurs when a fault executes. The manifested inability of a system or component to perform a required function within specified limits is known as a failure. A failure is evidenced by incorrect output, abnormal termination, or unmet time and space constraints. It is a dynamic process.

So, Error (or mistake or bug) ⟶ Fault (or defect) ⟶ Failure. For example,

Error (e.g., * replaced by /) ⟶ Defect (e.g., C = A/B) ⟶ (e.g., C = 2 instead of 8)

4. Incident: When a failure occurs, it may or may not be readily apparent to the user. An incident is the symptom associated with a failure that alerts the user to the occurrence of a failure. It is an unexpected occurrence that requires further investigation. It may not need to be fixed.

5. Test: Testing is concerned with errors, faults, failures, and incidents. A test is the act of exercising software with test cases. A test has two distinct goals—to find failures or to demonstrate correct execution.

6. Test case: A test case has an identity and is associated with program behavior. A test case also has a set of inputs and a list of expected outputs. The essence of software testing is to determine a set of test cases for the item to be tested.

   The test case template is shown below.

```
Test Case ID
Purpose
Preconditions
Inputs
Expected Outputs
Postconditions
Execution History
Date        Result        Version        Run By
```

FIGURE 1.2 Test Case Template.

There are 2 types of inputs:

a. Preconditions: Circumstances that hold prior to test case execution.

b. Actual inputs: That were identified by some testing method.

Expected outputs are also of two types:

a. Post conditions

b. Actual outputs

The act of testing entails establishing the necessary preconditions, providing the test case inputs, observing the outputs, and then comparing these with the expected outputs to determine whether the test passed.

The remaining information in a test case primarily supports testing team. Test cases should have an identity and a reason for being. It is also useful to record the execution history of a test case, including when and by whom it was run, the pass/fail result of each execution, and the version of the software on which it was run. This makes it clear that test cases are valuable, at least as valuable as source code. Test cases need to be developed, reviewed, used, managed, and saved. So, we can say that test cases occupy a central position in testing.

Test cases for ATM:

Preconditions: System is started.

Test case ID	Test case name	Test case description	Test steps			Test status (P/F)	Test priority
			Step	Expected result	Actual result		
Session 01	Verify Card	To verify whether the system reads a customer's ATM card.	Insert a readable card	Card is accepted; System asks for entry of PIN			High
			Insert an unreadable card	Card is ejected; System displays an error screen; System is ready to start a new session			High
	Validate PIN	To verify whether the system accepts customer's PIN	Enter valid PIN	System displays a menu of transaction types			High
			Enter invalid PIN	Customer is asked to re-enter PIN			High

(Continued)

Test case ID	Test case name	Test case description	Test steps			Test status (P/F)	Test priority
			Step	Expected result	Actual result		
			Enter incorrect PIN the first time, then correct PIN the second time	System displays a menu of transaction types.			High
			Enter incorrect PIN the first time and second time, then correct PIN the third time	System displays a menu of transaction types.			High
			Enter incorrect PIN three times	An appropriate message is displayed; Card is retained by machine; Session is terminated			High
Session 02	Validate User Session	To verify whether the system allows customer to perform a transaction	Perform a transaction	System asks whether customer wants another transaction			High
		To verify whether the system allows multiple transactions in one session	When system asks for another transaction, answer is yes	System displays a menu of transaction types			High

(Continued)

Test case ID	Test case name	Test case description	Test steps			Test status (P/F)	Test priority
			Step	Expected result	Actual result		
			When system asks for another transaction, answer is no	System ejects card and is ready to start a new session			High
	Verify Transac-tions	To verify whether the system allows to withdraw	Choose withdrawal transaction	System displays a menu of pos-sible withdrawal amounts			High
		To verify whether system performs a legal withdrawal transaction properly	Choose an amount that the system currently has and which is not greater than the account balance	System dispenses this amount of cash; System prints a correct receipt show-ing amount and correct updated balance			High
			Choose an amount greater than what the system currently has	System displays an appropriate message and asks customer to choose a different amount			High
			Press "Cancel" key	System displays an appropriate message and offers the customer the option of choosing another transaction			Medium

(Continued)

Test case ID	Test case name	Test case description	Test steps			Test status (P/F)	Test priority
			Step	Expected result	Actual result		
Deposit 01	Verify Deposit Transac-tion	To verify whether system allows a deposit	Choose deposit transaction	System displays a request for the customer to type the amount			High
		To verify whether the system performs a legal deposit transaction properly	Enter a legitimate amount	System displays entered amount			High
			Receipt	System prints a correct receipt showing the amount and correct updated balance			High
			Press "Can-cel" key	System displays an appropriate message and offers the customer the option of choosing another transaction			Medium

7. Test suite: A collection of test scripts or test cases that is used for validating bug fixes (or finding new bugs) within a logical or physical area of a product. For example, an acceptance test suite contains all of the test cases that were used to verify that the software has met certain predefined acceptance criteria.
8. Test script: The step-by-step instructions that describe how a test case is to be executed. It may contain one or more test cases.

9. Test ware: It includes all of testing documentation created during the testing process. For example, test specification, test scripts, test cases, test data, the environment specification.
10. Test oracle: Any means used to predict the outcome of a test.
11. Test log: A chronological record of all relevant details about the execution of a test.
12. Test report: A document describing the conduct and results of testing carried out for a system.

1.10. TESTING LIFE CYCLE

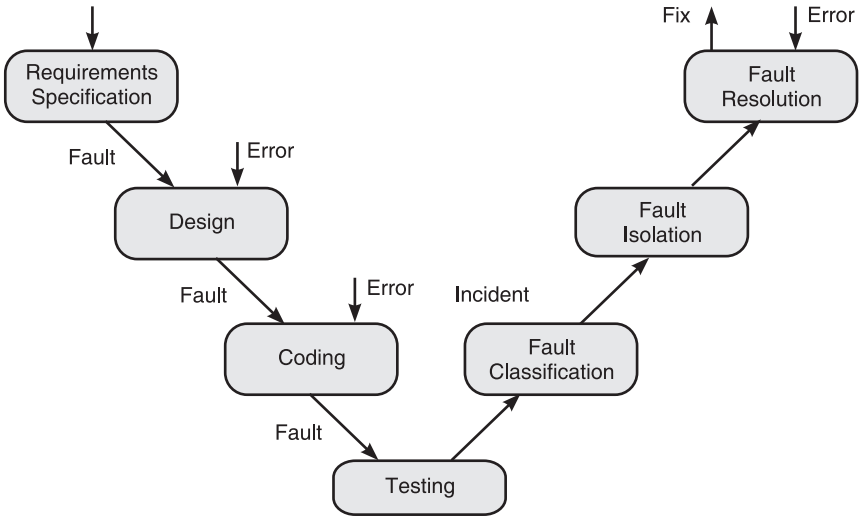


FIGURE 1.3 A Testing Life Cycle.

In the development phase, three opportunities arise for errors to be made resulting in faults that propagate through the remainder of the development process. The first three phases are putting bugs IN, the testing phase is finding bugs, and the last three phases are getting bugs OUT. The fault resolution step is another opportunity for errors and new faults. When a fix causes formerly correct software to misbehave, the fix is deficient.

1.11. WHEN TO STOP TESTING?

Testing is potentially endless. We cannot test until all defects are unearthed and removed. It is simply impossible. At some point, we have to stop testing and ship the software. The question is when?

Realistically, testing is a trade-off between budget, time, and quality. It is driven by profit models.

The *pessimistic approach* is to stop testing whenever some or any of the allocated resources—time, budget, or test cases—are exhausted.

The *optimistic stopping rule* is to stop testing when either reliability meets the requirement, or the benefit from continuing testing cannot justify the testing cost. ***[Yang]***

1.12. PRINCIPLES OF TESTING

To make software testing effective and efficient we follow certain principles. These principles are stated below.

1. Testing should be based on user requirements: This is in order to uncover any defects that might cause the program or system to fail to meet the client's requirements.
2. Testing time and resources are limited: Avoid redundant tests.
3. Exhaustive testing is impossible: As stated by Myer, it is impossible to test everything due to huge data space and the large number of paths that a program flow might take.
4. Use effective resources to test: This represents the most suitable tools, procedures, and individuals to conduct the tests. The test team should use tools like:
   a. *Deja Gnu:* It is a testing frame work for interactive or batch-oriented applications. It is designed for regression and embedded system testing. It runs on UNIX platform. It is a cross-platform operating system.
5. Test planning should be done early: This is because test planning can begin independently of coding and as soon as the client requirements are set.

6. Testing should begin "in small" and progress toward testing "in large": The smallest programming units (or modules) should be tested first and then expanded to other parts of the system.
7. Testing should be conducted by an independent third party.
8. All tests should be traceable to customer requirements.
9. Assign best people for testing. Avoid programmers.
10. Test should be planned to show software defects and not their absence.
11. Prepare test reports including test cases and test results to summarize the results of testing.
12. Advance test planning is a must and should be updated in a timely manner.

1.13. LIMITATIONS OF TESTING

1. Testing can show the presence of errors—not their absence.
2. No matter how hard you try, you will never find the last bug in an application.
3. The domain of possible inputs is too large to test.
4. There are too many possible paths through the program to test.
5. In short, maximum coverage through minimum test-cases. That is the challenge of testing.
6. Various testing techniques are complementary in nature and it is only through their combined use that one can hope to detect most errors.

NOTE *To see some of the most popular testing tools of 2017, visit the following site: https://www.guru99.com/testing-tools.html*

1.14. AVAILABLE TESTING TOOLS, TECHNIQUES, AND METRICS

There are an abundance of software testing tools that exist. Some of the early tools are listed below:

a. **Mothora:** It is an automated mutation testing tool-set developed at Purdue University. Using Mothora, the tester can create and execute test cases, measure test case adequacy, determine input-output correctness, locate and remove faults or bugs, and control and document the test.

b. **NuMega's Bounds Checker, Rational's Purify:** They are run-time checking and debugging aids. They can both check and protect against memory leaks and pointer problems.

c. **Ballista COTS Software Robustness Testing Harness [Ballista]:** It is a full-scale automated robustness testing tool. It gives quantitative measures of robustness comparisons across operating systems. The goal is to automatically test and harden commercial off-the-shelf (COTS) software against robustness failures.

SUMMARY

1. Software testing is an art. Most of the testing methods and practices are not very different from 20 years ago. It is nowhere near maturity, although there are many tools and techniques available to use. Good testing also requires a tester's creativity, experience, and intuition, together with proper techniques.
2. Testing is more than just debugging. It is not only used to locate errors and correct them. It is also used in validation, verification process, and reliability measurement.
3. Testing is expensive. Automation is a good way to cut down cost and time. Testing efficiency and effectiveness is the criteria for coverage based testing techniques.
4. Complete testing is infeasible. Complexity is the root of the problem.
5. Testing may not be the most effective method to improve software quality.

MULTIPLE CHOICE QUESTIONS

1. Software testing is the process of
   - **a.** Demonstrating that errors are not present
   - **b.** Executing the program with the intent of finding errors
   - **c.** Executing the program to show that it executes as per SRS
   - **d.** All of the above.
2. Programmers make mistakes during coding. These mistakes are known as
   - **a.** Failures
   - **b.** Defects
   - **c.** Bugs
   - **d.** Errors
3. Software testing is nothing else but
   - **a.** Verification only
   - **b.** Validation only
   - **c.** Both verification and validation
   - **d.** None of the above.
4. Test suite is a
   - **a.** Set of test cases
   - **b.** Set of inputs
   - **c.** Set of outputs
   - **d.** None of the above.
5. Which one is not the verification activity?
   - **a.** Reviews
   - **b.** Path testing
   - **c.** Walkthroughs
   - **d.** Acceptance testing
6. A break in the working of a system is called a(n)
   - **a.** Defect
   - **b.** Failure
   - **c.** Fault
   - **d.** Error
7. One fault may lead to
   - **a.** One failure
   - **b.** No failure
   - **c.** Many failures
   - **d.** All of the above.

8. Verification is
   - **a.** Checking product with respect to customer's expectations
   - **b.** Checking product with respect to SRS
   - **c.** Checking product with respect to the constraints of the project
   - **d.** All of the above.
9. Validation is
   - **a.** Checking the product with respect to customer's expectations
   - **b.** Checking the product with respect to specification
   - **c.** Checking the product with respect to constraints of the project
   - **d.** All of the above.
10. Which one of the following is not a testing tool?
   - **a.** Deja Gnu
   - **b.** TestLink
   - **c.** TestRail
   - **d.** SOLARIS

ANSWERS

1. b.	**2.** c.	**3.** c.	**4.** a.
5. d.	**6.** b.	**7.** d.	**8.** b.
9. a.	**10.** d.		

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

Q. 1. Are there some myths associated to software testing?

Ans. Some myths related to software testing are as follows:

1. Testing is a structured waterfall idea: Testing may be purely independent, incremental, and iterative activity. Its nature depends upon the context and adapted strategy.
2. Testing is trivial: Adequate testing means a complete understanding of application under test (AUT) and appreciating testing techniques.

3. Testing is not necessary: One can minimize the programming errors but cannot eliminate them. So, testing is necessary.
4. Testing is time consuming and expensive: We remember a saying—"Pay me now or pay me much more later" and this is true in the case of software testing as well. It is better to apply testing strategies now or else defects may appear in the final product which is more expensive.
5. Testing is destructive process: Testing a software is a diagnostic and creative activity which promotes quality.

Q. 2. Give one example of

a. Interface specification bugs
b. Algorithmic bugs
c. Mechanical bugs

Ans. a. Examples of interface specification bugs are:

i. Mismatch between what the client needs and what the server offers.
ii. Mismatch between requirements and implementation.

b. Examples of algorithmic bugs are:

i. Missing initialization.
ii. Branching errors.

c. Examples of mechanical bugs are:

i. Documentation not matching with the operating procedures.

Q. 3. Why are developers not good testers?

Ans. A person checking his own work using his own documentation has some disadvantages:

i. Misunderstandings will not be detected. This is because the checker will assume that what the other individual heard from him was correct.
ii. Whether the development process is being followed properly or not cannot be detected.

iii. 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.

iv. It may result in underestimation of the need for extensive testing.

v. It discourages the need for allocation of time and effort for testing.

Q. 4. Are there any constraints on testing?

Ans. The following are the constraints on testing:

i. Budget and schedule constraints

ii. Changes in technology

iii. Limited tester's skills

iv. Software risks

Q. 5. What are test matrices?

Ans. A test matrix shows the inter-relationship between functional events and tests. A complete test matrix defines the conditions that must be tested during the test process.

The *left side* of the matrix shows the functional events and the *top* identifies the tests that occur in those events. Within the matrix, cells are the process that needs to be tested. We can even *cascade* these test matrices.

Q. 6. What is a process?

Ans. It is defined as a set of activities that represent the way work is performed. The outcome of a process is usually a product or service. For example,

Examples of IT processes	Outcomes
1. Analyze business needs	Needs statement
2. Run job	Executed job
3. UNIT test	Defect-free unit

Q. 7. Explain PDCA view of a process?

Ans. A PDCA cycle is a conceptual view of a process. It is shown in Figure 1.4.

It has four components:

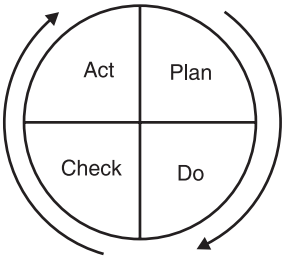


FIGURE 1.4

i. **P-Devise a plan:** Define your objective and find the conditions and methods required to achieve your objective. Express a specific objective numerically.

ii. **D-Execute (or Do) 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 all of the procedures and skills they need to fulfill the plan and a thorough understanding of job is needed. Then they can perform the work according to these procedures.

iii. **C-Check the results:** Check to determine whether work is progressing according to the plan and that the expected results are obtained. Also, compare the results of the work with the objectives.

iv. **A-Take necessary action:** If a check finds out an abnormality, i.e., if the actual value differs from the target value then search for its cause and try to mitigate the cause. This will prevent the recurrence of the defect.

Q. 8. Explain the V-model of testing?

Ans. According to the waterfall model, testing is a post-development activity. The spiral model took one step further by breaking the product into increments each of which can be tested separately. However, V-model brings in a new perspective that different types of testing apply at different levels. The V-model splits testing into two parts—design and execution. *Please note that the test design is done early*

while the test execution is done in the end. This early design of tests reduces overall delay by increasing parallelism between development and testing. It enables better and more timely validation of individual phases. The V-model is shown in Figure 1.5.

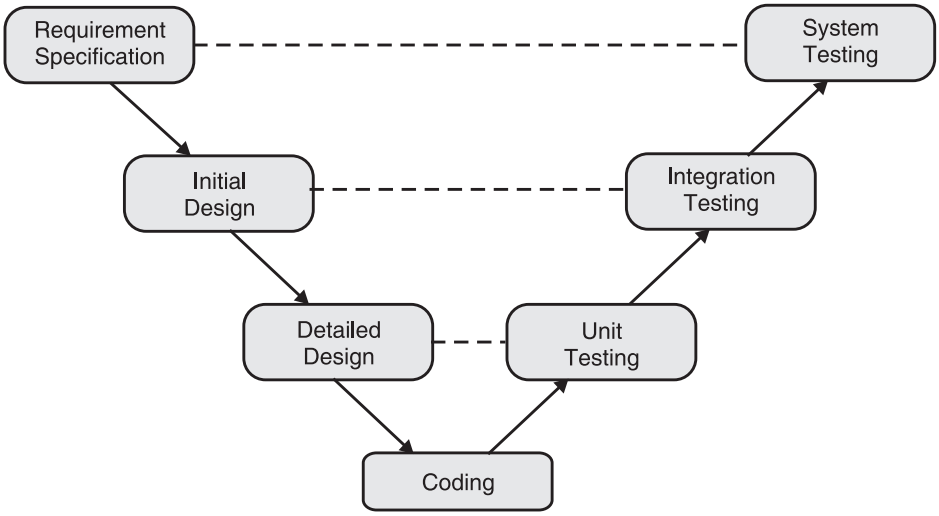


FIGURE 1.5

The levels of testing echo the levels of abstractions found in the waterfall model of SDLC. *Please note here, especially in terms of functional testing, that the three levels of definition, i.e., specification, initial design, and detailed design; correspond directly to three levels of testing—unit, integration, and system testing.*

A practical relationship exists between the levels of testing versus black- and white-box testing. Most practitioners say that structural testing is most appropriate at the unit level while functional testing is most appropriate at the system level.

REVIEW QUESTIONS

1. What is software testing?
2. Distinguish between positive and negative testing?

3. Software testing is software verification plus software validation. Discuss.
4. What is the need of testing?
5. Who should do testing? Discuss various people and their roles during development and testing.
6. What should we test?
7. What criteria should we follow to select test cases?
8. Can we measure the progress of testing?
9. "Software testing is an incremental process." Justify the statement.
10. Define the following terms:
    a. Error
    b. Fault
    c. Failure
    d. Incident
    e. Test
    f. Test case
    g. Test suite
    h. Test script
    i. Testware
11. Explain the testing life cycle?
12. When should we stop testing? Discuss pessimistic and optimistic approaches.
13. Discuss the principles of testing.
14. What are the limitations of testing?
15. a. What is debugging?
    b. Why exhaustive testing is not possible?
16. What are modern testing tools?
17. Write a template for a typical test case.
18. Differentiate between error and fault.
19. What is software testing? How is it different from debugging?
20. Differentiate between verification and validation?
21. Explain the concept of a test case and test plan.

22. **a.** Differentiate between positive testing and negative testing.

    b. Why is 100% testing not possible through either black-box or white-box testing techniques?

    c. Name two testing tools used for functional testing.

    d. What is static testing? Name two techniques to perform static testing.

23. "Software testing is an incremental process." Justify the statement.

24. **a.** Why are developers not good testers?

    b. Which approach should be followed to step testing?

25. **a.** Discuss the role of software testing during the software life cycle and why it is so difficult?

    b. What should we test? Comment on this statement. Illustrate the importance of testing.

    c. Will exhaustive testing (even if possible for very small programs) guarantee that the program is 100% correct?

    d. Define the following terms:

    i. Test suite

    ii. Bug

    iii. Mistake

    iv. Software failure

CHAPTER 2

SOFTWARE VERIFICATION AND VALIDATION

Inside this Chapter:

2.0. Introduction
2.1. Differences Between Verification and Validation
2.2. Differences Between QA and QC
2.3. Evolving Nature of Area
2.4. V&V Limitations
2.5. Categorizing V&V Techniques
2.6. Role of V&V in SDLC—Tabular Form
2.7. Proof of Correctness (Formal Verification)
2.8. Simulation and Prototyping
2.9. Requirements Tracing
2.10. Software V&V Planning (SVVP)
2.11. Software Technical Reviews (STRs)
2.12. Independent V&V (IV&V) Contractor
2.13. Positive and Negative Effects of Software V&V on Projects
2.14. Standard for Software Test Documentation (IEEE829)

2.0. INTRODUCTION

Software that satisfies its user expectations is a necessary goal of a successful software development organization. To achieve this goal, software engineering practices must be applied throughout the evolution of the software

product. Most of these practices attempt to create and modify software in a manner that maximizes the probability of satisfying its user expectations.

2.1. DIFFERENCES BETWEEN VERIFICATION AND VALIDATION

Software verification and validation (V&V) is a technical discipline of systems engineering. According to Stauffer and Fuji (1986), software V&V is *"a systems engineering process employing a rigorous methodology for evaluating the correctness and quality of software product through the software life cycle."*

According to Dr. Berry Boehm (1981), software V&V is performed in parallel with the software development and not at the conclusion of the software development. However, verification and validation are different. Table shows the differences between them.

Verification	Validation
1. It is a static process of verifying documents, design, and code.	**1.** It is a dynamic process of validating/testing the actual product.
2. It does not involve executing the code.	**2.** It involves executing the code.
3. It is human based checking of documents/files.	**3.** It is the computer-based execution of program.
4. Target is requirements specification, application architecture, high level and detailed design, and database design.	**4.** Target is actual product—a unit, a module, a set of integrated modules, and the final product.
5. It uses methods like inspections, walk throughs, desk-checking, etc.	**5.** It uses methods like black-box, gray-box, and white-box testing.
6. It, generally, comes first—before validation.	**6.** It generally follows verification.
7. It answers the question—Are we building the product right?	**7.** It answers the question—Are we building the right product?
8. It can catch errors that validation cannot catch.	**8.** It can catch errors that verification cannot catch.

Both of these are *essential* and *complementary.* Each provides its own sets of *error filters.*

Each has its own way of finding the errors in the software.

2.2. DIFFERENCES BETWEEN QA AND QC?

Quality assurance: The planned and systematic activities implemented in a quality system so that quality requirements for a product or service will be fulfilled is known as *quality assurance.*

Quality control: The observation techniques and activities used to fulfill requirements for quality is known as *quality control.*

Both are, however, different to each other. We tabulate The differences between them are shown below.

Quality Assurance (QA)	Quality Control (QC)
1. It is process related.	**1.** It is product related.
2. It focuses on the process used to develop a product.	**2.** It focuses on testing of a product developed or a product under development.
3. It involves the quality of the processes.	**3.** It involves the quality of the products.
4. It is a preventive control.	**4.** It is a detective control.
5. Allegiance is to development.	**5.** Allegiance is not to development.

2.3. EVOLVING NATURE OF AREA

As the complexity and diversity of software products continue to increase, the challenge to develop new and more effective V&V strategies continues. The V&V approaches that were reasonably effective on small batch-oriented products are not sufficient for concurrent, distributed, or embedded products. Thus, this area will continue to evolve as new research results emerge in response to new V&V challenges.

2.4. V&V LIMITATIONS

The overall objective of software V&V approaches is to ensure that the product is free from failures and meets its user's expectations. There are several theoretical and practical limitations that make this objective impossible to obtain for many products. They are discussed below:

1. Theoretical Foundations

Howden claims the most important theoretical result in program testing and analysis is that no general purpose testing or analysis procedure can be used to prove program correctness.

2. Impracticality of Testing All Data

For most programs, it is impractical to attempt to test the program with all possible inputs due to a combinational explosion. For those inputs selected, a testing oracle is needed to determine the correctness of the output for a particular test input.

3. Impracticality of Testing All Paths

For most programs, it is impractical to attempt to test all execution paths through the product due to a combinational explosion. It is also not possible to develop an algorithm for generating test data for paths in an arbitrary product due to the mobility to determine path feasibility.

4. No Absolute Proof of Correctness

Howden claims that there is no such thing as an absolute proof of correctness. Instead, he suggests that there are proofs of equivalency, i.e., proofs that one description of a product is equivalent to another description. Hence, unless a formal specification can be shown to be correct and, indeed, reflects exactly the user's expectations, no claims of product correctness can be made.

2.5. CATEGORIZING V&V TECHNIQUES

Various V&V techniques are categorized below:

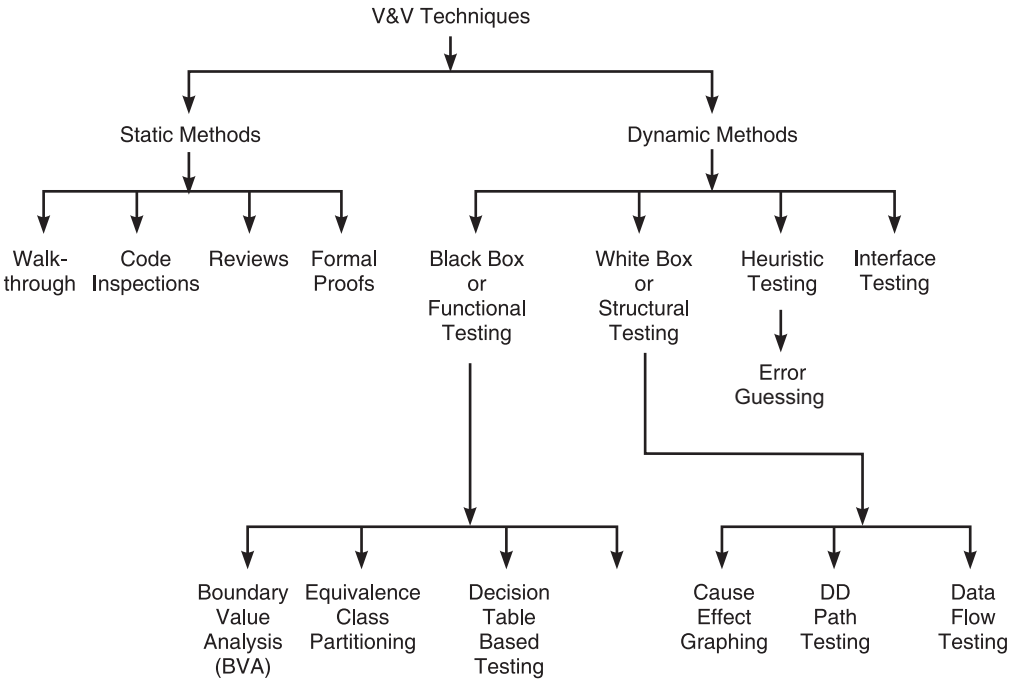


FIGURE 2.1 Types of V&V Techniques.

The static methods of V&V involves the review processes. Whereas dynamic methods like black-box testing can be applied at all levels, even at the system level. While the principle of white-box testing is that it checks for interface errors at the module level, black-box testing can also be done at the module level by testing boundary conditions and low-level functions like correctly displaying error messages.

2.6. ROLE OF V&V IN SDLC—TABULAR FORM *[IEEE STD. 1012]*

Traceability Analysis

It traces each software requirement back to the system requirements established in the concept activity. This is to ensure that each requirement

correctly satisfies the system requirements and that no extraneous software requirements are added. In this technique, we also determine whether any derived requirements are consistent with the original objectives, physical laws, and the technologies described in system document.

Interface Analysis

It is the detailed examination of the interface requirements specifications. The evaluation criteria is the same as that for requirements specification. The main focus is on the interfaces between software, hardware, user, and external software.

Criticality Analysis

Criticality is assigned to each software requirement. When requirements are combined into functions, the combined criticality of requirements form the criticality for the aggregate function. Criticality analysis is updated periodically as requirement changes are introduced. This is because such changes can cause an increase or decrease in a functions criticality which depends on how the revised requirement impacts system criticality.

Criticality analysis is a method used to locate and reduce high-risk problems and is performed at the beginning of the project. It identifies the functions and modules that are required to implement critical program functions or quality requirements like safety, security, etc.

Criticality analysis involves the following steps:

Step 1: Construct a block diagram or control flow diagram (CFD) of the system and its elements. Each block will represent one software function (or module) only.

Step 2: Trace each critical function or quality requirement through CFD.

Step 3: Classify all traced software functions as critical to:

- **a.** Proper execution of critical software functions.
- **b.** Proper execution of critical quality requirements.

Step 4: Focus additional analysis on these traced critical software functions.

Step 5: Repeat criticality analysis for each life cycle process to determine whether the implementation details shift the emphasis of the criticality.

Hazard and Risk Analysis

It is done during the requirements definition activity. Now hazards or risks are identified by further refining the system requirements into detailed software requirements. These risks are assessed for their impact on the system. A summary of these activities is given below:

V&V Activity	V&V Tasks	Key Issues
1. Requirements V&V	▪ Traceability analysis ▪ Software requirements evaluation ▪ Interface analysis ▪ Criticality analysis ▪ System V&V test plan generation ▪ Acceptance V&V test plan generation ▪ Configuration management assessment ▪ Hazard analysis ▪ Risk analysis	▪ Evaluates the correctness, completeness, accuracy, consistency, testability, and readability of software requirements. ▪ Evaluates the software interfaces. ▪ Identifies the criticality of each software function. ▪ Initiates the V&V test planning for the V&V system test. ▪ Initiates the V&V test planning for the V&V acceptance test. ▪ Ensures completeness and adequacy of the SCM process. ▪ Identifies potential hazards based on the product data during the specified development activity. ▪ Identifies potential risks based on the product data during the specified development activity.

(*Continued*)

2. Design V&V	▪ Traceability analysis ▪ Software design evaluation ▪ Interface analysis ▪ Criticality analysis ▪ Component V&V test plan generation and verification ▪ Integration V&V test plan generation and verification ▪ Hazard analysis ▪ Risk analysis	▪ Evaluates software design modules for correctness, completeness, accuracy, consistency, testability, and readability. ▪ Initiates the V&V test planning for the V&V component test. ▪ Initiates the V&V test planning for the V&V integration test.
3. Implementation V&V	▪ Traceability analysis ▪ Source code and source code documentation evaluation ▪ Interface analysis ▪ Criticality analysis ▪ V&V test case generation and verification ▪ V&V test procedure generation and verification ▪ Component V&V test execution and verification ▪ Hazard analysis ▪ Risk analysis	▪ Verifies the correctness, completeness, consistency, accuracy, testability, and readability of source code.
4. Test V&V	▪ Traceability analysis ▪ Acceptance V&V test procedure generation and verification ▪ Integration V&V test execution and verification ▪ System V&V test execution and verification ▪ Acceptance V&V test execution and verification	

5. Maintenance V&V	▪ SVVP (software verification and validation plan) revision ▪ Proposed change assessment ▪ Anomaly evaluation ▪ Criticality analysis ▪ Migration assessment ▪ Retirement assessment ▪ Hazard analysis ▪ Risk analysis	▪ Modifies the SVVP. ▪ Evaluates the effect on software of the operation anomalies. ▪ Verifies the correctness of software when migrated to a different operational environment. ▪ Ensures that the existing system continues to function correctly when specific software elements are retired.

2.7. PROOF OF CORRECTNESS (FORMAL VERIFICATION)

A proof of correctness is a mathematical proof that a computer program or a part thereof will, when executed, yield correct results, i.e., results fulfilling specific requirements. Before proving a program correct, the theorem to be proved must, of course, be formulated.

Hypothesis: The hypothesis of such a correctness theorem is typically a condition that the relevant program variables must satisfy immediately "before" the program is executed. This condition is called the *precondition.*

Thesis: The thesis of the correctness theorem is typically a condition that the relevant program variables must satisfy immediately "after" execution of the program. This latter condition is called the *postcondition.*

So, the correctness theorem is stated as follows:

"If the condition, V, is true before execution of the program, S, then the condition, P, will be true after execution of S."

where V is precondition and P is postcondition.

Notation: Such a correctness theorem is usually written as {V} S {P}, where V, S, and P have been explained above.

By program variable we broadly include input and output data, e.g., data entered via a keyboard, displayed on a screen, or printed on paper. Any externally observable aspect of the program's execution may be covered by the precondition and postcondition.

2.8. SIMULATION AND PROTOTYPING

Simulation and prototying are techniques for analyzing the expected behavior of a product. There are many approaches to constructing simulations and prototypes that are well documented in literature.

For V&V purposes, simulations and prototypes are normally used to analyze requirements and specifications to ensure that they reflect the user's needs. Because they are executable, they offer additional insight into the completeness and correctness of these documents.

Simulations and prototypes can also be used to analyze predicted product performance, especially for candidate product designs, to ensure that they conform to the requirements.

It is important to note that the utilization of simulation and prototyping as a V&V technique requires that the simulations and prototypes themselves be correct. Thus, the utilization of these techniques requires an additional level of V&V activity.

2.9. REQUIREMENTS TRACING

"It is a technique for ensuring that the product, as well as the testing of the product, addresses each of its requirements." The usual approach to performing requirements tracing uses matrices.

a. One type of matrix maps requirements to software modules. Construction and analysis of this matrix can help ensure that all requirements are properly addressed by the product and that the product does not have any superfluous capabilities. System verification diagrams are another way of analyzing requirements/modules traceability.

b. Another type of matrix maps requirements to test cases. Construction and analysis of this matrix can help ensure that all requirements are properly tested.

c. A third type of matrix maps requirements to their evaluation approach. The evaluation approaches may consist of various levels of testing, reviews, simulations, etc.

The requirements/evaluation matrix ensures that all requirements will undergo some form of V&V. Requirements tracing can be applied for all the products of the software evolution process.

2.10. SOFTWARE V&V PLANNING (SVVP)

The development of a comprehensive V&V plan is essential to the success of a project. This plan must be developed early in the project. Depending on the development approach followed, multiple levels of test plans may be developed corresponding to various levels of V&V activities. IEEE 83b has documented the guidelines for the contents of system, software, build, and module test plans.

The following steps for SVVP are listed below:

Step 1: Identification of V&V Goals

V&V goals must be identified from the requirements and specifications. These goals must address those attributes of the product that correspond to its user expectations. These goals must be achievable taking into account both theoretical and practical limitations.

Step 2: Selection of V&V Techniques

Once Step 1 (above) is finished, we must select specific techniques for each of the products that evolves during SDLC. These are given below:

a. **During the Requirements Phase:** The applicable techniques for accomplishing the V&V objectives for requirements are—technical reviews, prototyping, and simulations. The review process is often called a *system requirement review (SRR).*

b. **During the Specifications Phase:** The applicable techniques for this phase are technical reviews, requirements tracing, prototyping, and simulations. The requirements must be traced to the specifications.

c. **During the Design Phase:** The techniques for accomplishing the V&V objectives for designs are technical reviews, requirements tracing,

prototyping, simulation, and proof of correctness. We can go for two types of design reviews:

i. High-level designs that correspond to an architectural view of the product are often reviewed in a *preliminary design review (PDR).*

ii. Detailed designs are addressed by a *critical design review (CDR).*

d. **During the Implementation Phase:** The applicable techniques for accomplishing V&V objectives for implementation are technical reviews, requirements tracing, testing, and proof of correctness. Various code review techniques such as walk throughs and inspections exist.

   At the source-code level, several static analysis techniques are available for detecting implementation errors. The requirements tracing activity is concerned with tracing requirements to source-code modules. The bulk of the V&V activity for source code consists of testing. Multiple levels of testing are usually performed. At the module-level, proof-of-correctness techniques may be applied, if applicable.

e. **During the Maintenance Phase:** Because changes describe modifications to products, the same techniques used for V&V during development may be applied during modification. Changes to implementation require regression testing.

Step 3: Organizational Responsibilities

The organizational structure of a project is a key planning consideration for project managers. An important aspect of this structure is the delegation of V&V activities to various organizations.

This decision is based upon the size, complexity, and criticality of the product. Four types of organizations are addressed. These organizations reflect typical strategies for partitioning tasks to achieve V&V goals for the product. These are:

a. **Developmental Organization:** This type of organization has the following responsibilities:

   1. To participate in technical reviews for all of the evolution products.
   2. To construct prototypes and simulations.
   3. To prepare and execute test plans for unit and integration levels of testing. This is called *preliminary qualification testing (PQT).*
   4. To construct any applicable proofs of correctness at the module level.

b. **Independent Test Organization (ITO):** This type of organization has the following responsibilities:

   1. It enables test activities to occur in parallel with those of development.
   2. It participates in all of the product's technical reviews and monitors PQT effort.
   3. The primary responsibility of the ITO is the preparation and execution of the product's system test plan. This is sometimes referred to as the formal qualification test (FQT).

      The plan for this must contain the equivalent of a requirements/evaluation matrix that defines the V&V approach to be applied for each requirement.
   4. If the product must be integrated with other products, this integration activity is normally the responsibility of the ITO.

c. **Software Quality Assurance (SQA) Organizations:** The intent here is to identify some activities for ensuring software quality. Evaluations are the primary avenue for ensuring software quality. Some evaluation types are given below:

   i. Internal consistency of product
   ii. Understandability of product
   iii. Traceability to indicated documents
   iv. Consistency with indicated documents
   v. Appropriate allocation of sizing, timing, and resources
   vi. Adequate test coverage of requirements
   vii. Completeness of testing
   viii. Completeness of regression testing

d. **Independent V&V Contractor:** An independent contractor may be selected to do V&V. The scope of activities of this organization varies from that of an ITO (discussed above) and SQA organization.

Step 4: Integrating V&V Approaches

Once a set of V&V objectives has been identified, an overall integrated V&V approach must be determined. This approach involves the integration of techniques applicable to various life cycle phases as well as the delegation of these tasks among the project's organizations. The planning of this

integrated V&V approach is very dependent upon the nature of the product and the process used to develop it. Earlier the waterfall approach for testing was used and now incremental approach is used. Regardless of the approach selected, V&V progress must be tracked. Requirements/ evaluation matrices play a key role in this tracking by providing a means of insuring that each requirement of the product is addressed.

Step 5: Problem Tracking

- **i.** It involves documenting the problems encountered during the V&V effort.
- **ii.** Routing these problems to appropriate persons for correctness.
- **iii.** Ensuring that corrections have been done.
- **iv.** Typical information to be collected includes:
  - **a.** When the problem occurred
  - **b.** Where the problem occurred
  - **c.** State of the system before occurrence
  - **d.** Evidence of the problem
  - **e.** Priority for solving problem

NOTE *This fifth step is very important when we go with OS testing.*

Step 6: Tracking Test Activities

SVVP must provide a mechanism for tracking testing effort, testing cost, and testing quality. To do this the following data is collected:

- **a.** Number of tests executed
- **b.** Number of tests remaining
- **c.** Time used
- **d.** Resources used
- **e.** Number of problems found

These data can be used to compare actual test progress against scheduled progress.

Step 7: Assessment

It is important that the software V&V plan provide for the ability to collect data that can be used to assess both the product and the techniques used to develop it. This involves careful collection of error and failure data, as well as analysis and classification of these data.

2.11. SOFTWARE TECHNICAL REVIEWS (STRs)

A review process can be defined as a critical evaluation of an object. It includes techniques such as walk-throughs, inspections, and audits. Most of these approaches involve a group meeting to assess a work product.

Software technical reviews can be used to examine all of the products of the software evolution process. In particular, they are especially applicable and necessary for those products not yet in machine-processable form, such as requirements or specifications written in natural language.

2.11.1. RATIONALE FOR STRs

The main rationale behind STRs are as follows:

a. **Error-Prone Software Development and Maintenance Process:** The complexity and error-prone nature of developing and maintaining software should be demonstrated with statistics depicting error frequencies for intermediate software products. These statistics must also convey the message that errors occur throughout the development process and that the later these errors are detected, the higher the cost for their repair.

Summary:

i. Complexity of software development and maintenance processes.

ii. Error frequencies for software work products.

iii. Error distribution throughout development phases.

iv. Increasing costs for error removal throughout the life cycle.

b. **Inability to Test All Software:** It is not possible to test all software. Clearly exhaustive testing of code is impractical. Current technology

also does not exist for testing a specification or high level design. The idea of testing a software test plan is also bewildering. Testing also does not address quality issues or adherence to standards which are possible with review processes.

Summary:

i. Exhaustive testing is impossible.

ii. Intermediate software products are largely untestable.

c. **Reviews are a Form of Testing:** The degree of formalism, scheduling, and generally positive attitude afforded to testing must exist for software technical reviews if quality products are to be produced.

Summary:

i. Objectives

ii. Human based versus machine based

iii. Attitudes and norms

d. **Reviews are a Way of Tracking a Project:** Through identification of deliverables with well defined entry and exit criteria and successful review of these deliverables, progress on a project can be followed and managed more easily [Fagan]. In essence, review processes provide milestones with teeth. This tracking is very beneficial for both project management and customers.

Summary:

i. Individual developer tracking

ii. Management tracking

iii. Customer tracking

e. **Reviews Provide Feedback:** The instructor should discuss and provide examples about the value of review processes for providing feedback about software and its development process.

Summary:

i. Product

ii. Process

f. **Educational Aspects of Reviews:** It includes benefits like a better understanding of the software by the review participants that can be obtained by reading the documentation as well as the opportunity of acquiring additional technical skills by observing the work of others.

Summary:

i. Project understanding
ii. Technical skills

2.11.2. TYPES OF STRs

A variety of STRs are possible on a project depending upon the developmental model followed, the type of software product being produced, and the standards which must be adhered to. These developmental modes may be

i. Waterfall model
ii. Rapid prototyping
iii. Iterative enhancement
iv. Maintenance activity modeling

And the current standards may be

i. Military standards
ii. IEEE standards
iii. NBS standards

Reviews are classified as formal and Informal reviews. We tabulate the differences between them in tabular form.

Informal reviews	Formal reviews
i. It is a type of review that typically occurs spontaneously among peers.	i. It is a planned meeting.
ii. Reviewers have no responsibility.	ii. Reviewers are held accountable for their participation in the review.
iii. No review reports are generated.	iii. Review reports containing action items are generated and acted upon.

2.11.3. Review Methodologies

There are three approaches to reviews

a. Walkthrough (or presentation reviews)

b. Inspection (or work product reviews)

c. Audits

a. Walkthroughs (or Presentation Reviews) Walkthroughs are well defined by Yourdon. Walkthroughs can be viewed as presentation reviews in which a review participant, usually the developer of the software being reviewed, narrates a description of the software and the remainder of the review group provides their feedback throughout the presentation.

They are also known as *presentation reviews* because the bulk of the feedback usually occurs for the material actually presented at the level it is presented. Thus, advance preparation on the part of reviewers is often not detectable during a walkthrough. Walkthroughs suffer from these limitations as well as the fact that they may lead to disorganized and uncontrolled reviews.

Walkthroughs may also be stressful if the developer of the software is conducting the walkthrough. This has lead to many variations such as having someone other than the developer perform the walkthrough. It is also possible to combine multiple reviews into a single review such as a combined design and code walkthrough.

b. Inspections (or Walk Product Reviews): It is a formal approach. Inspections were first performed by Fagan at IBM. They require a high degree of preparation for the review participants but the benefits include a more systematic review of the software and a more controlled and less stressed meeting.

There are many variations of inspections, but all include some form of a checklist or agenda that guides the preparation of the review participants and serves to organize the review meeting itself. Inspections are also characterized by rigorous entry and exit requirements for the work products being inspected.

An inspection process involves the collection of data that can be used for feedback on the quality of the development and review

processes as well as to influence future validation techniques on the software itself.

Inspections	Walkthroughs
1. It is a five-step process that is well formalized.	**1.** It has fewer steps than inspections and is a less formal process.
2. It uses checklists for locating errors.	**2.** It does not use a checklist.
3. It is used to analyze the quality of the process.	**3.** It is used to improve the quality of the product.
4. This process takes a longer time.	**4.** It is a shorter process.
5. It focuses on training of junior staff.	**5.** It focuses on finding defects.

c. **Audits:** Audits should also be described as an external type of review process. Audits serve to ensure that the software is properly validated and that the process is producing its intended results.

2.12. INDEPENDENT V&V CONTRACTOR (IV&V)

An independent V&V contractor may sometimes be used to ensure independent objectivity and evaluation for the customer.

The use of a different organization, other than the software development group, for software V&V is called independent verification and validation (IV&V). Three types of independence are usually required:

i. **Technical Independence:** It requires that members of the IV&V team (organization or group) may not be personnel involved in the development of the software. This team must have some knowledge about the system design or some engineering background enabling them to understand the system. The IV&V team must not be influenced by the development team when the IV&V team is learning about the system requirements, proposed solutions for building the system, and problems encountered. Technical independence is crucial in the

team's ability to detect the subtle software requirements, software design, and coding errors that escape detection by development testing and SQA reviews.

The technical IV&V team may need to share tools from the computer support environment (e.g., compilers, assemblers, utilities) but should execute qualification tests on these tools to ensure that the common tools themselves do not mask errors in the software being analyzed and tested. The IV&V team uses or develops its own set of test and analysis tools separate from the developer's tools whenever possible.

ii. **Managerial Independence:** It means that the responsibility for IV&V belongs to an organization outside the contractor and program organizations that develop the software. While assurance objectives may be decided by regulations and project requirements, the IV&V team independently decides the areas of the software/system to analyze and test, techniques to conduct the IV&V, schedule of tasks, and technical issues to act on. The IV&V team provides its findings in a timely manner simultaneously to both the development team and the systems management.

iii. **Financial Independence:** It means that control of the IV&V budget is retained in an organization outside the contractor and program organization that develop the software. This independence protects against diversion of funds or adverse financial pressures or influences that may cause delay or stopping of IV&V analysis and test tasks and timely reporting of results.

2.13. POSITIVE AND NEGATIVE EFFECTS OF SOFTWARE V&V ON PROJECTS

Software V&V has some *positive effects* on a software project. The following are given below:

1. Better quality of software. This includes factors like completeness, consistency, readability, and testablity of the software.
2. More stable requirements.
3. More rigorous development planning, at least to interface with the software V&V organization.

4. Better adherence by the development organization to programming language and development standards and configuration management practices.
5. Early error detection and reduced false starts.
6. Better schedule compliance and progress monitoring.
7. Greater project management visibility into interim technical quality and progress.
8. Better criteria and results for decision making at formal reviews and audits.

Some *negative effects* of software V&V on a software development project include:

1. Additional project cost of software V&V (10–30% extra).
2. Additional interface involving the development team, user, and software V&V organization. For example, attendance at software V&V status meetings, anomaly resolution meetings.
3. Additional documentation requirement beyond the deliverable products, if software V&V is receiving incremental program and documentation releases.
4. Need to share computing facilities with and access to classified data for the software V&V organization.
5. Lower development staff productivity if programmers and engineers spend time explaining the system to software V&V analysts, especially if explanations are not documented.
6. Increased paperwork to provide written responses to software V&V error reports and other V&V data requirements. For example, notices of formal review and audit meetings, updates to software release schedule, and response to anomaly reports.
7. Productivity of development staff affected adversely in resolving invalid anomaly reports.

Some steps can be taken to minimize the negative effects and to maximize the positive effects of software V&V. To recover much of the costs, software V&V is started early in the software requirements phase. The interface activities for documentation, data, and software deliveries between developer and software V&V groups should be considered as an inherently necessary step required to evaluate intermediate development products.

To offset unnecessary costs, software V&V must organize its activities to focus on critical areas of the software so that it uncovers critical errors for the development group and thereby results in significant cost savings to the development process. To do this, software V&V must use its criticality analysis to identify critical areas. It must scrutinize each discrepancy before release to ensure that no false or inaccurate information is released to prevent the development group from wasting time on inaccurate or trivial reports.

To eliminate the need to have development personnel train the software V&V staff, it is imperative that software V&V select personnel who are experienced and knowledgeable about the software and its engineering application. When software V&V engineers and computer scientists reconstruct the specific details and idiosyncrasies of the software as a method of reconfirming the correctness of engineering and programming assumptions, they often find subtle errors. They gain detailed insight into the development process and an ability to spot critical errors early. The cost of the development interface is minimal, and at times nonexistent, when the software V&V assessment is independent.

Finally, the discrepancies detected in software and the improvement in documentation quality resulting from error correction suggests that software V&V costs are offset by having more reliable and maintainable software. Many companies rely on their software systems for their daily operations. Failure of the system, loss of data, and release of or tampering with sensitive information may cause serious work disruptions and serious financial impact. The costs of software V&V are offset in many application areas by increased reliability during operation and reduced costs of maintenance.

2.14. STANDARD FOR SOFTWARE TEST DOCUMENTATION *(IEEE829)*

- The IEEE829 standard for software test documentation describes a set of basic software test documents. It defines the content and form of each test document.
- In this addendum, we give a summary of the structure of the most important IEEE829 defined test documents.

- This addendum is based on the course materials by Jukka Paakki (and the IEEE829 standard).

Test Plan

1. Test-plan Identifier: Specifies the unique identifier assigned to the test plan.
2. Introduction: Summarizes the software items and features to be tested, provides references to the documents relevant for testing (for example, overall project plan, quality assurance plan, configuration management plan, applicable standards, etc.).
3. Test Items: Identifies the items to be tested including their version/revision level, provides references to the relevant item documentation (for example, requirements specification, design specification, user's guide, operations guide, installation guide, etc.), and identifies items which are specifically excluded from testing.
4. Features to be Tested: Identifies all software features and their combinations to be tested, identifies the test-design specification associated with each feature and each combination of features.
5. Features not to be Tested: Identifies all features and significant combinations of features which will not be tested, and the reasons for this.
6. Approach: Describes the overall approach to testing (the testing activities and techniques applied, the testing of non functional requirements such as performance and security, the tools used in testing); specifies completion criteria (for example, error frequency or code coverage); identifies significant constraints such as testing-resource availability and strict deadlines; serves for estimating the testing efforts.
7. Item Pass/Fail Criteria: Specifies the criteria to be used to determine whether each test item has passed or failed testing.
8. Suspension Criteria and Resumption: Specifies the criteria used to suspend all or a portion of the testing activity on the test items (for example, at the end of working day, due to hardware failure or other external exception, etc.), specifies the testing activities which must be repeated when testing is resumed.

9. Test Deliverables: Identifies the deliverable documents, typically test-design specifications, test-case specifications, test-procedure specifications, test-item transmittal reports, test logs, test-incident reports, description of test-input data and test-output data, and description of test tools.
10. Testing Tasks: Identifies the set of tasks necessary to prepare and perform testing (for example, description of the main phases in the testing process, design of verification mechanisms, plan for maintenance of the testing environment, etc.).
11. Environmental Needs: Specifies both the necessary and desired properties of the test environment (for example, hardware, communications and systems software, software libraries, test support tools, level of security for the test facilities, drivers and stubs to be implemented, office or laboratory space, etc.).
12. Responsibilities: Identifies the groups of persons responsible for managing, designing, preparing, executing, witnessing, checking, and resolving the testing process; identifies the groups responsible for providing the test items (Section 3) and the environmental needs (Section 11).
13. Staffing and Training Needs: Specifies the number of testers by skill level, and identifies training options for providing necessary skills.
14. Schedule: Includes test milestones (those defined in the overall project plan as well as those identified as internal ones in the testing process), estimates the time required to do each testing task, identifies the temporal dependencies between testing tasks, specifies the schedule over calendar time for each task and milestone.
15. Risks and Contingencies: Identifies the high-risk assumptions of the test plan (lack of skilled personnel, possible technical problems, etc.), specifies contingency plans for each risk (for example, employment of additional testers, increase of night shift, exclusion of some tests of minor importance, etc.).
16. Approvals: Specifies the persons who must approve this plan.

Test-Case Specification

1. Test-case Specification Identifier: Specifies the unique identifier assigned to this test-case specification.
2. Test Items: Identifies and briefly describes the items and features to be exercised by this test case, supplies references to the relevant item documentation (for example, requirements specification, design specification, user's guide, operations guide, installation guide, etc.).
3. Input Specifications: Specifies each input required to execute the test case (by value with tolerance or by name); identifies all appropriate databases, files, terminal messages, memory resident areas, and external values passed by the operating system; specifies all required relationships between inputs (for example, timing).
4. Output Specifications: Specifies all of the outputs and features (for example, response time) required of the test items, provides the exact value (with tolerances where appropriate) for each required output or feature.
5. Environmental Needs: Specifies the hardware and software configuration needed to execute this test case, as well as other requirements (such as specially trained operators or testers).
6. Special Procedural Requirements: Describes any special constraints on the test procedures which execute this test case (for example, special set-up, operator intervention, etc.).
7. Intercase Dependencies: Lists the identifiers of test cases which must be executed prior to this test case, describes the nature of the dependencies.

Test-Incident Report (Bug Report)

1. Bug-Report Identifier: Specifies the unique identifier assigned to this report.
2. Summary: Summarizes the (bug) incident by identifying the test items involved (with version/revision level) and by referencing the relevant documents (for example, test-procedure specification, test-case specification, test log).
3. Bug Description: Provides a description of the incident, so as to correct the bug, repeat the incident, or analyze it offline.

- Inputs
- Expected results
- Actual results
- Date and time
- Test-procedure step
- Environment
- Repeatability (whether repeated; whether occurring always, occasionally, or just once).
- Testers
- Other observers
- Additional information that may help to isolate and correct the cause of the incident; for example, the sequence of operational steps or history of user-interface commands that lead to the (bug) incident.

4. Impact: Priority of solving the incident/correcting the bug (urgent, high, medium, low).

Test-Summary Report

1. Test-Summary-Report Identifier: Specifies the unique identifier assigned to this report.
2. Summary: Summarizes the evaluation of the test items, identifies the items tested (including their version/revision level), indicates the environment in which the testing activities took place, supplies references to the documentation over the testing process (for example, test plan, test-design specifications, test-procedure specifications, test-item transmittal reports, test logs, test-incident reports, etc.).
3. Variances: Reports any variances/deviations of the test items from their design specifications, indicates any variances of the actual testing process from the test plan or test procedures, specifies the reason for each variance.
4. Comprehensiveness Assessment: Evaluates the comprehensiveness of the actual testing process against the criteria specified in the test plan, identifies features or feature combinations which were not sufficiently tested and explains the reasons for omission.
5. Summary of Results: Summarizes the success of testing (such as coverage), identifies all resolved and unresolved incidents.

6. Evaluation: Provides an overall evaluation of each test item including its limitations (based upon the test results and the item-level pass/fail criteria).
7. Summary of Activities: Summarizes the major testing activities and events, summarizes resource consumption (for example, total staffing level, total person-hours, total machine time, and total elapsed time used for each of the major testing activities).
8. Approvals: Specifies the persons who must approve this report (and the whole testing phase).

Inspection Checklist for Test Plans

1. Have all materials required for a test plan inspection been received?
2. Are all materials in the proper physical format?
3. Have all test plan standards been followed?
4. Has the testing environment been completely specified?
5. Have all resources been considered, both human and hardware/software?
6. Have all testing dependencies been addressed (driver function, hardware, etc.)?
7. Is the test plan complete, i.e., does it verify all of the requirements? (For unit testing: does the plan test all functional and structural variations from the high-level and detailed design?)
8. Is each script detailed and specific enough to provide the basis for test case generation?
9. Are all test entrance and exit criteria sufficient and realistic?
10. Are invalid as well as valid input conditions tested?
11. Have all pass/fail criteria been defined?
12. Does the test plan outline the levels of acceptability for pass/fail and exit criteria (e.g., defect tolerance)?
13. Have all suspension criteria and resumption requirements been identified?
14. Are all items excluded from testing documented as such?

15. Have all test deliverables been defined?
16. Will software development changes invalidate the plan? (Relevant for unit test plans only).
17. Is the intent of the test plan to show the presence of failures and not merely the absence of failures?
18. Is the test plan complete, correct, and unambiguous?
19. Are there holes in the plan or is there overlap in the plan?
20. Does the test plan offer a measure of test completeness and test reliability to be sought?
21. Are the test strategy and philosophy feasible?

Inspection Checklist for Test Cases

1. Have all materials required for a test case inspection been received?
2. Are all materials in the proper physical format?
3. Have all test case standards been followed?
4. Are the functional variations exercised by each test case required by the test plan? (Relevant for unit test case documents only.)
5. Are the functional variations exercised by each test case clearly documented in the test case description? (Relevant for unit test case documents only.)
6. Does each test case include a complete description of the expected input and output or result?
7. Have all testing execution procedures been defined and documented?
8. Have all testing dependencies been addressed (driver function, hardware, etc.)?
9. Do the test cases accurately implement the test plan?
10. Are all data set definitions and setup requirements complete and accurate?
11. Are operator instructions and status indicators complete, accurate, and simple?

12. Have all intercase dependencies been identified and described?
13. Is each condition tested once and only once?
14. Have all test entrance and exit criteria been observed?
15. Are the test cases designed to show the presence of failure and not merely the absence of failure?
16. Are the test cases designed to show omissions and extensions?
17. Are the test cases complete, correct, and unambiguous?
18. Are the test cases realistic?
19. Are the test cases documented so as to be 100% reproducible?
20. Has the entire testing environment been documented?
21. Has configuration management been set up, directories established, and have case data and tools been loaded?

SUMMARY

1. Software engineering technology has matured sufficiently to be addressed in approved and draft software engineering standards and guidelines.
2. Business, industries, and government agencies spend billions annually on computer software for many of their functions:
   - To manufacture their products.
   - To provide their services.
   - To administer their daily activities.
   - To perform their short- and long-term management functions.
3. As with other products, industries and businesses are discovering that their increasing dependence on computer technology to perform these functions, emphasizes the need for safe, secure, reliable computer systems. They are recognizing that software quality and reliability are vital to their ability to maintain their competitiveness and high technology posture in the market place. Software V&V is one of several methodologies that can be used for building vital quality software.

MULTIPLE CHOICE QUESTIONS

1. Which one of the following is product related?
   - **a.** Quality control
   - **b.** Quality assurance
   - **c.** Both (a) and (b)
   - **d.** None of the above.
2. Howden claims that
   - **a.** No general purpose testing can be used to prove program correctness.
   - **b.** There is no such thing as an absolute proof of correctness.
   - **c.** Both (a) and (b)
   - **d.** None of the above.
3. Static testing involves
   - **a.** Symbolic execution
   - **b.** Code walkthrough
   - **c.** Inspections
   - **d.** All of the above.
4. The role of V&V in SDLC is given in which of the following standards
   - **a.** IEEE std. 1012
   - **b.** IEEE std. 9012
   - **c.** IEEE std. 83b
   - **d.** None of the above.
5. Detailed designs are addressed by a
   - **a.** Preliminary design review
   - **b.** Critical design review
   - **c.** Both (a) and (b)
   - **d.** None of the above.
6. A planned meeting is also known as a
   - **a.** Informal review
   - **b.** Formal review
   - **c.** Technical review
   - **d.** Dynamic review
7. Structured walkthrough is a
   - **a.** Dynamic testing technique
   - **b.** Formal static testing technique
   - **c.** Informal static testing
   - **d.** Acceptance testing technique

8. Which of the following is not a validation activity?

a. Unit testing **b.** System testing

c. Acceptance testing **d.** Walkthroughs

9. Which of the following is not a verification activity?

a. Acceptance testing **b.** Inspections

c. Walkthroughs **d.** Buddy check

10. During validation

a. Process is checked.

b. Product is checked.

c. Developer's performance is evaluated.

d. The customer checks the product.

ANSWERS

1. a.	**2.** c.	**3.** d.	**4.** a.
5. b.	**6.** b.	**7.** c.	**8.** d.
9. a.	**10.** d.		

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

Q. 1. What sort of errors are covered by regression testing?

Ans. Regression testing includes mainly four types of errors:

i. Data Corruption Errors: Due to sharing of data, these errors result in side effects.

ii. Inappropriate Control Sequencing Errors: Due to the changes in the execution sequences, these errors result in side effects.

For example, an attempt to remove an item from a queue before it is placed into the queue.

iii. Resource Contention: Potential bottlenecks and deadlocks are some examples of these types of errors.

iv. Performance Deficiencies: Timing errors and storage utilization errors are some examples of these types of errors.

Q. 2. What is criticality analysis?

Ans. It is a method to locate and reduce high-risk problems. It is performed at the beginning of the project. It identifies the functions and modules that are required to implement critical program functions or quality requirements like safety, security, etc.

The steps of analysis are as follows:

Step 1: Develop a block diagram or control flow diagram of the system and its software elements. Each block or control flow box represents a system or software function (module).

Step 2: Trace each critical function or quality requirements through the block or control flow diagram.

Step 3: Classify all traced software functions (modules) as critical to either the proper execution of critical software functions or the quality requirements.

Step 4: Focus additional analysis on these traced critical software functions (modules).

Step 5: Repeat criticality analysis for each life-cycle process/activity to determine whether the implementation details shift the emphasis of the criticality.

Q. 3. What is traceability analysis?

Ans. Traceability analysis traces each software requirement back to the system requirements. This is done to ensure that each requirement correctly satisfies the system requirements. This analysis will also determine whether any derived software requirements are consistent with the original objectives, physical laws, and technologies described in the system document.

Q. 4. What is interface analysis?

Ans. It is a detailed examination of the interface requirements specifications. The evaluation criteria here is on the interfaces between the software and other hardware, user, and external software. Criticality analysis is continued and updated for the software. Criticality is assigned to each software requirement. When requirements are combined into functions, the combined criticality of requirements form the criticality for the aggregate function. The criticality analysis is updated periodically as requirement changes are introduced

as such changes can cause a functions criticality to increase or decrease depending on how the revised requirements impact system criticality.

Q. 5. Explain the tool support for review processes.

Ans. As tools become available to perform some of the tasks previously done by humans, the cost effectiveness of review processes increases.

For example, utilization of a compiler to detect syntax errors in code and thus alleviating this task for the reviewers.

Another example is the design and specification consistency checkers.

Q. 6. Suggest some techniques to find different types of errors.

Ans. Some of the techniques are discussed below:

i. **Algorithm Analysis:** It examines the logic and accuracy of the software requirements by translating algorithms into some language or structured format. The analysis involves rederiving equations or evaluating the suitability of specific numerical techniques. Algorithm analysis examines the correctness of the equations and numerical techniques, truncation and sounding effects, numerical precision of word storage and variables, and data typing influences.

ii. **Analytic Modeling:** It provides performance evaluation and capacity planning information on software design. It represents the program logic and processing of some kind of model and analyzes it for efficiency.

iii. **Control Flow Analysis:** It is used to show the hierarchy of main routines and their subfunctions. It checks that the proposed control flow is free of problems like unreachable or incorrect code.

iv. **Database Analysis:** It ensures that the database structure and access methods are compatible with the logical design. It is done on programs with significant data storage to ensure that common data and variable regions are used consistently between all calling routines, that data integrity is enforced and no data or variable can be accidentally overwritten by overflowing data tables, and that data typing and use are consistent throughout the program.

Q. 7. What is desk checking?

Ans. It involves the examination of the software design or code by an individual. It includes:

- Looking over the code for defects.
- Checking for correct procedure interfaces.
- Reading the comments and comparing it to external specifications and software design.

Q. 8. What are Petri-nets?

Ans. Petri-nets model system to ensure software design adequacy for catastrophic failure. The system is modeled using conditions and events represented by STDs. They can be executed to see how the software design will actually work under certain conditions.

Q. 9. What is program slicing?

Ans. **Slicing** is a program decomposition technique used to trace an output variable back through the code to identify all code statements relevant to a computation in the program.

Q. 10. What is test certification?

Ans. It ensures that the reported test results are the actual finding of the tests. Test related tools, media, and documentation are certified to ensure maintainability and repeatability of tests. This technique is also used to show that the delivered software product is identical to the software product that was subjected to V&V. It is used in critical software systems to verify that the required tests have been executed and that the delivered software product is identical to the product subjected to software V&V.

REVIEW QUESTIONS

1. **a.** Discuss briefly the V&V activities during the design phase of the software development process.

   b. Discuss the different forms of IV&V.

2. **a.** What is the importance of technical reviews in software development and maintenance life cycle?

   b. Briefly discuss how walkthroughs help in technical reviews.

3. **a.** Explain why validation is more difficult than verification.

   b. Explain validation testing.

4. Explain the following with the help of an example: "Verification and Validation."
5. Write short notes on V&V standards.
6. Write short notes on "Independent V&V contractor."
7. What is an independent test organization? Why is it necessary?
8. Discuss the role of verification and validation in software evolution. What are verification and validation objective standards?
9. What is the difference between verification and validation?
10. Discuss V&V approaches, standards, and objectives.
11. What is formal testing?
12. **a.** What is a software audit?

    b. Explain code walkthrough.
13. What are the different V&V activities performed at the coding phase of the software development process?
14. Discuss in brief software verification and validation plan.
15. **a.** Why are technical reviews conducted?

    b. Discuss a formal method for conducting technical reviews.
16. Discuss one testing technique used during the design phase of SDLC.
17. Explain all the phases through which we can approach testing using a sample software testing problem. Comment on difficulty of testing.
18. Briefly explain code walkthrough and inspection.
19. During code review you detect errors whereas during code testing you detect failures. Justify.
20. **a.** What are the testing activities performed during requirement analysis phase of SDLC.

    b. Briefly describe formal proof of correctness.
21. What is a requirements traceability matrix? Give its importance.

CHAPTER 3

SOFTWARE QUALITY

Inside this Chapter:

3.0. Introduction
3.1. Role of Process in Software Quality
3.2. Software Control
3.3. Quality Assurance
3.4. Quality Assurance Analyst
3.5. Quality Factor
3.6. Quality Management
3.7. Methods of Quality Management
3.8. Core Components of Quality
3.9. Core Aspects of Quality

3.0. INTRODUCTION

There are two ways of knowing whether good quality software has been produced or NOT:

1. Measuring the attributes of the software that has been developed (quality control).
2. Monitoring and controlling the development process of the software (quality assurance).

We can compare the process of developing software with that of preparing of a pizza. If we prepare a pizza (somehow) and then serve it, we will certainly get plenty of comments on its quality. By then it is too late. The only way is to take care the next time we make pizza. Ultimately, the consumers should be satisfied. So we define quality as *"A product which*

fulfills and continues to meet the purpose for which it was produced is a quality product."

The quality can be ensured provided that the process is ensured. Like we need a good recipe to prepare a good pizza, similarly we need good tools and methods during software development.

3.1. ROLE OF PROCESS IN SOFTWARE QUALITY

Software quality can be ensured at any of the following stages:

1. At the outset of a software development to clear the goals.
2. During development to take the development process towards the goals.
3. On completion to assess the finished piece of software.

Quality has been given many definitions:

1. Quality is fitness for use (J. Juran).
2. Quality is most profitable product line we have (H. Genee).
3. Quality is free (P. Crosby).
4. Quality is compliance to a standard (W. Perry).
5. Quality is a degree of excellence (R. Glass).
6. The degree to which a system, component, or process meets–
   a. Specified requirements, and
   b. Customer or user needs or expectations. (IEEE std. 610.12-1990)
7. The totality of features and characteristics of a product or service that bear or its ability to satisfy specified or implied needs. (ISO 9003-3-1991)

As per the report of Software Engineering Institute (SEI) in Information week, Jan. 21, 2002, "Experienced programmers inject one defect /10 LOC. If 99% of those are caught, 1000 bugs remain in a 1 million line program."

Testing shows you the lack of external quality: Correctness, efficiency, and reliability. The roots are invisible internal quality like software architecture (structure), complexity, coupling, reusability, readability, flexibility, etc.

3.2. SOFTWARE CONTROL

A quality plan describes a number of quality controls. A quality control is an activity that checks that the project's quality factors are being achieved and produces some documentary evidence. These documents are then available

to anyone (like a customer) with an interest in checking that the quality of the product is ensured. Depending on the quality factors for the project, quality controls might include:

Action	Document	Factor being checked
1. Collect a complexity metric for each component in the system	Data on complexity	Maintainability
2. Component test	Test result	Correctness
3. Walkthrough to examine component for reusability	Minutes of walkthrough	Reusability

3.3. QUALITY ASSURANCE (QA)

Quality assurance (QA) means ensuring that a software system meets its quality goals. The goals differ from one project to another. They must be clear and can be selected from that list of quality factors (given earlier). To achieve its goals, a project must use effective tools and methods. Also checks must be carried out during the development process at every available opportunity to see that the process at every available opportunity to see that the process is being carried out correctly.

To ensure that effective tools and methods are being used, an organization distills its best practices and documents in to a *quality manual*. This is like a library of all effective tools, methods, and notations. This manual describes all of the standards and procedures that are available to be used. A *standard* defines a range, limit, tolerance, or norm of some measurable attribute against which compliance can be judged. For example, during white-box testing, every source code statement must be executed at least once.

A *procedure* prescribes a way of doing something (rules, steps, guidelines, plans). For example, black-box testing, white-box testing, and a walkthrough must be used to verify each component (module) of software.

QA is more effective if it is planned in advance (along with the planning of other SDLC activities). The project manager (PM) must:

a. **Decide** which quality factors are vital for a particular project (like high reliability and maintainability).

b. **Select** standards and producers from the quality manual that are appropriate to meeting the quality goals (like the use of complexity metrics to check maintainability).

c. **Assemble** these into a quality assurance plan for the project. This describes what the procedures and standards are, when they will be done, and who does then.

3.4. QA ANALYST

A QA analyst is responsible for applying the principles and practices of SQA throughout SDLC. Main functions are:

a. To create end-to-end test plans.
b. To execute the plan.
c. To manage all activities in the plan.

All objectives must be met and the solution should be in terms of functionality performance, reliability, stability, and compatibility with other internal and external systems.

The analyst does this by ensuring that every phase and feature of software solution is tested. He or she may even do back-end (DBMS-based) testing.

3.5. QUALITY FACTOR(S)

We now present a list of desirable quality factors:

a. **Correctness:** The extent to which the software meets its specifications and meet its users requirements.
b. **Reliability:** The degree to which the software continues to work without failing.
c. **Performance:** The amount of main memory and processor time that the software uses.
d. **Integrity:** The degree to which the software enforces control over access to information by users.
e. **Usability:** The ease of use of the software.
f. **Maintainability:** The effort required to find and fix a fault.
g. **Flexibility:** The effort required to change the software to meet changed requirements.
h. **Testability:** The effort required to test the software effectively.

i. **Portability:** The effort required to transfer the software to a different hardware and/or software platform.

j. **Reusability:** The extent to which the software or a component within it can be reused within some other software.

k. **Interoperability:** The effort required to make the software work in conjunction with some other software.

l. **Security:** The extent to which the software is safe from external sabotage that may damage it and impair its use.

Please note that some of these qualities can be mutually contradictory such as if high performance is required then probability suffers. Also note that not every attribute is desired in every software. Instead for each project it is important to identify the salient factors before development begins.

3.6. QUALITY MANAGEMENT

Present day competitive environments demand a better quality product or service at lower rates. Please understand that only those organizations which manage productivity and quality on a continuous basis are in a position to compete in an global market place.

As per ISO 8402: Quality Vocabulary – "The totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs" is quality.

Management is the creation and the continuous improvement of an organizational system that when used by organizational members lead to increased value for the customers of its products or services.

Incremental competition, rapidly changing technology, and demanding customers are the primary forces operating on managers today.

Managers Must

a. Keep the organization in touch with the external environment.

b. Guide performance.

Management Operates at Three Levels

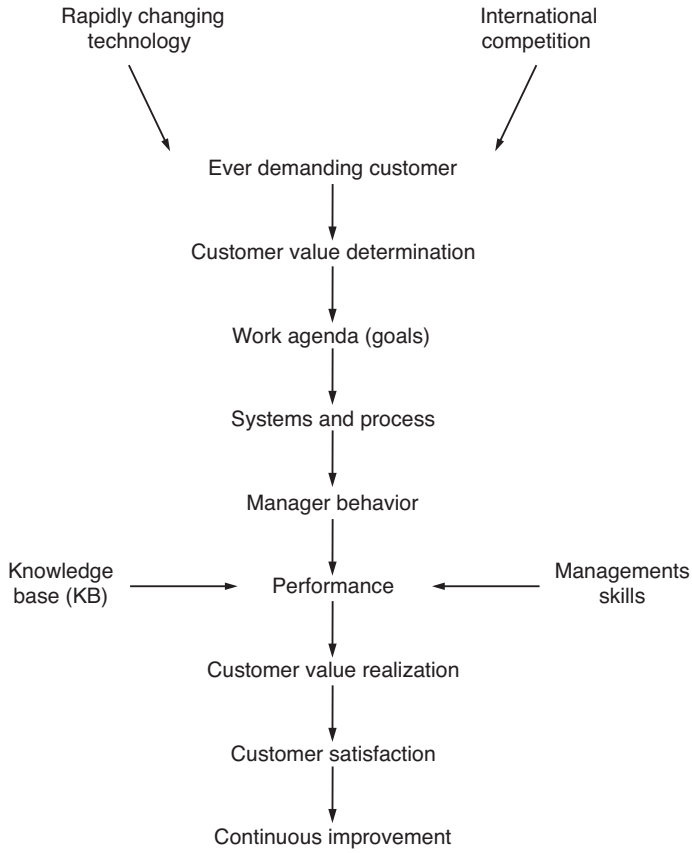
1. **Strategic level:** It is concerned with the direction of the organization relative to the environment.
2. **Tactical level:** It relates to the implementation of the strategic plan.
3. **Operational level:** It deals with the near-term achievement of goals.

NOTE *Another way to view management's responsibilities is through the organizational levels of top, middle, and first line.*

In the last ten years, managers have increasingly recognized that systems and processes management is also their task. Mangers also have responsibility for the ethical conduct of their organizations.

3.7. METHODS OF QUALITY MANAGEMENT

This is a model of TQ in managerial work. To show the correct path of responsibility to managers, the following model is used.



3.8. CORE COMPONENTS OF QUALITY

The four major features that symbolize quality are:

Four features: Totality, Improvement, Foundation, and Documentation are represented in the form of a diamond as shown in Figure 3.1.

1. **Totality:** Means that this effort is all encompassing. This implies that area and functions, all activities, all employees are striving for optimum quality all the time. Then the quality target is 100%.
2. **Documentation:** It is an integrated people-machine-information (PMI) relations that make this TQM effort happen. Please understand that it is the proper dissemination of this PMI relationship to all people that constitutes good documentation. Also note that this dissemination of information means that each person can visualize his or her own work assignments and responsibilities in a quality activity. It is this self-visualization that tends to a total quality effort (TQE).
3. **Improvement:** The ability to bring about an order-of-magnitude improvements in all quality activities to the company is the cornerstone of any TQM activity. This has led to CPI activity under the TQM umbrella where CPI stands for *constant process improvement* and is defined as a systematic approach taken by all employees to achieve the highest levels of quality.
4. **Foundation:** Foundation should be solid. This comes from the company's organizational structure and systems. A well-organized company means properly managed activities.

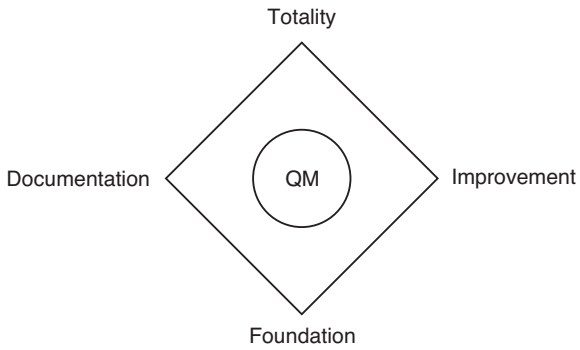
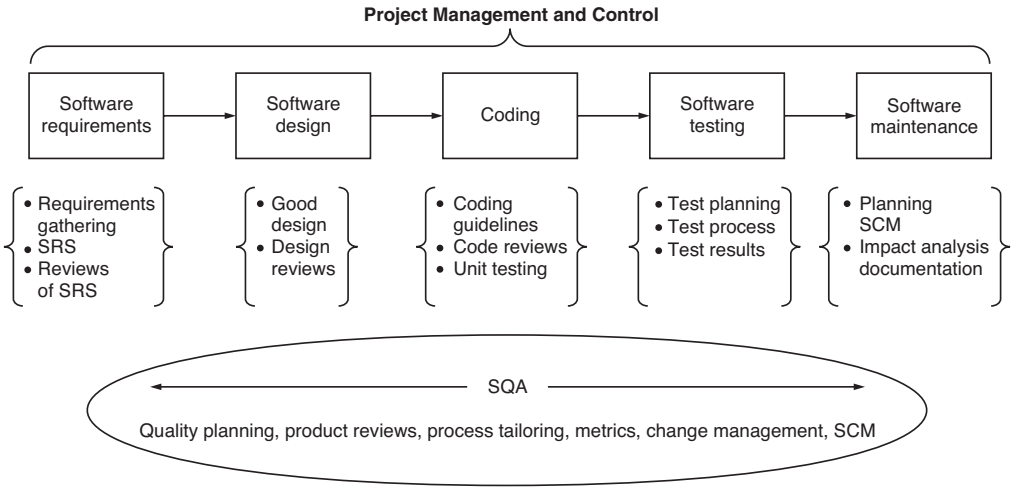


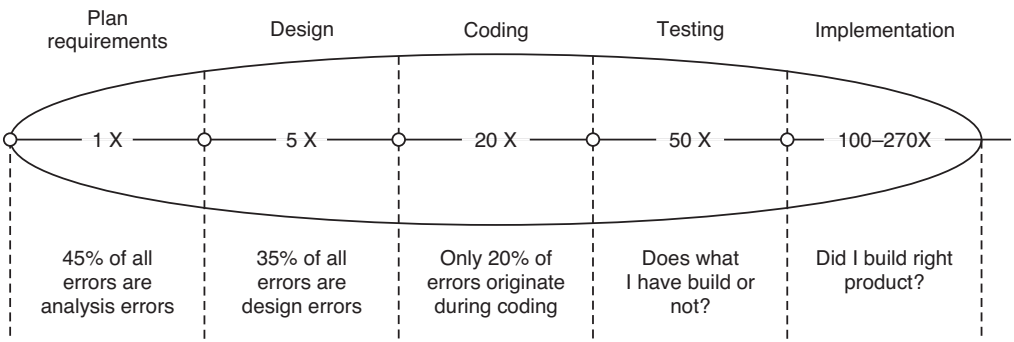
FIGURE 3.1 Diamond of Quality Components.

3.9. CORE ASPECTS OF QUALITY

The quality assurance begins early during SDLC as follows:



As per B. Boehm, in his book SEE,



SUMMARY

In this chapter, we have studied what is software quality, how it can be improved, controlled, and ensured. We have also studied quality management, the core components of quality, and its core aspects.

MULTIPLE CHOICE QUESTIONS

1. Reliability (failure-free operation) of software is a:
   - **a.** Quality measure
   - **b.** Design feature
   - **c.** Testing techniques
   - **d.** None of the above.
2. Quality refers to:
   - **a.** Conformance to standards
   - **b.** Design approach
   - **c.** Coding approach
   - **d.** None of the above.
3. Which one of the following symbolizes quality?
   - **a.** Improvement
   - **b.** Totality
   - **c.** Foundation
   - **d.** All of the above.
4. PMI stands for:
   - **a.** Project Management Interface
   - **b.** People Management Interest
   - **c.** People Machine Information
   - **d.** None of the above.
5. CPI stands for:
   - **a.** Constant process improvement
   - **b.** Common process improvement
   - **c.** Constant process improvement
   - **d.** None of the above.
6. CM and SCM are put under:
   - **a.** SQA
   - **b.** SPM
   - **c.** Testing
   - **d.** None of the above.
7. Who is responsible for quality training?
   - **a.** Quality manager
   - **b.** Bank manger
   - **c.** System manager
   - **d.** None of the above.

8. An approach that attempts to ensure that all processes are performed without error is:

a. Error-free **b.** Zero-defect

c. 0/1-defect **d.** None of the above.

9. Quality is often related to:

a. Freedom **b.** Philosophy

c. Mathematics **d.** None of the above.

10. All costs incurred in maintaining quality of software and performing quality-related activities are put under:

a. Quality factors **b.** Quality control

c. Quality costs **d.** None of the above.

ANSWERS

1. a.	**2.** a.	**3.** d.	**4.** c.
5. c.	**6.** a.	**7.** a.	**8.** b.
9. a.	**10.** c.		

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

Q. 1. Software is to be developed to control a fly-by-wire airplane. What are likely to be the important factors?

Ans. Correctness and reliability

Q. 2. List some other quality factors that can be quantified.

Ans. Cost, size

Q. 3. What quality factors would the measurement of fault density help achieve?

Ans. Correctness and reliability

Q. 4. Suggest a formula for computing complexity of a program, other than those given by McCabe.

Ans. The suggested formula is:

Complexity = No. of decision + (No. of data references – No. of statements)

In this formula, if each statement refers to one data item only, the second term is zero.

Q. 5. Define: **a.** Fault density
b. MTTF

Ans. **a.** It is defined as the estimated number of faults per 1000 LOC.
b. MTTF or mean-time-to-failure is defined as the average time for a system to perform without failing.

This can be measured simply by logging the times at which failures occur and simply calculating the average time between successive failures. Then, this gives a predication for the future failure rate of the system.

Q. 6. How will you measure correctness and reliability?

Ans. Correctness can be measured using fault density as a metric.
Reliability can be measured using MTTF as a metric.

Q. 7. What will be the cyclomatic complexity of a program that consists of only a sequence of statements?

Ans. Such a program will have V(G) = 1, however long it is. Thus, the smallest value of this metric is 1.

Q. 8. Give at least 10 reasons as to why we must believe in quality.

Ans. This is because:

1. Quality is profitable.
2. Poor quality is penalized.
3. Quality distinguishes.
4. Quality is expected.
5. Quality renews.
6. Quality unites and inspires.
7. Quality delivers.
8. Quality has pride.
9. Quality has self-interest.
10. Quality is accelerated improvement.

Q. 9. What are the major benefits of good quality of software?

Ans. Some of the benefits of good quality of software are:

1. More efficient process.
2. Higher productivity.
3. High employee's morale.
4. Less fire fighting and more interest in creativity.
5. Lower costs.
6. More customer satisfaction.
7. Higher profits.

Q. 10. What is a zero-defect approach?

Ans. Zero defects (ZD) is an approach that attempts to ensure that all processes are performed without error. This is done by providing trainings and motivations to all employees.

Q. 11. In what situations will quality factors be used?

Ans. The situations are as follows:

a. At the outset of a software development (to clarify the goals).

b. During development to guide development process towards the goals.

c. On completion to assess the conflicted piece of software.

CHAPTER 4

BLACK-BOX (OR FUNCTIONAL) TESTING TECHNIQUES

Inside this Chapter:

4.0. Introduction to Black-Box (or Functional Testing)
4.1. Boundary Value Analysis (BVA)
4.2. Equivalence Class Testing
4.3. Decision Table Based Testing
4.4. Cause-Effect Graphing Technique
4.5. Comparison on Black-Box (or Functional) Testing Techniques
4.6. Kiviat Charts

4.0. INTRODUCTION TO BLACK-BOX (OR FUNCTIONAL TESTING)

The term *Black-Box* refers to the software which is treated as a black-box. By treating it as a black-box, we mean that the system or source code is not checked at all. It is done from the customer's viewpoint. The test engineer engaged in black-box testing only knows the set of inputs and expected outputs and is unaware of how those inputs are transformed into outputs by the software. We will now discuss various techniques of performing black-box testing.

4.1. BOUNDARY VALUE ANALYSIS (BVA)

It is a black-box testing technique that believes and extends the concept that the density of defect is more towards the boundaries. This is done for the following reasons:

i. Programmers usually are not able to decide whether they have to use <= operator or < operator when trying to make comparisons.

ii. Different terminating conditions of for-loops, while loops, and repeat loops may cause defects to move around the boundary conditions.

iii. The requirements themselves may not be clearly understood, especially around the boundaries, thus causing even the correctly coded program to not perform the correct way.

Strongly typed languages such as Ada and Pascal permit explicit definition of variable ranges. Other languages such as COBOL, FORTRAN, and C are not strongly typed, so boundary value testing is more appropriate for programs coded in such languages.

4.1.1. WHAT IS BVA?

The basic idea of BVA is to use input variable values at their minimum, just above the minimum, a nominal value, just below their maximum, and at their maximum. That is, {min, min+, nom, max–, max}. This is shown in Figure 4.1.

BVA is based upon a critical assumption that is known as single fault assumption theory. According to this assumption, we derive the test cases on the basis of the fact that failures are not due to a simultaneous occurrence

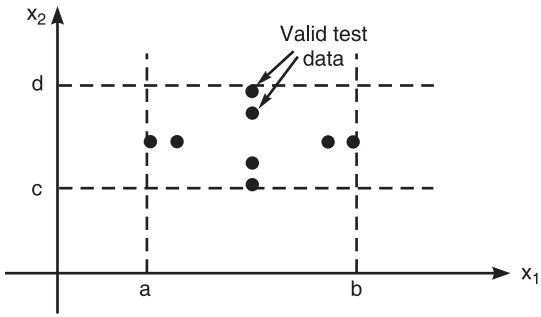


FIGURE 4.1 BVA Test Cases.

of two (or more) faults. So we derive test cases by holding the values of all but one variable at their nominal values and letting that variable assume its extreme values.

If we have a function of n-variables, we hold all but one at the nominal values and let the remaining variable assume the min, min+, nom, max–, and max values, repeating this for each variable. Thus, for a function of n variables, BVA yields (4n + 1) test cases.

4.1.2. LIMITATIONS OF BVA

1. Boolean and logical variables present a problem for boundary value analysis.
2. BVA assumes the variables to be truly independent which is not always possible.
3. BVA test cases have been found to be rudimentary because they are obtained with very little insight and imagination.

4.1.3. ROBUSTNESS TESTING

Another variant to BVA is robustness testing. In *BVA*, we are within the legitimate boundary of our range. That is, we consider the following values for testing:

{min, min+, nom, max–, max} whereas in *robustness testing*, we try to cross these legitimate boundaries also. So, now we consider these values for testing:

{min–, min, min+, nom, max–, max, max+}

Again, with robustness testing, we can focus on exception handling. With strongly typed languages, robustness testing may be very awkward. For example, in PASCAL, if a variable is defined to be within a certain range, values outside that range result in run-time errors that abort normal execution.

For a program with n-variables, robustness testing will yield (6n + 1) test-cases. So, we can draw a graph now. (Figure 4.2)

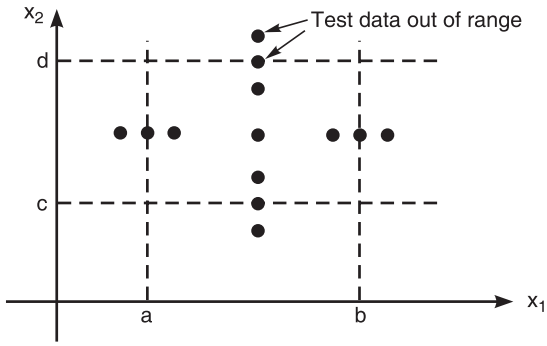


FIGURE 4.2 Robustness Test Cases.

Each dot represents a test value at which the program is to be tested. In robustness testing, we cross the legitimate boundaries of input domain. In the graph of Figure 4.2, we show this by dots that are outside the range [a, b] of variable x_1. Similarly, for variable x_2, we have crossed its legitimate boundary of [c, d] of variable x_2.

This type of testing is quite common in electric and electronic circuits.

Furthermore, this type of testing also works on single fault assumption theory.

4.1.4. WORST-CASE TESTING

If we reject our basic assumption of single fault assumption theory and focus on what happens when we reject this theory—it simply means that we want to see what happens *when more than one variable has an extreme value*. This is multiple path assumption theory. In electronic circuit analysis, this is called as *"worst-case analysis."* We use this idea here to generate worst-case test cases.

For each variable, we start with the five- element set that contains the min, min+, nom, max–, and max values. We then take the *Cartesian product* of these sets to generate test cases. This is shown in Figure 4.3.

For a program with n-variables, 5^n test cases are generated.

NOTE *Robust worst-case testing yields 7^n test cases.*

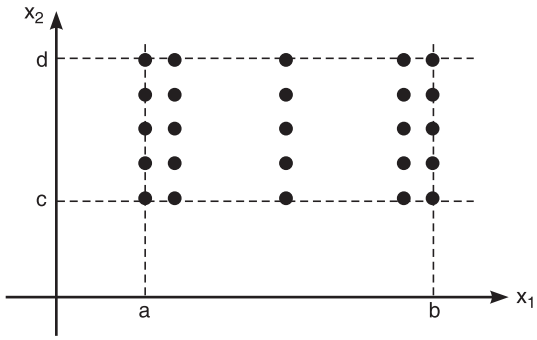


FIGURE 4.3 Worst-Case Test Cases.

4.1.5. EXAMPLES WITH THEIR PROBLEM DOMAIN

4.1.5.1. TEST CASES FOR THE TRIANGLE PROBLEM

Before we generate the test cases, first we need to give the problem domain:

Problem Domain: "The triangle program accepts three integers, a, b, and c, as input. These are taken to be the sides of a triangle. The integers a, b, and c must satisfy the following conditions:

C_1: $1 \le a \le 200$ $\quad$ C_4: $a < b + c$
C_2: $1 \le b \le 200$ $\quad$ C_5: $b < a + c$
C_3: $1 \le c \le 200$ $\quad$ C_6: $c < a + b$

The output of the program may be: Equilateral, Isosceles, Scalene, or "NOT-A-TRIANGLE."

How to Generate BVA Test Cases?

We know that our range is [1, 200] where 1 is the lower bound and 200 is the upper bound. Also, we find that this program has three inputs—a, b, and c. So, for our case

$$n = 3$$

$\because$ BVA yields $(4n + 1)$ test cases, so we can say that the total number of test cases will be $(4 \times 3 + 1) = 12 + 1 = 13$.

Table 4.1 shows those 13 test cases.

TABLE 4.1 BVA test cases for triangle problem.

Case ID	a	b	c	Expected output
1.	100	100	1	Isosceles
2.	100	100	2	Isosceles
3.	100	100	100	Equilateral
4.	100	100	199	Isosceles
5.	100	100	200	Not a triangle
6.	100	1	100	Isosceles
7.	100	2	100	Isosceles
8.	100	100	100	Equilateral
9.	100	199	100	Isosceles
10.	100	200	100	Not a triangle
11.	1	100	100	Isosceles
12.	2	100	100	Isosceles
13.	100	100	100	Equilateral
14.	199	100	100	Isosceles
15.	200	100	100	Not a triangle

Please note that we explained above that we can have 13 test cases (4n + 1) for this problem. But instead of 13, now we have 15 test cases. Also, test case ID number 8 and 13 are redundant. So, we ignore them. However, we do not ignore test case ID number 3 as we must consider at least one test case out of these three. Obviously, it is mechanical work!

We can say that these 13 test cases are sufficient to test this program using BVA technique.

Question for Practice

1. Applying the robustness testing technique, how would you generate the test cases for the triangle problem given above?

4.1.5.2. TEST CASES FOR NEXT DATE FUNCTION

Before we generate the test cases for the Next Date function, we must know the problem domain of this program:

Problem Domain Next Date is a function of three variables: month, date, and year. It returns the date of next day as output. It reads current date as input date. The conditions are:

C_1: $1 \leq \text{month} \leq 12$
C_2: $1 \leq \text{day} \leq 31$
C_3: $1900 \leq \text{year} \leq 2025$

If any of conditions C_1, C_2, or C_3 fails, then this function produces an output "value of month not in the range 1...12."

Because many combinations of dates exist this function just displays one message: "Invalid Input Date."

Complexities in Next Date Function

A very common and popular problem occurs if the year is a leap year. We have taken into consideration that there are 31 days in a month. But what happens if a month has 30 days or even 29 or 28 days? A year is called as a leap year if it is divisible by 4, unless it is a century year. Century years are leap years only if they are multiples of 400. So, 1992, 1996, and 2000 are leap years while 1900 is not a leap year.

Furthermore, in this Next Date problem, we find examples of *Zipf's law* also, which states that "80% of the activity occurs in 20% of the space." Here also, much of the source-code of the Next Date function is devoted to the leap year considerations.

How to Generate BVA Test Cases for This Problem?

The Next Date program takes date as input and checks it for validity. If valid, it returns the next date as its output.

As we know, with single fault assumption theory, $(4n + 1)$ test cases can be designed. Here, also $n = 3$. So, the total number of test cases are $(4 \times 3 + 1) = 12 + 1 = 13$.

The boundary value test cases are

Case ID	Month (mm)	Day (dd)	Year (yyyy)	Expected Output
1.	6	15	1900	16 June, 1900
2.	6	15	1901	16 June, 1901
3.	6	15	1962	16 June, 1962
4.	6	15	2024	16 June, 2024
5.	6	15	2025	16 June, 2025
6.	6	1	1962	2 June, 1962
7.	6	2	1962	1 June, 1962
8.	6	30	1962	1 July, 1962
9.	6	31	1962	Invalid date as June has 30 days.
10.	1	15	1962	16 January, 1962
11.	2	15	1962	16 February, 1962
12.	11	15	1962	16 November, 1962
13.	12	15	1962	16 December, 1962

So, we have applied BVA on our Next Date problem.

Question for Practice

1. Applying robustness testing, how would you generate the test cases for the Next Date function given above.

4.1.5.3. Test cases for the commission problem

Before we generate the test cases, we must formulate the problem statement or domain for commission problem.

Problem Domain: A rifle salesperson sold rifle locks, stocks, and barrels that were made by a gunsmith. Locks cost $45, stocks cost $30, and barrels cost $25. This salesperson had to sell at least one complete rifle per month, and the production limits were such that the most the salesperson could sell in a month was 70 locks, 80 stocks, and 90 barrels. The salesperson used to send the details of sold items to the gunsmith. The gunsmith then computed the salesperson's commission as follows:

a. 10% on sales up to and including $1000

b. 15% of the next $800

c. 20% on any sales in excess of $1800

The commission program produced a monthly sales report that gave the total number of locks, stocks, and barrels sold, the salesperson's total dollar sales and finally, the commission.

How to Generate BVA Test Cases for this Problem?

We have 3 inputs in this program. So, we need (4n + 1) = 4 * 3 + 1 = 12 + 1 = 13 test cases to test this program.

The boundary value test cases are listed below in Table. We can also find that the monthly sales are limited as follows:

1 ≤ locks ≤ 70
1 ≤ stocks ≤ 80
1 ≤ barrels ≤ 90

Case ID	Locks	Stocks	Barrels	Sales
1.	35	40	1	2800
2.	35	40	2	2825
3.	35	40	45	3900
4.	35	40	89	5000
5.	35	40	90	5025
6.	35	1	45	2730
7.	35	2	45	2760
8.	35	40	45	3900
9.	35	79	45	5070
10.	35	80	45	5100
11.	1	40	45	2370
12.	2	40	45	2415
13.	35	40	45	3900
14.	69	40	45	5430
15.	70	40	45	5475

Out of these 15 test cases, 2 are redundant. So, 13 test cases are sufficient to test this program.

4.1.6. GUIDELINES FOR BVA

1. The normal versus robust values and the single-fault versus the multiple-fault assumption theory result in better testing. These methods can be applied to both input and output domain of any program.
2. Robustness testing is a good choice for testing internal variables.
3. Keep in mind that you can create extreme boundary results from non-extreme input values.

4.2. EQUIVALENCE CLASS TESTING

The use of equivalence classes as the basis for functional testing has two motivations:

a. We want exhaustive testing
b. We want to avoid redundancy

This is not handled by the BVA technique as we can see massive redundancy in the tables of test cases.

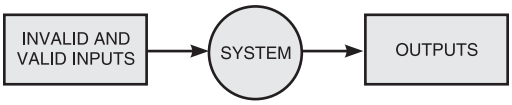


FIGURE 4.4 Equivalence Class Partitioning.

In this technique, the input and the output domain is divided into a finite number of equivalence classes. Then, we select one representative of each class and test our program against it. It is assumed by the tester that if one representative from a class is able to detect error then why should he consider other cases. Furthermore, if this single representative test case did not detect any error then we assume that no other test case of this class can detect error. In this method, we consider both valid and invalid input domains. The system is still treated as a black-box meaning that we are not bothered about its internal logic.

The idea of equivalence class testing is to identify test cases by using one element from each equivalence class. If the equivalence classes are chosen wisely, the potential redundancy among test cases can be reduced.

For example, in our triangle problem, we would certainly have a test case for an equilateral triangle and we might pick the triple (10, 10, 10) as inputs for a test case. If this is so then it is obvious that there is no sense in testing for inputs like (8, 8, 8) and (100, 100, 100). Our intuition tells us that these would be "treated the same" as the first test case. Thus, they would be redundant. The key and the craftsmanship lies in the choice of the equivalence relation that determines the classes.

Four types of equivalence class testing are discussed below:

1. Weak normal equivalence class testing
2. Strong normal equivalence class testing
3. Weak robust equivalence class testing
4. Strong robust equivalence class testing

We will discuss these one by one.

4.2.1. WEAK NORMAL EQUIVALENCE CLASS TESTING

The word "weak" means single fault assumption. This type of testing is accomplished by using one variable from each equivalence class in a test case. We would, thus, end up with the weak equivalence class test cases as shown in Figure 4.5.

Each dot in Figure 4.5 indicates a test data. From each class we have one dot meaning that there is one representative element of each test case.

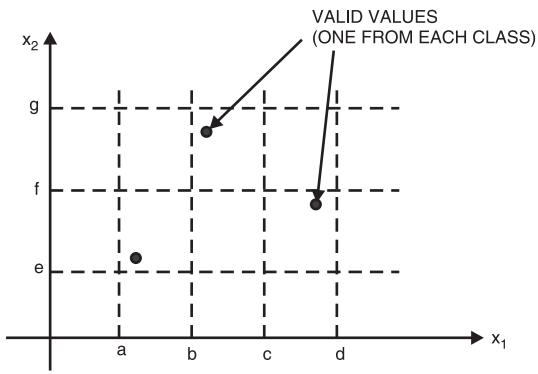


FIGURE 4.5 Weak Normal Equivalence Class Test Cases.

In fact, we will have, always, the same number of weak equivalence class test cases as the classes in the partition.

4.2.2. STRONG NORMAL EQUIVALENCE CLASS TESTING

This type of testing is based on the multiple fault assumption theory. So, now we need test cases from each element of the Cartesian product of the equivalence classes, as shown in Figure 4.6.

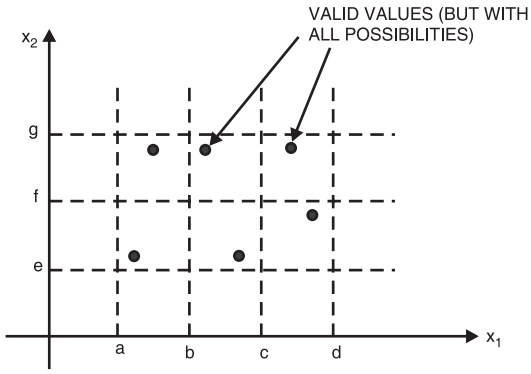


FIGURE 4.6 Strong Normal Equivalence Class Test Cases.

Just like we have truth tables in digital logic, we have similarities between these truth tables and our pattern of test cases. The Cartesian product guarantees that we have a notion of "completeness" in two ways:

a. We cover all equivalence classes.

b. We have one of each possible combination of inputs.

4.2.3. WEAK ROBUST EQUIVALENCE CLASS TESTING

The name for this form of testing is counter intuitive and oxymoronic. The word "weak" means single fault assumption theory and the word "robust" refers to invalid values. The test cases resulting from this strategy are shown in Figure 4.7.

Two problems occur with robust equivalence testing. They are listed below:

1. Very often the specification does not define what the expected output for an invalid test case should be. Thus, testers spend a lot of time defining expected outputs for these cases.

2. Also, strongly typed languages like Pascal and Ada, eliminate the need for the consideration of invalid inputs. Traditional equivalence testing is a product of the time when languages such as FORTRAN, C, and COBOL were dominant. Thus, this type of error was common.

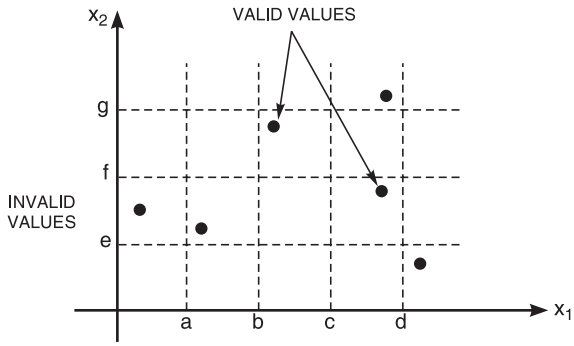


FIGURE 4.7 Weak Robust Equivalence Class Test Cases.

4.2.4. STRONG ROBUST EQUIVALENCE CLASS TESTING

This form of equivalence class testing is neither counter intuitive nor oxymoronic but is redundant. As explained earlier "robust" means consideration of invalid values and "strong" means multiple fault assumption. We obtain the test cases from each element of the Cartesian product of all the equivalence classes. This is shown in Figure 4.8.

We find here that we have 8 robust (invalid) test cases and 12 strong or valid inputs. Each is represented with a dot. So, totally we have 20 test cases (represented as 20 dots) using this technique.

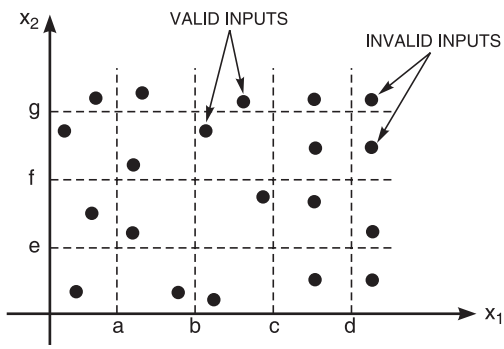


FIGURE 4.8 Strong Robust Equivalence Class Test Cases.

4.2.5. Solved Examples

4.2.5.1. *Equivalence class test cases for the triangle problem*

As stated in the problem definition earlier, we note that in our triangle problem four possible outputs can occur:

a. NOT-A-TRIANGLE

b. Scalene

c. Isosceles

d. Equilateral

We can use these to identify output (range) equivalence classes as follows:

01 = {<a, b, c>: the triangle is equilateral}
02 = {<a, b, c>: the triangle is isosceles}
03 = {<a, b, c>: the triangle is scalene}
04 = {<a, b, c>: sides a, b, and c do not form a triangle}

Now, we apply these four techniques of equivalence class partitioning one by one to this problem.

a. The four *weak normal equivalence class* test cases are:

Case ID	a	b	c	Expected output
WN_1	5	5	5	Equilateral
WN_2	2	2	3	Isosceles
WN_3	3	4	5	Scalene
WN_4	4	1	2	Not a triangle

b. Because no valid subintervals of variables a, b, and c exist, so the strong normal equivalence class test cases are identical to the weak normal equivalence class test cases.

c. Considering the invalid values for a, b, and c yields the following additional *weak robust equivalence class* test cases:

Case ID	a	b	c	Expected output
WR_1	–1	5	5	Invalid value of a
WR_2	5	–1	5	Invalid value of b
WR_3	5	5	–1	Invalid value of c
WR_4	201	5	5	Out of range value of a
WR_5	5	201	5	Out of range value of b
WR_6	5	5	201	Out of range value of c

d. While *strong robust equivalance class* test cases are:

Case ID	a	b	c	Expected output
SR_1	–1	5	5	Invalid value of a
SR_2	5	–1	5	Invalid value of b
SR_3	5	5	–1	Invalid value of c
SR_4	–1	–1	5	Invalid values of a and b
SR_5	5	–1	–1	Invalid values of b and c
SR_6	–1	5	–1	Invalid values of a and c
SR_7	–1	–1	–1	Invalid values of a, b, and c

Please note that the expected outputs describe the invalid input values thoroughly.

4.2.5.2. EQUIVALENCE CLASS TEST CASES FOR NEXT DATE FUNCTION

The actual craft of choosing the test cases lies in this example. Recall that Next Date is a function of three variables— month (mm), day (dd), and year (yyyy). Assume that their ranges are

$$1 \le \text{month} \le 12$$
$$1 \le \text{day} \le 31$$
$$1812 \le \text{year} \le 2012$$

So, based on valid values, the equivalence classes are:

$$M_1 = \{\text{month: } 1 \le \text{month} \le 12\}$$
$$D_1 = \{\text{day: } 1 \le \text{day} \le 31\}$$
$$Y_1 = \{\text{year: } 1812 \le \text{year} \le 2012\}$$

And the invalid equivalence classes are:

M_2 = {month: month < 1}
M_3 = {month: month > 12}
D_2 = {day: day < 1}
D_3 = {day: day > 31}
Y_2 = {year: year < 1812}
Y_3 = {year: year > 2012}

a. & **b.** Because the number of valid classes equals the number of independent variables, only one weak normal equivalence class test case occurs and it is identical to the strong normal equivalence class test case:

Case ID	Month	Day	Year	Expected output
WN_1, SN_1	6	15	1912	6/16/1912

So, we get this test case on the basis of valid classes – M_1, D_1, and Y_1 above.

c. Weak robust test cases are given below:

Case ID	Month	Day	Year	Expected output
WR_1	6	15	1912	6/16/1912
WR_2	–1	15	1912	Invalid value of month as month cannot be –ve.
WR_3	13	15	1912	Invalid value of month as month <12, always.
WR_4	6	–1	1912	Invalid value of day as day cannot be –ve.
WR_5	6	32	1912	Invalid value of day as we cannot have 32 days in any month.
WR_6	6	15	1811	Invalid value of year as its range is 1812 to 2012 only.
WR_7	6	15	2013	Invalid value of year.

So, we get 7 test cases based on the valid and invalid classes of the input domain.

d. Strong robust equivalence class test cases are given below:

Case ID	Month	Day	Year	Expected output
SR_1	–1	15	1912	Invalid value of month as month cannot be –ve.
SR_2	6	–1	1912	Invalid value of day as day is –ve.
SR_3	6	15	1811	Invalid value of year.
SR_4	–1	–1	1912	Invalid month and day.
SR_5	6	–1	1811	Invalid day and year.
SR_6	–1	15	1811	Invalid month and year.
SR_7	–1	–1	1811	Invalid month, day, and year.

Modified Equivalence Class for this Problem

We need the modified classes as we know that at the end of a month, the next day is 1 and the month is incremented. At the end of a year, both the day and the month are reset to 1 and the year is also incremented. Finally, the problem of leap year makes determining the last day of a month interesting. With all this in mind, we postulate the following equivalence classes:

M_1 = {month: month has 30 days}
M_2 = {month: month has 31 days}
M_3 = {month: month is February}
D_1 = {day: $1 \leq$ day ≤ 28}
D_2 = {day: day = 29}
D_3 = {day: day = 30}
D_4 = {day: day = 31}
Y_1 = {year: year = 2000}
Y_2 = {year: year is a leap year}
Y_3 = {year: year is a common year}

So, now what will be the weak equivalence class test cases?

As done earlier, the inputs are mechanically selected from the approximate middle of the corresponding class:

Case ID	Month	Day	Year	Expected output
WN_1	6	14	2000	6/15/2000
WN_2	7	29	1996	7/30/1996
WN_3	2	30	2002	2/31/2002 (Impossible)
WN_4	6	31	2000	7/1/2000 (Impossible)

This *mechanical selection* of input values makes no consideration of our domain knowledge and thus, we have two impossible dates. This will always be a problem with "automatic" test case generation because all of our domain knowledge is not captured in the choice of equivalence classes.

The *strong normal equivalence class test* cases for the revised classes are:

Case ID	Month	Day	Year	Expected output
SN_1	6	14	2000	6/15/2000
SN_2	6	14	1996	6/15/1996
SN_3	6	14	2002	6/15/2002
SN_4	6	29	2000	6/30/2000
SN_5	6	29	1996	6/30/1996
SN_6	6	29	2002	6/30/2002
SN_7	6	30	2000	6/31/2000 (Impossible date)
SN_8	6	30	1996	6/31/1996 (Impossible date)
SN_9	6	30	2002	6/31/2002 (Impossible date)
SN_{10}	6	31	2000	7/1/2000 (Invalid input)
SN_{11}	6	31	1996	7/1/1996 (Invalid input)
SN_{12}	6	31	2002	7/1/2002 (Invalid input)
SN_{13}	7	14	2000	7/15/2000
SN_{14}	7	14	1996	7/15/1996
SN_{15}	7	14	2002	7/15/2002

Case ID	Month	Day	Year	Expected output
SN_{16}	7	29	2000	7/30/2000
SN_{17}	7	29	1990	7/30/1990
SN_{18}	7	29	2002	7/30/2002
SN_{19}	7	30	2000	7/31/2000
SN_{20}	7	30	1996	7/31/1996
SN_{21}	7	30	2002	7/31/2002
SN_{22}	7	31	2000	8/1/2000
SN_{23}	7	31	1996	8/1/1996
SN_{24}	7	31	2002	8/1/2002
SN_{25}	2	14	2000	2/15/2000
SN_{26}	2	14	1996	2/15/1996
SN_{27}	2	14	2002	2/15/2002
SN_{28}	2	29	2000	3/1/2000 (Invalid input)
SN_{29}	2	29	1996	3/1/1996
SN_{30}	2	29	2002	3/1/2002 (Impossible date)
SN_{31}	2	30	2000	3/1/2000 (Impossible date)
SN_{32}	2	30	1996	3/1/1996 (Impossible date)
SN_{33}	2	30	2002	3/1/2002 (Impossible date)
SN_{34}	6	31	2000	7/1/2000 (Impossible date)
SN_{35}	6	31	1996	7/1/1996 (Impossible date)
SN_{36}	6	31	2002	7/1/2002 (Impossible date)

So, three month classes, four day classes, and three year classes results in $3 \times 4 \times 3 = 36$ strong normal equivalence class test cases. Furthermore, adding two invalid classes for each variable will result in 150 strong robust equivalence class test cases.

It is difficult to show these 150 classes here.

4.2.5.3. EQUIVALENCE CLASS TEST CASES FOR THE COMMISSION PROBLEM

The valid classes of the input variables are as follows:

L_1 = {locks: 1 ≤ locks ≤ 70}
L_2 = {locks = –1}
S_1 = {stocks: 1 ≤ stocks ≤ 80}
B_1 = {barrels: 1 ≤ barrels ≤ 90}

The corresponding invalid classes of the input variables are as follows:

L_3 = {locks: locks = 0 or locks < –1}
L_4 = {locks: locks > 70}
S_2 = {stocks: stocks < 1}
S_3 = {stocks: stocks > 80}
B_2 = {barrels: barrels < 1}
B_3 = {barrels: barrels > 90}

a. & **b.** As the number of valid classes is equal to the number of independent variables, so we have exactly one weak normal equivalence class test case and again, it is identical to the strong normal equivalence class test case. It is given in the following table.

Case ID	Locks	Stocks	Barrels	Expected output
WN_1, SN_1	35	40	45	3900

c. Also, we have seven weak robust test cases as given below:

Case ID	Locks	Stocks	Barrels	Expected output
WR_1	35	40	45	3900
WR_2	0	40	45	Invalid input
WR_3	71	40	45	Invalid input
WR_4	35	0	45	Invalid input
WR_5	35	81	45	Invalid input
WR_6	35	40	0	Invalid input
WR_7	35	40	91	Invalid input

d. And finally, the strong robust equivalence class test cases are as follows:

Case ID	Locks	Stocks	Barrels	Expected output
SR_1	–1	40	45	Value of locks not in the range 1–70.
SR_2	35	–1	45	Value of stocks not in the range 1–80.
SR_3	35	40	–1	Value of barrels not in the range 1–90.
SR_4	–1	–1	45	Values of locks and stocks are not in their ranges.
SR_5	–1	40	–1	Values of locks and barrels are not in their ranges.
SR_6	35	–1	–1	Values of stocks and barrels are not in their ranges.
SR_7	–1	–1	–1	Values of locks, stocks, and barrels are not in their ranges.

4.2.6. GUIDELINES FOR EQUIVALENCE CLASS TESTING

The following are guidelines for equivalence class testing:

1. The weak forms of equivalence class testing (normal or robust) are not as comprehensive as the corresponding strong forms.
2. If the implementation language is strongly typed and invalid values cause run-time errors then it makes no sense to use the robust form.
3. If error conditions are a high priority, the robust forms are appropriate.
4. Equivalence class testing is approximate when input data is defined in terms of intervals and sets of discrete values. This is certainly the case when system malfunctions can occur for out-of-limit variable values.
5. Equivalence class testing is strengthened by a hybrid approach with boundary value testing (BVA).
6. Equivalence class testing is used when the program function is complex. In such cases, the complexity of the function can help identify useful equivalence classes, as in the next date problem.
7. Strong equivalence class testing makes a presumption that the variables are independent and the corresponding multiplication of test cases raises issues of redundancy. If any dependencies occur, they will often generate "error" test cases, as shown in the next date function.

8. Several tries may be needed before the "right" equivalence relation is established.
9. The difference between the strong and weak forms of equivalence class testing is helpful in the distinction between progression and regression testing.

4.3. DECISION TABLE BASED TESTING

Of all the functional testing methods, those based on decision tables are the most rigorous because decision tables enforce logical rigor.

4.3.1. WHAT ARE DECISION TABLES?

Decision tables are a precise and compact way to model complicated logic. They are ideal for describing situations in which a number of combinations of actions are taken under varying sets of conditions.

Rules →

Condition Stub	Condition Entry
Action Stub	Action Entry

FIGURE 4.9 Structure of Decision Table.

It is another popular black-box testing technique. A decision table has four portions:

a. Stub portion
b. Entry portion
c. Condition portion
d. Action portion

A column in the entry portion is a *rule*. Rules indicate which actions are taken for the conditional circumstances indicated in the condition portion of the rule. Decision tables in which all conditions are binary are called *limited entry decision tables*. If conditions are allowed to have several values, the resulting tables are called *extended entry decision tables*.

To identify test cases with decision tables, we follow certain steps:

Step 1. For a module identify input conditions (causes) and action (effect).

Step 2. Develop a cause-effect graph.

Step 3. Transform this cause-effect graph, so obtained in step 2 to a decision table.

Step 4. Convert decision table rules to test cases. Each column of the decision table represents a test case. That is,

Number of Test Cases = Number of Rules

For a limited entry decision table, if n conditions exist, there must be 2^n rules.

4.3.2. ADVANTAGES, DISADVANTAGE, AND APPLICATIONS OF DECISION TABLES

Advantages of Decision Tables

1. This type of testing also works iteratively. The table that is drawn in the first iteration acts as a stepping stone to derive new decision table(s) if the initial table is unsatisfactory.
2. These tables guranatee that we consider every possible combination of condition values. This is known as its "*completeness property*." This property promises a form of complete testing as compared to other techniques.
3. Decision tables are *declarative*. There is no particular order for conditions and actions to occur.

Disadvantages of Decision Tables

Decision tables do not scale up well. We need to "factor" large tables into smaller ones to remove redundancy.

Applications of Decision Tables

This technique is useful for applications characterized by any of the following:

a. Prominent if-then-else logic.
b. Logical relationships among input variables.
c. Calculations involving subsets of the input variables.
d. Cause-and-effect relationships between inputs and outputs.
e. High cyclomatic complexity.

Technique: To identify test cases with decision tables, we interpret conditions as inputs and actions as output. The rules are interpreted as test cases. Because the decision table can mechanically be forced to be complete, we know we have a comprehensive set of test cases.

Example of a Decision Table: The triangle problem

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}
C_1: a < b + c?	F	T	T	T	T	T	T	T	T	T	T
C_2: b < a + c?	—	F	T	T	T	T	T	T	T	T	T
C_3: c < a + b?	—	—	F	T	T	T	T	T	T	T	T
C_4: c = b?	—	—	—	T	T	T	T	F	F	F	F
C_5: a = c?	—	—	—	T	T	F	F	T	T	F	F
C_6: b = c?	—	—	—	T	F	T	F	T	F	T	F
a_1: Not a triangle	×	×	×								
a_2: Scalene											×
a_3: Isosceles							×		×	×	
a_4: Equilateral				×							
a_5: Impossible					×	×		×			

FIGURE 4.10 Example of Decision Table.

Each "-" (hyphen) in the decision table represents a "don't care" entry. Use of such entries has a subtle effect on the way in which complete decision tables are recognized. For limited entry decision tables, if n conditions exist, there must be 2^n rules. When don't care entries indicate that the condition is irrelevant, we can develop a rule count as follows:

Rule 1. Rules in which no "don't care" entries occur count as one rule.

Note that each column of a decision table represents a rule and the number of rules is equal to the number of test cases.

Rule 2. Each "don't care" entry in a rule doubles the count of that rule.

Note that in this decision table we have 6 conditions (C_1—C_6). Therefore,

$$n = 6$$

Also, we can have 2^n entries, i.e., $2^6 = 64$ entries. Now we establish the rule and the rule count for the above decision table.

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	
C_1: a < b + c?	F	T	T	T	T	T	T	T	T	T	T	
C_2: b < a + c?	—	F	T	T	T	T	T	T	T	T	T	
C_3: c < a + b?	—	—	F	T	T	T	T	T	T	T	T	
C_4: a = b?	—	—	—	T	T	T	T	F	F	F	F	
C_5: a = c?	—	—	—	T	T	F	F	T	T	F	F	
C_6: b = c?	—	—	—	T	F	T	F	T	F	T	F	
Rule Count	32	16	8	1	1	1	1	1	1	1	1	= 64
a_1: Not a triangle	×	×	×									
a_2: Scalene											×	
a_3: Isosceles						×		×	×			
a_4: Equilateral				×								
a_5: Impossible					×	×		×				

FIGURE 4.11 Decision Table With Rule Counts.

From the previous table we find that the rule count is 64. And we have already said that $2^n = 2^6 = 64$. So, both are 64.

The question, however, is to find out why the rule count is 32 for the Rule-1 (or column-1)?

We find that there are 5 don't cares in Rule-1 (or column-1) and hence $2^n = 2^5 = 32$. Hence, the rule count for Rule-1 is 32. Similarly, for Rule-2, it is $2^4 = 16$ and $2^3 = 8$ for Rule-3. However, from Rule-4 through Rule-11, the number of don't care entries is 0 (zero). So rule count is $2^0 = 1$ for all these columns. Summing the rule count of all columns (or R_1-R_{11}) we get a total of 64 rule count.

Many times some problems arise with these decision tables. Let us see how.

Consider the following example of a redundant decision table:

Conditions	1–4	5	6	7	8	9
C_1	T	F	F	F	F	T
C_2	—	T	T	F	F	F
C_3	—	T	F	T	F	F
a_1	×	×	×	—	—	—
a_2	—	×	×	×	—	×
a_3	×	—	×	×	×	×

FIGURE 4.12 Redundant Decision Table.

Please note that the action entries in Rule-9 and Rules 1–4 are NOT identical. It means that if the decision table were to process a transaction in which C_1 is true and both C_2 and C_3 are false, both rules 4 and 9 apply. We observe two things

1. Rules 4 and 9 are *in-consistent* because the action sets are different.
2. The whole table is *non-deterministic* because there is no way to decide whether to apply Rule-4 or Rule-9.

Also note carefully that there is a bottom line for testers now. They should take care when don't care entries are being used in a decision table.

4.3.3. EXAMPLES

4.3.3.1. TEST CASES FOR THE TRIANGLE PROBLEM USING DECISION TABLE BASED TESTING TECHNIQUE

We have already studied the problem domain for the famous triangle problem in previous chapters. Next we apply the decision table based technique on the triangle problem. The following are the test cases:

Case ID	a	b	c	Expected output
D_1	4	1	2	Not a triangle
D_2	1	4	2	Not a triangle
D_3	1	2	4	Not a triangle
D_4	5	5	5	Equilateral
D_5	?	?	?	Impossible
D_6	?	?	?	Impossible
D_7	2	2	3	Isosceles
D_8	?	?	?	Impossible
D_9	2	3	2	Isosceles
D_{10}	3	2	2	Isosceles
D_{11}	3	4	5	Scalene

FIGURE 4.13 Test Cases For Triangle Problem.

So, we get a total of 11 functional test cases out of which three are impossible cases, three fail to satisfy the triangle property, one satisfies the equilateral triangle property, one satisfies the scalene triangle property, and three ways to get an isoceles triangle.

4.3.3.2. TEST CASES FOR NEXT DATE FUNCTION

In the previous technique of equivalence partitioning we found that certain logical dependencies exist among variables in the input domain. These dependencies are lost in a Cartesian product. The decision table format allows us to use the notion of "impossible action" to denote impossible combinations of conditions. Thus, such dependencies are easily shown in decision tables.

From the problem domain for the next date function, we know that there are some critical situations like the treatment of leap years, special treatment to year 2000 (Y2K), and special treatment to December month and a 28 day month is also to be given. So, we go for 3 tries before we derive its test cases.

First try: Special treatment to leap years.
Second try: Special treatment to year = 2000.
Third try: Special treatment to December month and days = 28.

First try: Special treatment to leap years.

The art of testing and the actual craftsmanship lies in identifying appropriate conditions and actions. Considering the following equivalence classes again:

M_1 = {Month: Month has 30 days}
M_2 = {Month: Month has 31 days}
M_3 = {Month: Month is February}
D_1 = {day: $1 \le$ day ≤ 28}
D_2 = {day: day = 29}
D_3 = {day: day = 30}
D_4 = {day: day = 31}
Y_1 = {year: year is a leap year}
Y_2 = {year: year is not a leap year}

Based on these classes, we draw the decision table:

Conditions								
C_1: month in M_1	T							
C_2: month in M_2		T						
C_3: month in M_3			T					
C_4: day in D_1								
C_5: day in D_2				:				
C_6: day in D_3					:			
C_7: day in D_4						:	:	:
C_8: year in y_1								
a_1: Impossible	:	:	:	:	:	:	:	:
a_2: Next date								

FIGURE 4.14 First Try Decision Table.

Herein, we have $3 \times 4 \times 2 = 24$ elements and $2^n = 2^8 = 256$ entries. Many of the rules would be impossible (note that the classes Y_1 and Y_2 collapse into one condition C_8). Some of these rules are impossible because

a. there are too many days in a month.

b. they cannot happen in a non-leap year.

c. they compute the next date.

Second try: Special treatment given to year 2000.

As we know, the year 2000 is a leap year. Hence, we must give special treatment to this year by creating a new class-Y_1 as follows:

M_1 = {month: month has 30 days}
M_2 = {month: month has 31 days}
M_3 = {month: month is February}
D_1 = {day: $1 \le$ day ≤ 28}
D_2 = {day: day = 29}
D_3 = {day: day = 30}
D_4 = {day: day = 31}
Y_1 = {year: year = 2000}
Y_2 = {year: year is a leap year}
Y_3 = {year: year is a common year}

This results in $3 \times 4 \times 3 = 36$ rules that represents the Cartesian product of 3 month classes (M_1.... M_3), 4 day classes (D_1....D_4), and 3 year classes.

So, after second try, our decision table is

Why is the rule count value 3 in column 1?

In column 1 or Rule 1 of this decision table, we have 3 possibilities with don't care:

$\{M_1, D_1, Y_1\}$
$\{M_1, D_1, Y_2\}$
$\{M_1, D_1, Y_3\}$

i.e., with "–" or don't care we have either Y_1 or Y_2 or Y_3.

Also note that in Rule 8, we have three impossible conditions, shown as "?" (question mark). Also, we find that this table has certain problems with December month. We will fix these next in the third try.

Third try: Special treatment to December month and to number of days = 28.

Rules → **Conditions** ↓	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	
C1: month in C2: day in C3: year in	M1 D1 –	M1 D2 –	M1 D3 –	M1 D4 –	M2 D1 –	M2 D2 –	M2 D3 –	M2 D4 –	M3 D1 Y1	M3 D1 Y2	M3 D1 Y3	M3 D2 Y1	M3 D2 Y2	M3 D2 Y3	M3 D3 –	M3 D4 –	
Rule count	3	3	3	3	3	3	3	3	1	1	1	1	1	1	3	3	36 rule count
Actions a1: impossible				×								×		×	×	×	
a2: increment day	×	×			×	×	×			×							
a3: reset day			×					×	×		×		×				
a4: increment month			×					?	×		×		×				
a5: reset month								?									
a6: increment year								?									

FIGURE 4.15 Second Try Decision Table With 36 Rule Count.

Because we know that we have serious problems with the last day of last month, i.e., December. We have to change month from 12 to 1. So, we modify our classes as follows:

M_1 = {month: month has 30 days}
M_2 = {month: month has 31 days except December}
M_3 = {month: month is December}
D_1 = {day: $1 \leq$ day ≤ 27}
D_2 = {day: day = 28}
D_3 = {day: day = 29}
D_4 = {day: day = 30}
D_5 = {day: day = 31}
Y_1 = {year: year is a leap year}
Y_2 = {year is a common year}

The Cartesian product of these contain 40 elements. Here, we have a 22-rule decision table. This table gives a clearer picture of the Next Date function than does the 36-rule decision table and is given below:

In this table, the first five rules deal with 30-day months. Notice that the leap year considerations are irrelevant. Rules (6 – 10) and (11 – 15) deal with 31-day months where the first five with months other than December and the second five deal with December. No impossible rules are listed in this portion of the decision table.

Still there is some redundancy in this table. Eight of the ten rules simply increment the day. Do we really require eight separate test cases for this sub-function? No, but note the type of observation we get from the decision table.

Finally, the last seven rules focus on February and leap year. This decision table analysis could have been done during the detailed design of the Next Date function.

Further simplification of this decision table can also be done. If the action sets of two rules in a decision table are identical, there must be at least one condition that allows two rules to be combined with a don't care entry. In a sense, we are identifying equivalence classes of these rules. For example, rules 1, 2, and 3 involve day classes as D_1, D_2, and D_3 (30 day classes). These can be combined together as the action taken by them is the same. Similarly, for other rules other combinations can be done. The corresponding test cases are shown in the table as in Figure 4.17.

Rules → Conditions ↓	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
C1: month in C2: day in C3: year in	M1 D1 –	M1 D2 –	M1 D3 –	M1 D4 –	M1 D5 –	M2 D1 –	M2 D2 –	M2 D3 –	M2 D4 –	M2 D5 –	M3 D1 –	M3 D2 –	M3 D3 –	M3 D4 –	M3 D5 –	M4 D1 –	M4 D2 Y1	M4 D2 Y2	M4 D3 Y1	M4 D3 Y2	M4 D4 –	M4 D5 –
Actions a1: impossible					×															×	×	×
a2: increment day	×	×	×		×	×	×	×		×	×	×	×	×		×	×					
a3: reset day				×						×					×			×	×			
a4: increment month				×						×								×	×			
a5: reset month															×							
a6: increment year															×							

FIGURE 4.16 Decision Table for the Next Date Function.

Case ID	Month	Day	Year	Expected output
1–3	April	15	2001	April 16, 2001
4	April	30	2001	May 1, 2001
5	April	31	2001	Impossible
6–9	January	15	2001	January 16, 2001
10	January	31	2001	February 1, 2001
11–14	December	15	2001	December 16, 2001
15	December	31	2001	January 1, 2002
16	February	15	2001	February 16, 2001
17	February	28	2004	February 29, 2004
18	February	28	2001	March 1, 2001
19	February	29	2004	March 1, 2004
20	February	29	2001	Impossible
21–22	February	30	2001	Impossible

FIGURE 4.17 Test Cases for Next Date Problem Using Decision Table Based Testing.

Because, we have 22 rules there are 22 test cases that are listed above.

4.3.3.3. Test cases for the commission problem

The commission problem is not suitable to be solved using this technique of decision table analysis because very little decisional logic is used in the problem. The variables in the equivalence classes are truly independent, therefore no impossible rules will occur in a decision table in which conditions correspond to the equivalence classes. Thus, we will have the same test cases as we did for equivalence class testing.

4.3.4. Guidelines for Decision Table Based Testing

The following guidelines have been found after studying the previous examples:

1. This technique works well where lot of decision making takes place such as the triangle problem and next date problem.
2. The decision table technique is indicated for applications characterized by any of the following:
   - Prominant if-then-else logic.
   - Logical relationships among input variables.
   - Calculations involving subsets of the input variables.
   - Cause-and-effect relationships between inputs and outputs.
   - High cyclomatic complexity.

3. Decision tables do not scale up well. We need to "factor" large tables into smaller ones to remove redundancy.
4. It works iteratively meaning that the table drawn in the first iteration, and acts as a stepping stone to design new decision tables, if the initial table is unsatisfactory.

4.4. CAUSE-EFFECT GRAPHING TECHNIQUE

Cause-effect graphing is basically a hardware testing technique adapted to software testing. It is a black-box method. It considers only the desired external behavior of a system. This is a testing technique that aids in selecting test cases that logically relate causes (inputs) to effects (outputs) to produce test cases.

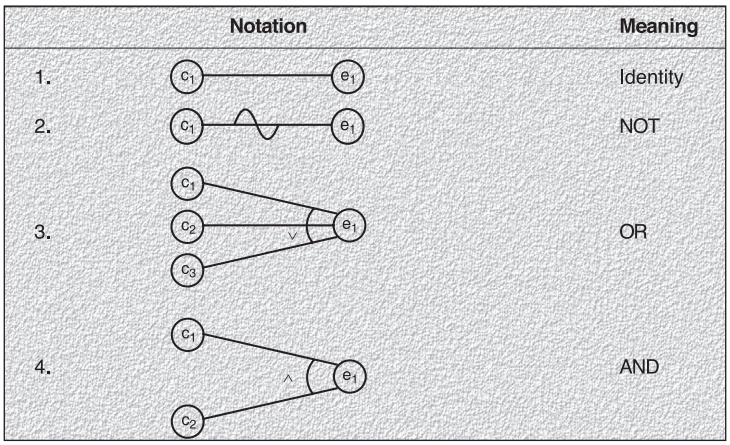
4.4.1. CAUSES AND EFFECTS

A "cause" represents a distinct input condition that brings about an internal change in the system. An "effect" represents an output condition, a system transformation, or a state resulting from a combination of causes.

Myer suggests the following steps to derive test cases:

Step 1. For a module, identify the input conditions (causes) and actions (effect).
Step 2. Develop a cause-effect graph.
Step 3. Transform the cause-effect graph into a decision table.
Step 4. Convert decision table rules to test cases. Each column of the decision table represents a test case.

Basic cause-effect graph symbols used are given below:



Consider each node as having the value 0 or 1 where 0 represents the "absent state" and 1 represents the "present state." Then the identity function states that if c_1 is 1, e_1 is 1, or we can say if c_1 is 0, e_1 is 0.

The NOT function states that if C_1 is 1, e_1 is 0 and vice versa. Similarly, the OR function states that if C_1 or C_2 or C_3 is 1, e_1 is 1 else e_1 is 0. The AND function states that if both C_1 and C_2 are 1, e_1 is 1; else e_1 is 0. The AND and OR functions are allowed to have any number of inputs.

4.4.2. TEST CASES FOR THE TRIANGLE PROBLEM

We follow the steps listed in Section 4.4.1 to design the test cases for our triangle problem:

Step 1. First, we must identify the causes and its effects. The causes are:

C_1: Side x is less than sum of y and z
C_2: Side y is less than sum of x and z
C_3: Side z is less than sum of x and y
C_4: Side x is equal to side y
C_5: Side x is equal to side z
C_6: Side y is equal to side z

The effects are:

e_1: Not a triangle
e_2: Scalene triangle
e_3: Isosceles triangle
e_4: Equilateral triangle
e_5: Impossible

Step 2. Its cause-effect graph is shown in Figure 4.18.

Step 3. We transform it into a decision table:

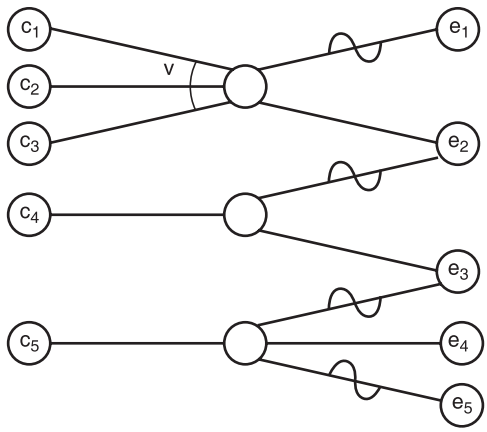


FIGURE 4.18 Cause Effect Graph for Triangle Problem.

Conditions	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}
C_1: x < y + z?	0	1	1	1	1	1	1	1	1	1	1
C_2: y < x + z?	×	0	1	1	1	1	1	1	1	1	1
C_3: z < x + y?	×	×	0	1	1	1	1	1	1	1	1
C_3: x = y?	×	×	×	1	1	1	1	0	0	0	0
C_4: x = z?	×	×	×	1	1	0	0	1	1	0	0
C_5: x = z?	×	×	×	1	1	0	0	1	1	0	0
C_6: y = z?	×	×	×	1	0	1	0	1	0	1	0
e_1: Not a triangle	1	1	1								
e_2: Scalene											1
e_3: Isosceles							1		1	1	
e_4: Equilateral				1							
e_5: Impossible					1	1		1			

Step 4. Because there are 11 rules, we get 11 test cases and they are:

Test case	x	y	z	Expected output
1	4	1	2	Not a triangle
2	1	4	2	Not a triangle
3	1	2	4	Not a triangle
4	5	5	5	Equilateral
5	?	?	?	Impossible
6	?	?	?	Impossible
7	2	2	3	Isosceles
8	?	?	?	Impossible
9	2	3	2	Isosceles
10	3	2	2	Isosceles
11	3	4	5	Scalene

4.4.3. TEST CASES FOR PAYROLL PROBLEM

Problem 1. Consider the payroll system of a person.

a. If the salary of a person is less than \$70,000 and expenses do not exceed \$30,000, then a 10% tax is charged by the IT department.

b. If the salary is greater than \$60,000 and less than or equal to \$3000 and expenses don't exceed \$40,000, then a 20% tax is charged by the IT department.

c. For a salary greater than \$3000, a 5% additional surcharge is also charged.

d. If expenses are greater than \$40,000, the surcharge is 9%.

Design test-cases using decision table based testing technique.

Solution. See the following steps:

Step 1. All causes and their effects are identified:

Causes	Effects
C_1: Salary < = 70,000	E_1: 10% tax is charged.
C_2: Salary > 60,000 and Salary < = 3000	E_2: 20% tax is charged.
C_3: Salary > 3000	E_3: (20% tax) + (5% surcharge) is charged.
C_4: Expenses < = 30,000	E_4: (20% tax) + (9% surcharge) is charged.
C_5: Expenses < = 40,000	
C_6: Expenses > 40,000	

Step 2. It's cause-effect graph is drawn.

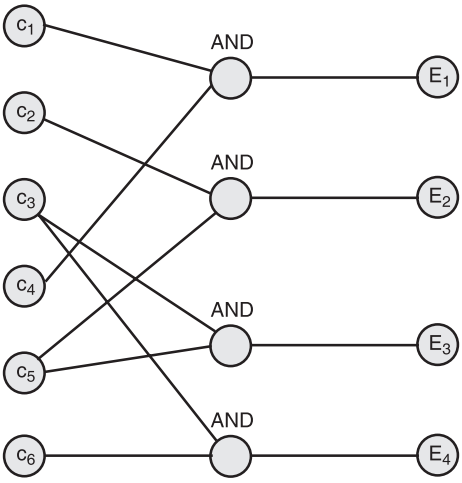


FIGURE 4.19 Cause Effects Graph.

Step 3. We *transform* this cause-effect graph into a decision table. Please note that these "causes" and "effects" are nothing else but "conditions" and "actions" of our decision table. So, we get:

		1	2	3	4
Conditions (or Causes)	C_1	1	0	0	0
	C_2	0	1	0	0
	C_3	0	0	1	1
	C_4	1	0	0	0
	C_5	0	1	1	0
	C_6	0	0	0	1
Actions (or Effects)	E_1	×	—	—	—
	E_2	—	×	—	—
	E_3	—	—	×	—
	E_4	—	—	—	×

FIGURE 4.20 Decision Table.

That is, if C_1 and C_4 are 1 (or true) then the effect (or action) is E_1. Similarly, if C_2 and C_5 is 1 (or true), action to be taken is E_2, and so on.

Step 4. Because there are 4 rules in our decision table above, we must have at least 4 test cases to test this system using this technique.

These test cases can be:

1. Salary = 20,000, Expenses = 2000
2. Salary = 100,000, Expenses = 10,000
3. Salary = 300,000, Expenses = 20,000
4. Salary = 300,000, Expenses = 50,000

So we can say that a decision table is used to derive the test cases which can also take into account the boundary values.

4.4.4. GUIDELINES FOR THE CAUSE-EFFECT FUNCTIONAL TESTING TECHNIQUE

1. If the variables refer to physical quantities, domain testing and equivalence class testing are indicated.

2. If the variables are independent, domain testing and equivalence class testing are indicated.
3. If the variables are dependent, decision table testing is indicated.
4. If the single-fault assumption is warranted, boundary value analysis (BVA) and robustness testing are indicated.
5. If the multiple-fault assumption is warranted, worst-case testing, robust worst-case testing, and decision table testing are identical.
6. If the program contains significant exception handling, robustness testing and decision table testing are indicated.
7. If the variables refer to logical quantities, equivalence class testing and decision table testing are indicated.

4.5. COMPARISON ON BLACK-BOX (OR FUNCTIONAL) TESTING TECHNIQUES

4.5.1. TESTING EFFORT

The functional methods that we have studied so far vary both in terms of the number of test cases generated and the effort to develop these test cases. To compare the three techniques, namely, boundary value analysis (BVA), equivalence class partitioning, and decision table based technique, we consider the following curve shown in Figure 4.21.

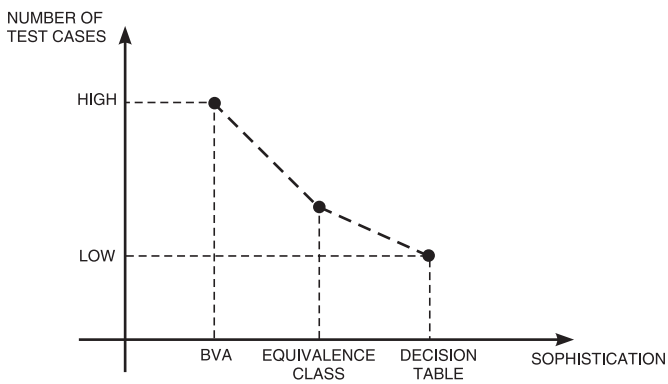


FIGURE 4.21 Test Cases as Per the Testing Method.

The domain-based techniques have no recognition of data or logical dependencies. They are very mechanical in the way they generate test cases. Because of this, they are also easy to automate. The techniques like equivalence class testing focus on data dependencies and thus we need to show our craft. The thinking goes into the identification of the equivalence classes and after that, the process is mechanical. *Also note that from the graph, the decision table based technique is the most sophisticated because it requires the tester to consider both data and logical dependencies.* As we have seen in our example of the next date function, we had to go three times but once we get a good and healthy set of conditions, the resulting test cases are complete and minimal.

Now, consider another graph to show the effort required to identify the test cases versus its sophistication.

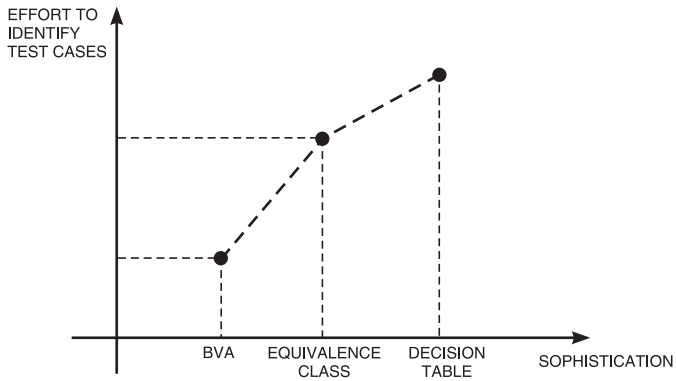


FIGURE 4.22 Test Case Identification Effort as per Testing Method.

We can say that the effort required to identify test cases is the lowest in BVA and the highest in decision tables. The end result is a trade-off between the test case effort identification and test case execution effort. If we shift our effort toward more sophisticated testing methods, we reduce our test execution time. This is very important as tests are usually executed several times. *Also note that, judging testing quality in terms of the sheer number of test cases has drawbacks similar to judging programming productivity in terms of lines of code.*

The examples that we have discussed so far show these trends.

4.5.2. Testing Efficiency

What we found in all of these functional testing strategies is that either the functionality is untested or the test cases are redundant. So, gaps do occur in functional test cases and these gaps are reduced by using more sophisticated techniques.

We can develop various ratios of the total number of test cases generated by method-A to those generated by method-B or even ratios on a test case basis. This is more difficult but sometimes management demands numbers even when they have little meaning. When we see several test cases with the same purpose, sense redundancy, detecting the gaps is quite difficult. If we use only functional testing, the best we can do is compare the test cases that result from two methods. In general, the more sophisticated method will help us recognize gaps but nothing is guaranteed.

4.5.3. Testing Effectiveness

How can we find out the effectiveness of the testing techniques?

a. By being dogmatic, we can select a method, use it to generate test cases, and then run the test cases. We can improve on this by not being dogmatic and allowing the tester to choose the most appropriate method. We can gain another incremental improvement by devising appropriate hybrid methods.

b. The second choice can be the structural testing techniques for the test effectiveness. This will be discussed in subsequent chapters.

Note, however, that the best interpretation for testing effectiveness is most difficult. We would like to know how effective a set of test cases is for finding faults present in a program. This is problematic for two reasons.

1. It presumes we know all the faults in a program.

2. Proving that a program is fault free is equivalent to the famous halting problem of computer science, which is known to be impossible.

What Is the Best Solution?

The best solution is to work backward from fault types. Given a particular kind of fault, we can choose testing methods (functional and structural) that are likely to reveal faults of that type. If we couple this with knowledge of the most likely kinds of faults, we end up with a pragamatic approach to testing effectiveness. This is improved if we track the kinds of faults and their frequencies in the software we develop.

4.5.4. GUIDELINES FOR FUNCTIONAL TESTING

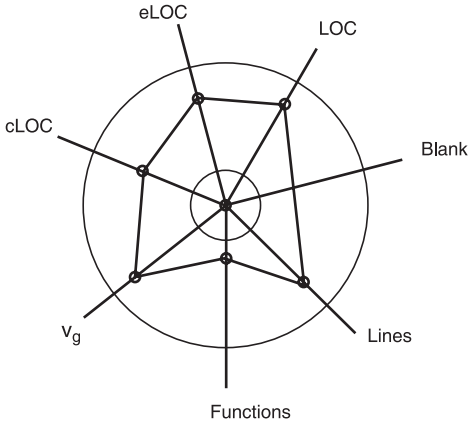
1. If the variables refer to physical quantities then domain testing and equivalence class testing are used.
2. If the variables are independent then domain testing and equivalence class testing are used.
3. If the variables are dependent, decision table testing is indicated.
4. If the single-fault assumption is warranted then BVA and robustness testing are used.
5. If the multiple-fault assumption is warranted then worst case testing, robust worst case testing, and decision table based testing are indicated.
6. If the program has significant exception handling, robustness testing and decision table based testing are identical.
7. If the variable refers to logical quantities, equivalence class testing and decision table testing are indicated.

4.6. KIVIAT CHARTS

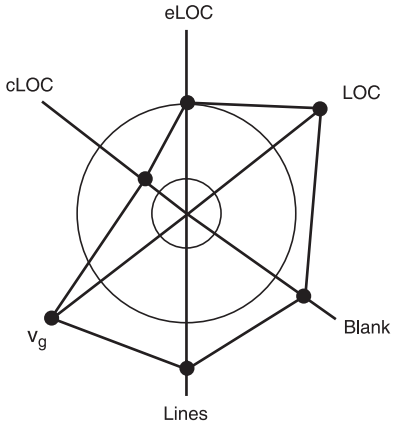
A kiviat chart visually displays a set of metrics that provides easy viewing of multiple metrics against minimum and maximum thresholds. Each radial of a Kiviat chart is a metric. All metrics are scaled so that all maximums are on a common circle and all minimums are on a common circle.

In the charts below, the circle is the maximum threshold and the band is the minimum threshold. The band between the two is the acceptable range.

The chart on the left shows that all metrics are well within the acceptable range. The chart on the right shows an example where all metrics are above maximum limits.



Good. All metrics within limits.



Bad. All but one (cLOC) of the metrics above high limit.

FIGURE 4.23 DevCodeMetricsWeb screen shot with Kiviat: *http://devcodemetrics.sourceforge.net/*

The best graphical or visual representation of complex concepts and data can be communicated through Kiviat charts. In addition, Kiviat charts provide a quick focus on the areas that are in most need of attention and can be customized to use different scales to represent various problems. This deliverable will help you get smart on what a Kiviat chart is and how to use it properly.

NOTE *For an update on Kiviat charts using R! visit: http://metrico.statanything.com/diagramas-kiviat-spider-charts-em-r/*

4.6.1. THE CONCEPT OF BALANCE

Why Is It Needed?

The problem – multiple dimensions:

- Computer: processor
  - printer
  - disks
  - CDROM
  - modem
- Software
- Personnel

All these together create multiple dimensions and hence the concept of balance is needed.

Kiviat Charts: The Method

The following steps are usually followed to draw Kiviat charts:

Step 1: Choose factors to be measured.
Step 2: Define factors so that for half the optimum utilization is 1 and the other half is 0.
Step 3: Mark the factors around the chart so that axes with an optimum of 0 alternate with those with an optimum of 1.
Step 4: Mark the values on the factor axes.
Step 5: Join the marks to form a "star."
Step 6: Evaluate the result.

We shall now consider four different cases to draw Kiviat charts.

Case 1: Kiviat charts—A perfectly balanced system.

Consider a sample data given below. Its Kiviat chart is also shown below:

A	0.8
B	0.2
C	0.8
D	0.2
E	0.8
F	0.2
G	0.8
H	0.2

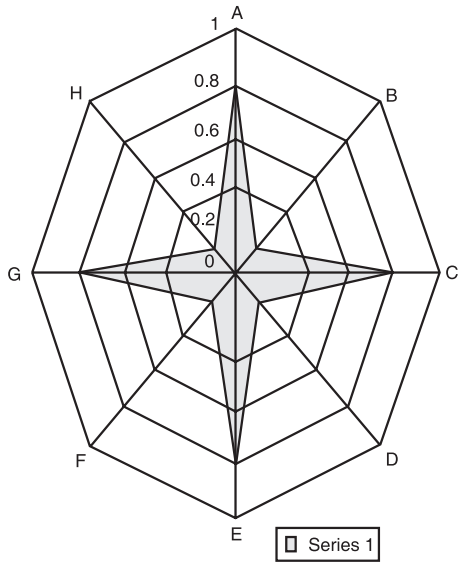


FIGURE 4.24

Case 2: Kiviat charts—A well-balanced system.

Consider another sample data for a well-balanced system. Using the steps listed earlier, we draw its kiviat chart.

A	0.9
B	0.15
C	0.85
D	0.2
E	0.9
F	0.1
G	0.8
H	0.2

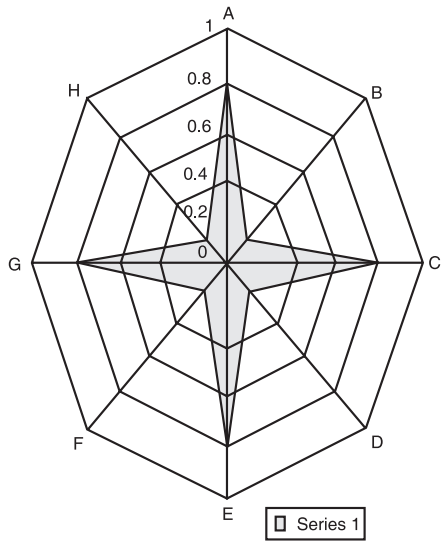


FIGURE 4.25

Case 3: Kiviat charts—A poorly balanced system.

Now consider a poorly balanced system. Its kiviat chart is shown below:

A	0.4
B	0.6
C	0.5
D	0.4
E	0.7
F	0.4
G	0.6
H	0.5

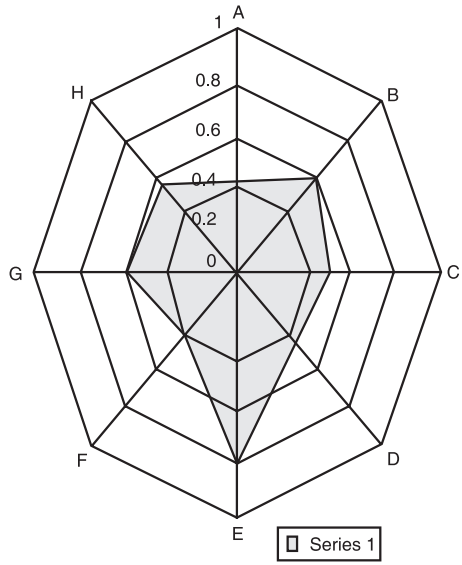


FIGURE 4.26

Case 4: Kiviat charts—A perfectly balanced system.

Consider another set of sample data. We draw its kiviat chart.

A	1
B	0
C	1
D	0
E	1
F	0
G	1
H	0

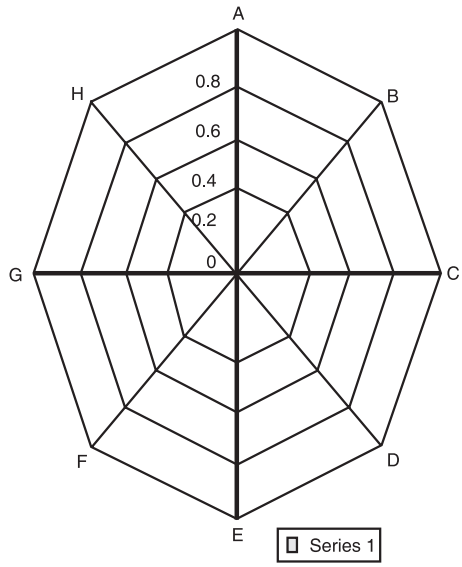


FIGURE 4.27

Kiviat Charts—A Different Example (CASE STUDY)

Problem: Comparing Internet diffusion in countries.
Measures: pervasiveness
geographic dispersion
sectoral absorption
connectivity infrastructure
organizational infrastructure
sophistication of use
Countries compared:
Finland
Jordan
Israel

Internet Dimensions Compared

Dimensions	Jordan	Israel	Finland
Pervasiveness	3	4	4
Geographic dispersion	1	4	3
Sectoral absorption	2	3	3
Connectivity infrastructure	1	2	3
Organizational infrastructure	2	4	4
Sophistication of use	1	3	4

Internet Dimensions Compared in the Form of a Kiviat Chart

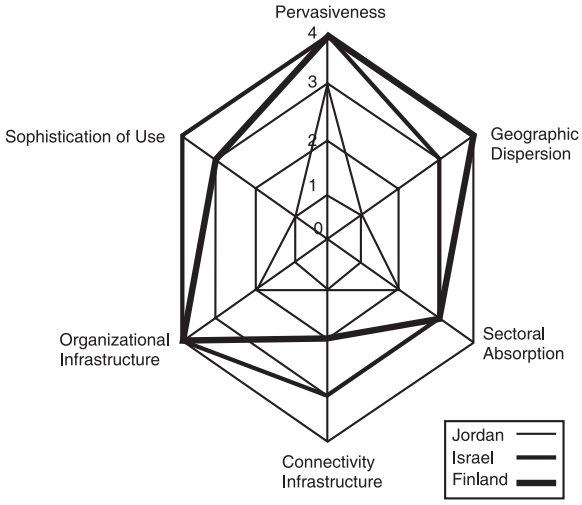


FIGURE 4.28

Problem with Kiviat Charts

The Cost/Utilization Method

- The problem with Kiviat charts - how to develop a metric for multi-dimensional data?
- The solution: express everything in terms of a common measure - cost.
- There are then two dimensions - utilization and cost - which when multiplied yield a cost/ utilization factor for each system component.

Illustrative Cost/Utilization Histograms

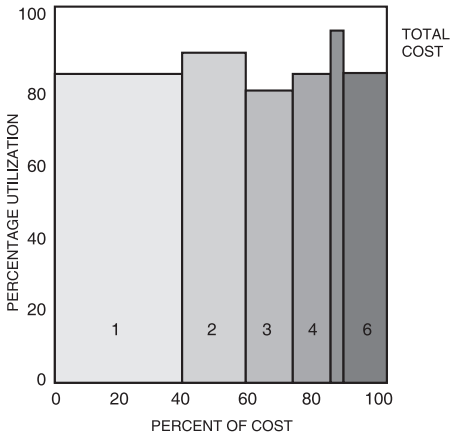


FIGURE 4.29

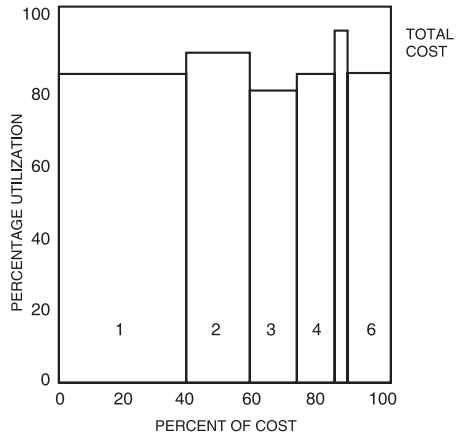


FIGURE 4.30

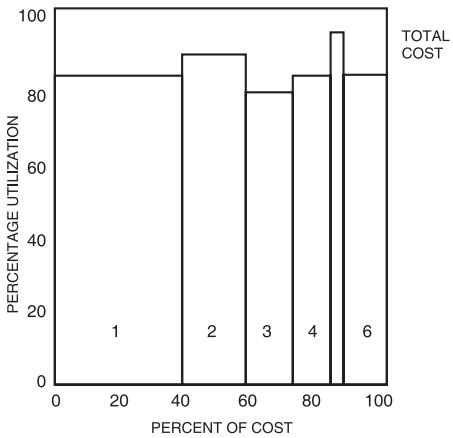


FIGURE 4.31

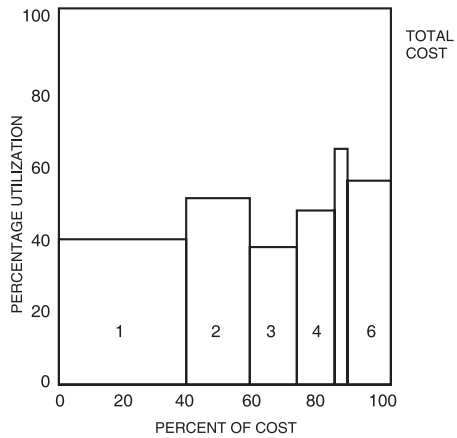


FIGURE 4.32

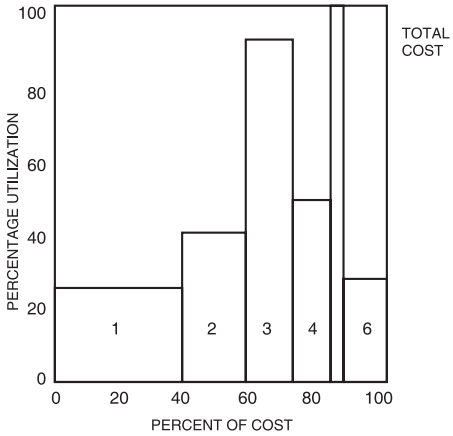


FIGURE 4.33

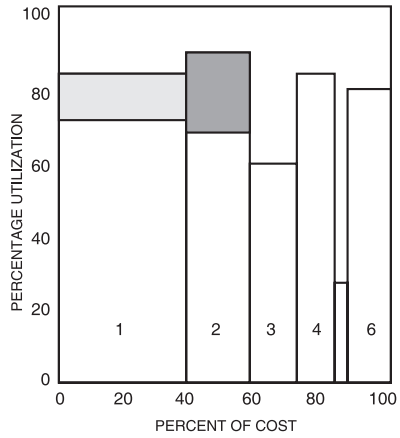


FIGURE 4.34

Cost/Utilization—The Method

The following steps are shown below:

1. Choose factors to be measured.
2. Determine the cost of each factor as a percent of total system cost.
3. Determine the utilization of each factor.
4. Prepare a chart showing the cost and utilization of each factor.
5. Compute the measure of cost/utilization, F.
6. Compute the measure of balance, B.
7. Evaluate the resulting chart and measures.

Cost/Utilization—The Measures

Cost/Utilization: $F = \sum_i u_i p_l$

where: u_i = percent utilization of factor i

p_i = cost contribution of factor i

Balance: $B = 1 - 2\sqrt{\sum (F - u_i)^2 \times p_i}$

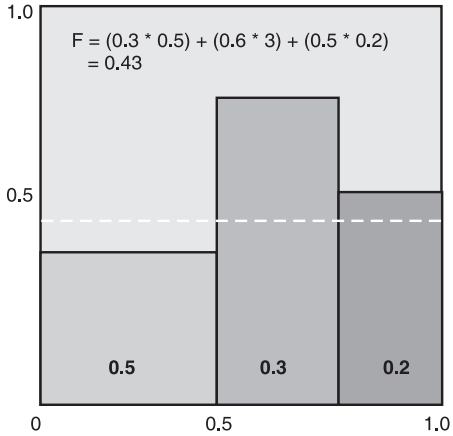


FIGURE 4.35 Cost/Utilization—The Measures.

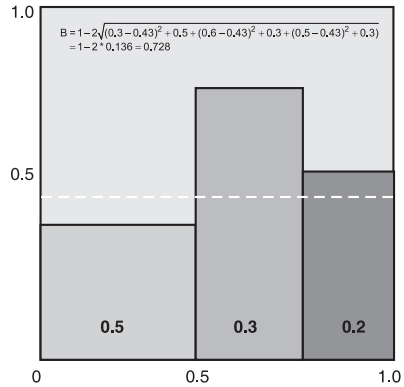


FIGURE 4.36 Cost/Utilization—The Measures.

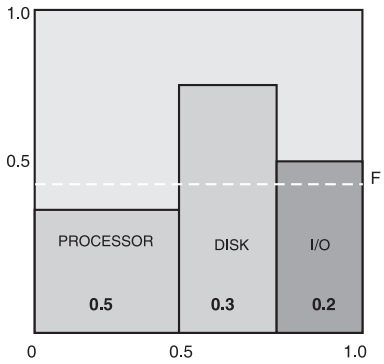


FIGURE 4.37 Cost/Utilization—Interpretation.

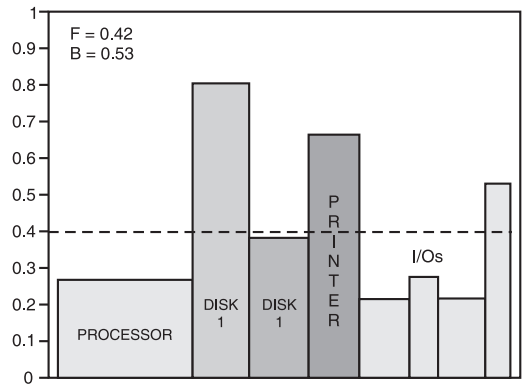


FIGURE 4.38 Cost/Utilization—Interpretation.

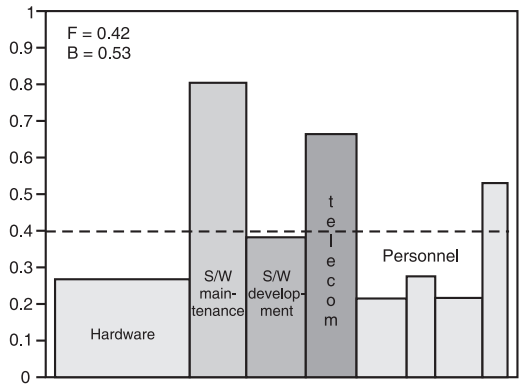


FIGURE 4.39 Cost/Utilization—Interpretation.

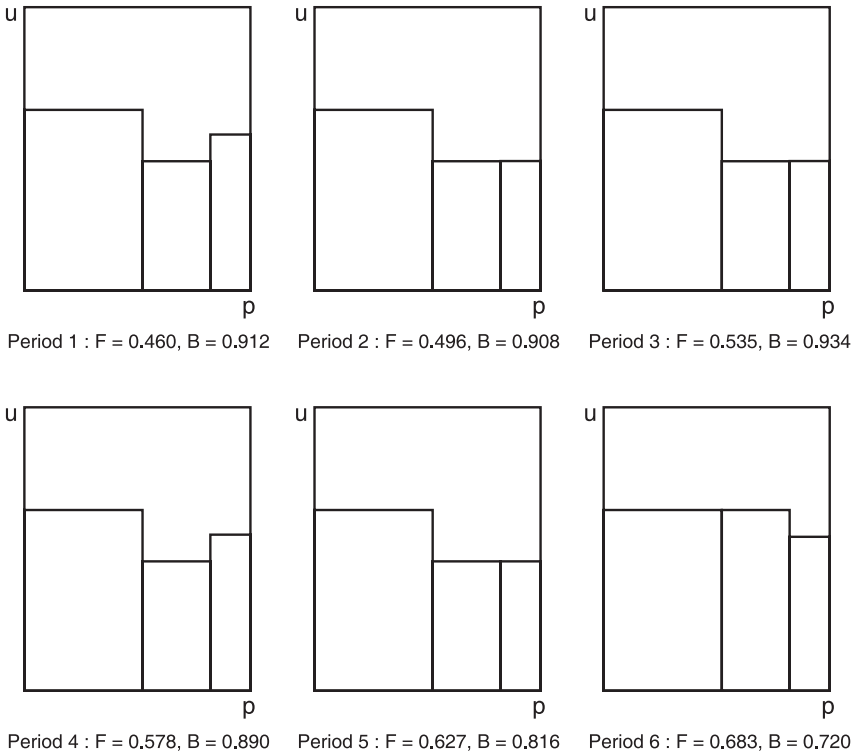


FIGURE 4.40 Trace of Cost/Utilization Criteria.

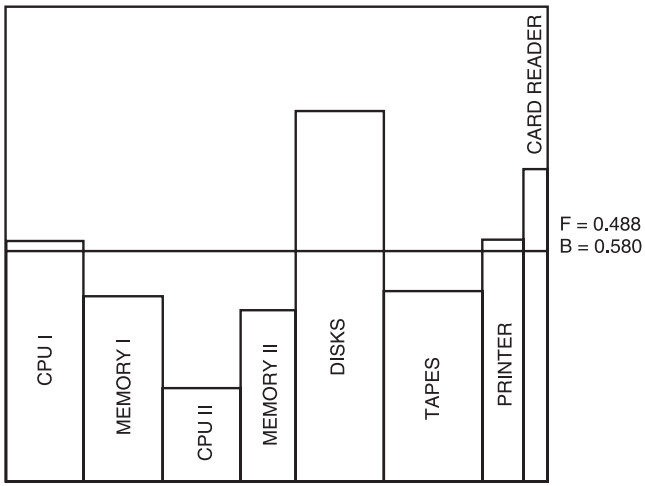


FIGURE 4.41 Composite Cost/Utilization Histogram for Two Real Linked Systems.

Conclusions

It is essential to maintain balance between system components in order to:

- reduce costs.
- maintain smooth functioning with no bottlenecks.
- attain effectiveness AND efficiency.

SUMMARY

We summarize the scenarios under which each of these techniques will be useful:

When we want to test scenarios that have	The most effective black-box testing technique to use
1. Output values dictated by certain conditions depending on values of input variables.	Decision tables
2. Input values in ranges, with each range showing a particular functionality.	Boundary Value Analysis (BVA)
3. Input values divided into classes.	Equivalence partitioning
4. Checking for expected and unexpected input values.	Positive and negative testing
5. Workflows, process flows, or language processors.	Graph based testing
6. To ensure that requirements are tested and met properly.	Requirements based testing
7. To test the domain expertise rather than product specification.	Domain testing
8. To ensure that the documentation is consistent with the product.	Documentation testing

MULTIPLE CHOICE QUESTIONS

1. Which is not a functional testing technique?

a. BVA **b.** Decision table

c. Regression testing **d.** None of the above.

2. One weakness of BVA and equivalence partitioning is
   - **a.** They are not effective.
   - **b.** They do not explore combinations of input circumstances.
   - **c.** They explore combinations of input circumstances.
   - **d.** None of the above.
3. Decision tables are useful in a situation where
   - **a.** An action is taken under varying sets of corditions.
   - **b.** A number of combinations of actions are taken under varying sets of conditions.
   - **c.** No action is taken under varying sets of conditions.
   - **d.** None of the above.
4. "Causes" and "Effects" are related to
   - **a.** Input and output
   - **b.** Output and input
   - **c.** Destination and source
   - **d.** None of the above.
5. Functionality of a software is tested by
   - **a.** White-box testing.
   - **b.** Black-box testing.
   - **c.** Regression testing.
   - **d.** None of the above.
6. If n represents the number of variables in a program then BVA yields how many test cases?
   - **a.** 4n + 2
   - **b.** 4n + 1
   - **c.** n + 2
   - **d.** n + 1
7. For a function of n variables, the robustness testing will yield
   - **a.** 6n + 1 test cases.
   - **b.** 6n + 2 test cases.
   - **c.** 6n + 4 test cases.
   - **d.** None of the above.
8. Worst case testing yields
   - **a.** 5n test cases.
   - **b.** 5n + 1.
   - **c.** 5^n.
   - **d.** None of the above.
9. BVA is based upon
   - **a.** Single fault assumption theory.
   - **b.** Multiple fault assumption theory.
   - **c.** Both of the above.
   - **d.** None of the above.

10. In decision tables, which of the following is true?

a. Number of test cases is equal to number of rules (or columns)

b. Number of test cases is not equal to number of rules (or columns)

c. Both (a) and (b)

d. None of the above.

ANSWERS

1. c.	**2.** b.	**3.** b.	**4.** a.
5. b.	**6.** b.	**7.** a.	**8.** c.
9. a.	**10.** a.		

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

Q. 1. Why we need to perform both types of testing?

Ans. A functional (Black-box) test case might be taken from the documentation description of how to perform a certain function. For example, accepting the bar code input.

On the other hand, a structural test case might be taken from a technical documentation manual.

Both methods together validate the entire system and is shown in Table.

Test Phase	Performed by	Verification	Validation
Requirements Review	Developers, Users	×	
Unit Testing	Developers		×
Integrated Testing	Developers		×
System Testing	Developers, Users		×

Q. 2. What is the source of knowledge for functional testing?

Ans. The following items are the knowledge source of functional (or black-box) testing:

a. Requirements document

b. Specifications

c. Domain knowledge

d. Defect analysis data

Q. 3. What is special value testing?

Ans. It is a sort of functional testing technique. It is most intuitive and least uniform. Special value testing occurs when a tester uses his or her domain knowledge, experience with similar programs, and information about soft-spots (i.e., critical areas) to device test cases. It is also known as *adhoc testing*. It is dependent on the tester's ability. Even though special value/adhoc testing is highly subjective, it often results in a set of test cases that is more effective in revealing faults than the test cases generated by other methods.

Q. 4. What is random testing?

Ans. A testing technique in which instead of always choosing the min, min+, nom, max–, and max values of a bounded variable, we use a random number generator to pick test case values. This will reduce bias during testing.

Q. 5. Write down the differences between static and dynamic testing?

Ans. The differences between static and dynamic testing are shown below:

Static testing	Dynamic testing
1. It talks about prevention.	It talks about cure.
2. It is more cost effective.	It is less cost effective.
3. It may achieve 100% statement coverage.	It may achieve less than 50% statement coverage as it finds the errors only in the part of codes that are actually executed.
4. It is not time consuming.	It is time consuming as it may involve running several test cases.
5. It can be done before compilation.	It can be done only after executables are ready.

Q. 6. Give some advantages and disadvantages of functional test cases?

Ans. Functional test cases have two main advantages:

i. They are independent of how the software is implemented. So, even if the implementation changes, the test cases are still useful.

ii. Test case development can occur in parallel with the implementation, thereby reducing overall project development internal.

Functional test cases have two main disadvantages:

i. Some unavoidable redundancies may exist among test cases.

ii. There exists a possibility of gaps of untesting software.

Both of these problems can be solved if we combine the test cases, so obtained from both functional and structural testing.

Q. 7. Differentiate between positive and negative testing?

Ans. Let us tabulate the differences between the two:

	Positive Testing	Negative Testing
1.	Positive testing tries to prove that a given product does what it is supposed to do.	Negative testing is done to show that the product does not fail when an unexpected input is given.
2.	A positive test case is one that verifies the requirements of the product with a set of expected output.	A negative test case will have the input values that may not have been represented in the SRS. These are unknown conditions for the product.
3.	The purpose of positive testing is to prove that the product works as per the user specifications.	The purpose of negative testing is to try and break the system.
4.	Positive testing checks the product's behavior.	Negative testing covers scenarios for which the product is not designed and coded.
5.	Positive testing is done to verify the known test conditions.	Negative testing is done to break the product with unknown test conditions, i.e., test conditions that lie outside SRS.

(Continued)

6.	If all documented requirements and test conditions are covered then it gives 100% coverage.	There is no end to negative testing and 100% coverage is impractical here.
	For example: A product delivering an error when it is expected to give error.	**For example:** A product not delivering an error when it should or delivering an error when it should not.

Q. 8. What is domain testing?

Ans. Domain testing is a sort of testing strategy wherein we do not look even at SRS of a software product but test purely on the basis of domain knowledge and expertise in the domain of application.

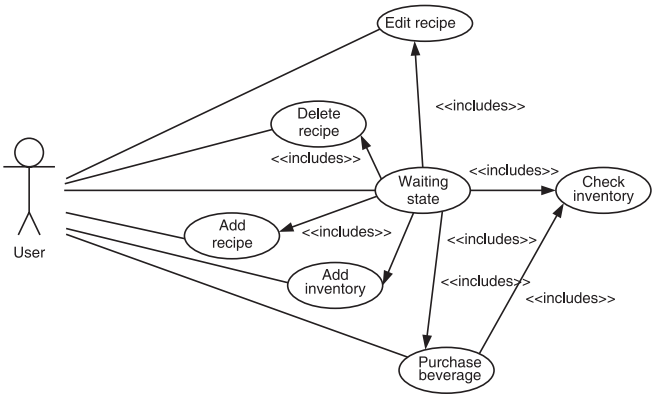
For example, in a banking software, knowing the account opening, closing, etc. processes, enables a tester to test that functionality better.

Q. 9. What is documentation testing?

Ans. A testing done to ensure that the documentation is consistent with the product.

Q. 10. What are the characteristics of a good test?

Ans.
- **i.** A good test has a high probability of finding an error.
- **ii.** A good test is not redundant.
- **iii.** A good test is the "best of breed."
- **iv.** A good test should be neither too simple nor too complex.



Q. 11. Consider the above use case diagram for coffee maker. Find at least ten acceptance test cases and black-box test cases and document it.

Ans. Test cases for coffee maker.

Preconditions: Run coffee maker by switching on power supply.

Test case id	Test case name	Test case description	Test steps		Actual result	Test status (P/F)
			Step	Expected result		
Acc01	Waiting state	When the coffee maker is not in use, it waits for user input.		System displays menu as follows: **1.** Add recipe **2.** Delete recipe **3.** Edit a recipe **4.** Add inventory **5.** Check inventory **6.** Purchase beverage		
Acc02	Add a recipe	Only three recipes may be added to the coffee maker.	Add the recipe. A recipe consists of a name, price, units of coffee, units of dairy creamer, units of chocolate, water.	Each recipe name must be unique in the recipe list.		
Acc03	Delete a recipe	A recipe may be deleted from the coffeemaker if it exists in the list of recipes in the coffee maker.	Choose the recipe to be deleted by its name.	A status message is printed and the coffee maker is returned to the waiting state.		

Test case id	Test case name	Test case description	Test steps		Actual result	Test status (P/F)
			Step	Expected result		
Acc04	Edit a recipe	The user will be prompted for the recipe name they wish to edit.	Enter the recipe name along with various units.	Upon completion, a status message is printed and the coffee maker is returned to the waiting state.		
Acc05	Add inventory	Inventory may be added to the machine at any time. (Inventory is measured in integer units.)	Type the inventory: coffee, dairy creamer, water.	Inventory is added to the machine and a status message is printed.		
Acc06	Check inventory	Inventory may be checked at any time.	Enter the units of each item.	System displays the inventory.		
Acc07	Purchase beverage	The user will not be able to purchase a beverage if they do not deposit enough money.	Enter the units and amount.	**1.** System dispensed the change, if user paid more than the price of the beverage. **2.** System returns user's money if there is not enough inventory.		

Test ID	Description/steps	Expected results	Actual results	Test status (P/F)
checkOptions	Precondition: Run CoffeeMaker Enter: 0	Program exits		
	Enter: 1	Add recipe functionality		
	Enter: 2	Delete recipe functionality		
	Enter: 3	Edit recipe functionality		
	Enter: 4	Add inventory functionality		
	Enter: 5	Inventory displays		
	Enter: 6	Make coffee functionality		
addRecipe1	Precondition: Run CoffeeMaker Enter: 1 Name: Coffee Price: 10 Coffee: 3 Dairy creamer: 1 Chocolate: 0	Coffee successfully added. Return to main menu.		
addRecipe2	Precondition: Run CoffeeMaker Enter: 1 Name: Coffee Price:10 Coffee: 3 Dairy creamer: 1 Chocolate: 0	Coffee could not be added. (Recipe name must be unique in the recipe list.) Return to main menu.		

(Continued)

Test ID	Description/steps	Expected results	Actual results	Test status (P/F)
addRecipe3	Precondition: Run CoffeeMaker Enter: 1 Name: Mocha Price: –50	Mocha could not be added. Price can not be negative. Return to main menu.		
addRecipe4	Precondition: Run CoffeeMaker Enter: 1 Name: Mocha Price: 60 Coffee: –3	Mocha could not be added. Units of coffee can not be negative. Return to main menu.		
addRecipe5	Precondition: Run CoffeeMaker Enter: 1 Name: Mocha Price: 20 Coffee: –3 Dairy creamer: –2	Mocha could not be added. Units of dairy creamer can not be negative. Return to main menu.		
addRecipe6	Precondition: Run CoffeeMaker Enter: 1 Name: Mocha Price: 20 Coffee: 3 Dairy creamer: 2 Chocolate: –3	Mocha could not be added. Units of chocolate can not be negative. Return to main menu.		
addRecipe7	Precondition: Run CoffeeMaker Enter: 1 Name: Mocha Price: a	Please input an integer. Return to main menu.		

Test ID	Description/steps	Expected results	Actual results	Test status (P/F)
addRecipe8	Precondition: Run CoffeeMaker Enter: 1 Name: Mocha Price: 20 Coffee: a	Please input an integer. Return to main menu.		
addRecipe9	Precondition: Run CoffeeMaker Enter: 1 Name: Mocha Price: 20 Coffee: 3 Dairy creamer: 2 Chocolate: a	Please input an integer. Return to main menu.		
addRecipe10	Precondition: Run CoffeeMaker Enter: 1 Name: Hot chocolate Price: 20 Coffee: 3 Dairy creamer: 2 Chocolate: 3	Coffee successfully added. Return to main menu.		
deleteRecipe1	Precondition: addRecipe1 has run successfully Enter: 2 Enter: 3	Successfully deleted. Return to main menu.		

(Continued)

Test ID	Description/steps	Expected results	Actual results	Test status (P/F)
deleteRecipe2	Precondition: Run CoffeeMaker Enter: 2	There are no recipes to delete. Return to main menu.		
editRecipe1	Precondition: addRecipe1 has run successfully Enter: 3 Name: Coffee Price: 10 Coffee: 3 Dairy creamer: 1 Chocolate: 0	Coffee successfully edited. Return to main menu.		
editRecipe2	Precondition: addRecipe1 has run successfully Enter: 3 Name: Coffee Price: 10 Coffee: 5 Dairy creamer: 4 Chocolate: 0	Coffee successfully edited. Return to main menu.		
editRecipe3	Precondition: addRecipe1 has run successfully Enter: 3 Name: Mocha Price: 20	Mocha could not be added. Price can not be negative.		

REVIEW QUESTIONS

1. Perform the following:
   a. Write a program to find the largest number.
   b. Design a test case for program at 2(a) using a decision table.
   c. Design equivalence class test cases.
2. Explain any five symbols used in the cause-effect graphing technique?
3. How do you measure:
   a. Test effectiveness?
   b. Test efficiency?
4. Write a short paragraph:
   a. Equivalence testing.
5. Explain the significance of boundary value analysis. What is the purpose of worst case testing?
6. Describe cause-effect graphing technique with the help of an example.
7. a. Discuss different types of equivalence class tests cases.
   b. Consider a program to classify a triangle. Its input is a triple of the integers (day x, y, z) and date types or input parameters ensure that they will be integers greater than zero and less than or equal to 200. The program output may be any of the following words: scalene, isosceles, equilateral, right angle triangle, not a triangle. Design the equivalence class test cases.
8. How can we measure and evaluate test effectiveness? Explain with the help of 11 step S/W testing process.
9. What is the difference between:

   Equivalence partitioning and boundary value analysis methods?
10. Consider the previous date function and design test cases using the following techniques:
    a. Boundary value analysis.
    b. Equivalence class partitioning.

    The function takes current date as an input and returns the previous date of the day as the output.

All variables have integer values subject to conditions as follows:

C1: $1 \leq \text{month} \leq 2$

C2: $1 \leq \text{day} \leq 31$

C3: $1920 \leq \text{year} \leq 2000$.

11. Compare the single/multiple fault assumption theory with boundary value and equivalence class testing.

12. Explain the cause-effect graphing technique in detail. Why is it different from other functional testing techniques?

13. Differentiate between positive and negative functional testing.

14. What is domain testing? Illustrate through a suitable example.

15. **a.** "Software testing is an incremental process." Justify the statement.

b. What do we mean by functional analysis? Discuss in brief any method to perform functional analysis.

c. Consider a program to determine whether a number is "odd" or "even" and print the message "Number is Even" or "Number is Odd." The number may be any valid integer. Design boundary value and equivalence class test cases.

16 **a.** Consider a program that calculates the roots of a quadratic equation with a, b, and c in range [1, 100]. Design test cases using boundary value and robustness testing.

b. What is decision table testing?

17. Write a short paragraph on cause-effect graphing.

18. Discuss any two model-based black-box testing approaches?

19. A program computes the grade of each student based on the average of marks obtained by them in physics, chemistry, and math. Maximum marks in each subject is 60. Grades are awarded as follows:

Marks Range	Grade
60	A+
59–60	A
49–40	B
39–30	C
29–30	D
19–0	F

Design robust test cases and identify equivalence class test cases for output and input domains for this problem.

20. What is the difference between weak normal and strong normal equivalence class testing?

21. Consider a program for the determination of previous date. Its input is a triple of day, month, and year with the values in the range:

$1 \leq month \leq 12$

$1 \leq day \leq 31$

$1900 \leq year \leq 2025$

The possible outputs are "Previous date" and "Invalid date." Design a decision table and equivalence classes for input domain.

22. Consider a program given below for the selection of the largest of numbers.

```
main ( )
     {
float A, B, C;
printf ("Enter 3 values:");
scanf ("%d%d%d", &A, &B, &C);
printf ("Largest value is");
if (A > B)
     {
     if (A > C)
          printf ("%d\n", A);
     else
          printf ("%d\n", C);
     }
     else
     {
     if (C > B)
          printf ("%d", C);
     else
          printf ("%f", B);
     }
}
```

a. Design the set of test cases using BVA technique and equivalence class testing technique.

b. Select a set of test cases that will provide 100% statement coverage.

c. Develop a decision table for this program.

23. Consider the above program and show that why is it practically impossible to do exhaustive testing?

24. a. Consider the following point-based evaluation system for a trainee salesman of an organization:

Points earned	Management action
0–20	Thank you
21–40	Extend probation
41–60	Confirmation
61–80	Promotion
81–100	Promotion with a letter of recommendation

Generate the test cases using equivalence class testing.

b. Explain functional testing.

25. Design test cases using the functional testing taking any example program? Explain how and why complete testing is not possible by highlighting some cases of the example given.

26. Let us consider an example of grading the students in an academic institution. The grading is done according to the following rules:

Marks obtained	Grade
80–100	Distinction
60–79	First division
50–59	Second division
40–49	Third division
0–39	Fail

Generate test cases using the equivalence class testing technique.

27. Consider the following point-based evaluation system for a salesman of an organization.

Points earned	Grade	Management action
80–100	A+	Raise of $10,000
75–80	A–	Raise of $5,000
70–75	A	Raise of $3000
65–70	B+	Raise of $1000
60–65	B	No increment
50–60	C	Warning

Generate the test cases using equivalence class testing.

28. Consider a program for the determination of previous date. Its input is a triple of day, year, month with values in range:

$1 \leq \text{month} \leq 12$

$1 \leq \text{day} \leq 31$

$1912 \leq \text{year} \leq 2020$

Design a decision table for the given scenario.

29. Consider the program for the determination of next date in a calender. Its input is a triple of day, month, and year with the following range:

$1 \leq \text{month} \leq 12$

$1 \leq \text{day} \leq 31$

$1900 \leq \text{year} \leq 2025$

The possible outputs would be next date or invalid date. Design boundary value, robust, and worst test cases for this program.

30. a. Explain cause-effect graphing technique in detail. Why is it different from other functional techniques?

b. Design test cases using functional testing taking any example program. Explain how and why complete testing is not possible by highlighting some cases of the example given.

31. Consider an example of grading a student in a university. The grading is done as below:

Average marks	Grade
90–100	A+
75–89	A
60–74	B
50–59	C
Below 50	Fail

The marks of any three subjects are considered for the calculation of average marks. Scholarships of $1000 and $500 are given to students securing more than 90% and 85% marks, respectively. Develop a decision table, cause effect graph, and generate test cases for the above scenario.

CHAPTER 5

WHITE-BOX (OR STRUCTURAL) TESTING TECHNIQUES

Inside this Chapter:

5.0. Introduction to White-Box Testing or Structural Testing or Clear-Box or Glass-Box or Open-Box Testing

5.1. Static Versus Dynamic White-Box Testing

5.2. Dynamic White-Box Testing Techniques

5.3. Mutation Testing Versus Error Seeding—Differences in Tabular Form

5.4. Comparison of Black-Box and White-Box Testing in Tabular Form

5.5. Practical Challenges in White-Box Testing

5.6. Comparison on Various White-Box Testing Techniques

5.7. Advantages of White-Box Testing

5.0. INTRODUCTION TO WHITE-BOX TESTING OR STRUCTURAL TESTING OR CLEAR-BOX OR GLASS-BOX OR OPEN-BOX TESTING

White-box testing is a way of testing the external functionality of the code by examining and testing the program code that realizes the external functionality. It is a methodology to design the test cases that uses the control

structure of the application to design test cases. White-box testing is used to test the program code, code structure, and the internal design flow.

5.1. STATIC VERSUS DYNAMIC WHITE-BOX TESTING

A number of defects get amplified because of incorrect translation of requirements and design into program code. We shall now study different techniques of white-box testing. As shown in Figure 5.1, white-box texting is classified as static and dynamic.

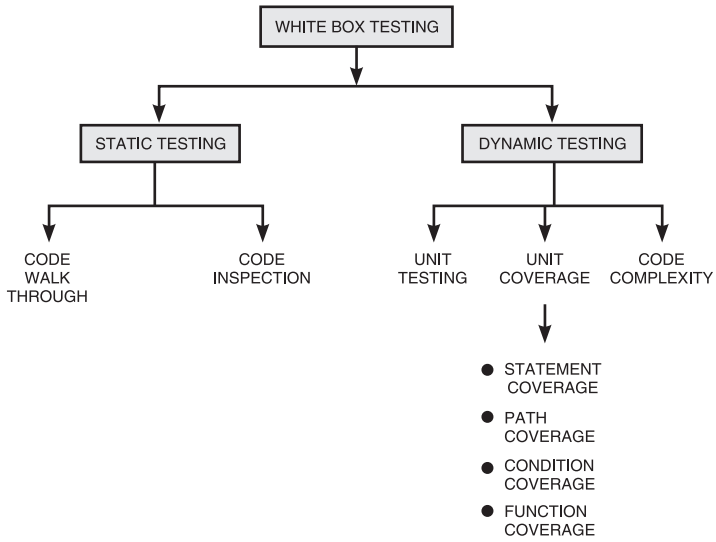


FIGURE 5.1 Classification of White-Box Testing.

As already discussed in Chapter 2, static testing is a type of testing in which the program source code is tested without running it (not the binaries or executable). We only need to examine and review the code. It means that we need not execute the code. We need to find out whether

- The code works according to the functional requirements.
- The code has been written in accordance with the design developed earlier in the project life cycle.
- The code for any functionality has been missed.
- The code handles errors properly.

Static testing can be done by humans or with the help of specialized tools. So, static white-box testing is the process of carefully and methodically reviewing the software design, architecture, or code for bugs without executing it. It is sometimes referred to as structural analysis [PATT01].

We next discuss some of the dynamic white-box testing techniques one by one.

5.2. DYNAMIC WHITE-BOX TESTING TECHNIQUES

In dynamic testing, we test a running program. So, now binaries and executables are desired. We try to test the internal logic of the program now. It entails running the actual product against some pre-designed test cases to exercise as much of the code as possible.

5.2.1. UNIT/CODE FUNCTIONAL TESTING

It is the process of testing in which the developer performs some quick checks prior to subjecting the code to more extensive code coverage testing or code complexity testing. It can be performed in many ways:

1. At the initial stages, the developer or tester can perform certain tests based on the input variables and the corresponding expected output variables. This can be a quick test. If we repeat these tests for multiple values of input variables then the confidence level of the developer to go to the next level increases.
2. For complex modules, the tester can insert some print statements in between to check whether the program control passes through all statements and loops. It is important to remove the intermediate print statements after the defects are fixed.
3. Another method is to run the product under a debugger or an integrated development environment (IDE). These tools involve single stepping of instructions and setting break points at any function or instruction.

All of these initial tests actually fall under "debugging" category rather than under "testing" category of activities. We put them under "white-box testing" head as all are related to the knowledge of code structure.

5.2.2. CODE COVERAGE TESTING

Code coverage testing involves designing and executing test cases and finding out the percentage of code that is covered by testing. The percentage of code covered by a test is found by adopting a technique called the *instrumentation of code*. These tools rebuild the code, do product linking with a set of libraries provided by the tool, monitor the portions of code covered, and report on the portions of the code that are covered frequently, so that the critical or most used portions of code can be identified. We will next discuss some basic testing techniques based on code coverage.

5.2.2.1. STATEMENT COVERAGE

In most of the programming languages, the program construct may be a sequential control flow, a two-way decision statement like if-then-else, a multi-way decision statement like switch, or even loops like while, do, repeat until and for.

Statement coverage refers to writing test cases that execute each of the program statements. We assume that the more the code is covered, the better the testing of the functionality.

For a set of sequential statements (i.e., with no conditional branches), test cases can be designed to run through from top to bottom. However, this may not always be true in two cases:

1. If there are asynchronous exceptions in the code, like divide by zero, then even if we start a test case at the beginning of a section, the test case may not cover all the statements in that section. Thus, even in the case of sequential statements, coverage for all statements may not be achieved.
2. A section of code may be entered from multiple points.

In case of an if-then-else statement, if we want to cover all the statements then we should also cover the "then" and "else" parts of the if statement. This means that we should have, for each if-then-else, at least one test case to test the "then" part and at least one test case to test the "else" part.

The multi-way, switch statement can be reduced to multiple two-way if statements. Thus, to cover all possible switch cases, there would be multiple test cases.

A good percentage of the defects in programs come about because of loops that do not function properly. More often, loops fail in what are called "boundary conditions." So, we must have test cases that

1. Skip the loop completely so that the situation of the termination condition being true before starting the loop is tested.
2. Check the loop for n = 1.
3. Try covering the loop at the boundary values of n, i.e., just below n and just above n.

The statement coverage for a program, which is an indication of the percentage of statements actually executed in a set of tests, can be calculated as follows:

$$\text{Statement Coverage} = \left[\frac{\text{Total Statements Exercised}}{\text{Total Number of Executable Statements in Program}}\right] \times 100$$

For example, if the total number of statements exercised = 08
Total number of executable statements in program = 10

$$\therefore \quad \text{Statement coverage} = \frac{08}{\cancel{10}} \times 10\cancel{0} = 80\%$$

Please note from the above discussion that as the type of statement progresses from a simple sequential statement to if-then-else and through loops, the number of test cases required to achieve statement coverage increases. As we have already seen, exhaustive testing (100%) is not possible, so exhaustive coverage of all statements in a program will be impossible for practical purposes.

Even if we get a high level of the statement coverage, it does not mean that the program is defect free.

Consider a program that implements wrong requirements and, if such a code is fully tested with say, 100% coverage, it is still a wrong program. Hence 100% code coverage does not mean anything. Consider another example.

```
    i = 0 ;
if (code = = "y")
   {
        statement -1 ;
        statement-2 ;
        :
        :
        statement - n ;
   }
   else
        result = {marks/ i} * 100 ;
```

In this program, when we test with code = "y," we will get 80% code coverage. But if the data distribution in the real world is such that 90% of the time the value of code is not = "y," then the program will fail 90% of the time because of the exception-divide by zero. Thus, even with a code coverage of 80%, we are left with a defect that hits the users 90% of the time. The path coverage technique, discussed next, overcomes this problem.

5.2.2.2. *PATH COVERAGE*

In the path coverage technique, we split a program into a number of distinct paths. A program or a part of a program can start from the beginning and take any of the paths to its completion. The path coverage of a program may be calculated based on the following formula:

$$\text{Path Coverage} = \left[\frac{\text{Total Path Exercised}}{\text{Total Number of paths in Program}}\right] \times 100$$

For example, if the total path exercised = 07

Total number of paths in program = 10

$$\therefore \quad \text{Path coverage} = \frac{07}{\cancel{10}} \times 100 = 7\cancel{0}\%$$

Consider the following flow graph in Figure 5.2.

There are different paths in this particular flow graph.

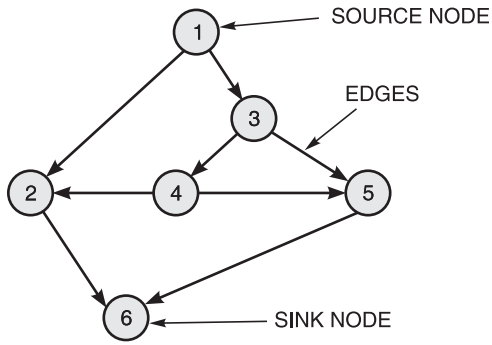


FIGURE 5.2 Flow Graph Example.

Some of the paths are:

Path – 1: $1 \rightarrow 2 \rightarrow 6$
Path – 2: $1 \rightarrow 3 \rightarrow 5 \rightarrow 6$
Path – 3: $1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$
Path – 4: $1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 6$

Regardless of the number of statements in each of these paths, if we can execute these paths, then we would have covered most of the typical scenarios.

Path coverage provides a stronger condition of coverage than statement coverage as it relates to the various logical paths in the program rather than just program statements.

5.2.2.3. CONDITION COVERAGE

In the above example, even if we have covered all the paths possible, it does not mean that the program is fully tested. Path testing is not sufficient as it does not exercise each part of the Boolean expressions, relational expressions, and so on. This technique of condition coverage or predicate monitors whether every operand in a complex logical expression has taken on every TRUE/FALSE value. Obviously, this will mean more test cases and the number of test cases will rise exponentially with the number of conditions and Boolean expressions. For example, in if-then-else, there are 2^2 or 4 possible

true/false conditions. The condition coverage which indicates the percentage of conditions covered by a set of test cases is defined by the formula:

$$\text{Condition Coverage} = \left[\frac{\text{Total Decisions Exercised}}{\text{Total Number of Decisions in Program}}\right] \times 100$$

Please note that this technique of condition coverage is much stronger criteria than path coverage, which in turn is much stronger criteria than statement coverage.

5.2.2.4. *FUNCTION COVERAGE*

In this white-box testing technique, we try to identify how many program functions are covered by test cases. So, while providing function coverage, test cases can be written so as to exercise each of different functions in the code.

The following are the advantages of this technique:

1. Functions (like functions in C) are easier to identify in a program and, hence, it is easier to write test cases to provide function coverage.
2. Because functions are at a higher level of abstraction than code, it is easier to achieve 100% function coverage.
3. It is easier to prioritize the functions for testing.
4. Function coverage provides a way of testing traceability, that is, tracing requirements through design, coding, and testing phases.
5. Function coverage provides a natural transition to black-box testing.

Function coverage can help in improving the performance as well as the quality of the product. For example, if in a networking software, we find that the function that assembles and disassembles the data packets is being used most often, it is appropriate to spend extra effort in improving the quality and performance of that function. Thus, function coverage can help in improving the performance as well as the quality of the product.

Better code coverage is the result of better code flow understanding and writing effective test cases. Code coverage up to 40–50% is usually achievable. Code coverage of more than 80% requires an enormous amount of effort and understanding of the code.

The multiple code coverage techniques discussed so far are not mutually exclusive. They supplement and augment one another. While statement coverage can provide a basic comfort factor, path, decision, and function coverage provide more confidence by exercising various logical paths and functions.

5.2.3. CODE COMPLEXITY TESTING

Two questions that come to mind are:

1. Which of the paths are independent? If two paths are not independent, then we may be able to minimize the number of tests.
2. Is there any limit to the number of tests that must be run to ensure that all the statements have been executed at least once?

The answer to the above questions is a metric that quantifies the complexity of a program and is known as *cyclomatic complexity*. It is also known as *structural complexity* because it gives the internal view of the code. It is a number which provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

5.2.3.1. CYCLOMATIC COMPLEXITY, ITS PROPERTIES AND ITS MEANING IN TABULAR FORM

McCabe IQ covers about 146 different counts and measures. These metrices are grouped according to six main "collections" each of which provides a different level of granularity and information about the code being analyzed. The collections are given below:

i. McCabe metrics based on cyclomatic complexity, V(G).
ii. Execution coverage metrics based on any of branch, path, or Boolean coverage.
iii. Code grammar metrics based around line counts and code structure counts such as nesting.
iv. OO metrics based on the work of Chidamber and Kemerer.
v. Derived metrics based on abstract concepts such as understability, maintainability, comprehension, and testability.
vi. Custom metrics imported from third-party software/systems, e.g., defect count.

McCabe IQ provides for about 100 individual metrics at the method, procedure, function, control, and section/paragraph level. Also, there are 40 metrices at the class/file and program level.

Categories of Metrics

There are three categories of metrics:

1. McCabe metrics
2. OO metrics
3. Grammar metrics

Please remember that when collecting metrics, we rely upon subordinates who need to "buy into" the metrics program. Hence, it is important to only collect what you intend to use.

We should keep in mind, the Hawthorne Effect which states that when you collect metrics on people, the people being measured will change their behavior. Either of these practices will destroy the efficiency of any metrics program.

The three metrics categories are explained below.

I. McCabe metrics

a. **Cyclomatic complexity, V(G):** It is the measure of the amount of logic in a code module of 3rd and 4th generation languages. If V(G) is excessively high then it leads to impenetrable code, i.e., a code that is at a higher risk due to difficulty in testing. The threshold value is 10. When V(G) > 10, then the likelihood of code being unreliable is much higher. Please remember that a high V(G) shows a decreased quality in the code resulting in higher defects that become costly to fix.

b. **Essential complexity:** It is a measure of the degree to which a code module contains unstructured constructs. If the essential complexity is excessively high, it leads to impenetrable code, i.e., a code that is at higher risk due to difficulty in testing. Furthermore, the higher value will lead to increased cost due to the need to refactor, or worse, reengineer the code. The threshold value is 4. When the essential complexity is more than 4 then the likelihood of the code being unmaintainable is much higher. Please note that a high essential complexity indicates increased maintenance costs with decreased code quality. Some organizations have used the *essential density metric (EDM)* defined as:

$$\text{EDM} = \frac{\text{Essential Complexity}}{\text{Cyclomatic Complexity}}$$

c. **Integration complexity:** It is a measure of the interaction between the modules of code within a program. Say, S0 and S1 are two derivatives of this complexity.

where S0 — Provides an overall measure of size and complexity of a program's design. It will not reflect the internal calculations of each module. It is the sum of all the integration complexity in a program ($\Sigma iv(g)$).

and S1 = (S0 – Number of methods + 1).

This is primarily used to determine the number of tests for the "some test" that is designed to ensure that the application would execute without issues in module interaction.

d. **Cyclomatic density (CD):** It is a measure of the amount of logic in the code. It is expressed as follows:

$$CD = \frac{\text{Decisions Made}}{\text{Lines of Executable Code}}$$

By eliminating the size factor, this metric reduces complexity strictly to modules that have unusually dense decision logic. Please note that the higher the CD value, the denser the logic.

The CD metric should be in the range of 0.14 to 0.42 for the code to be simple and comprehensible.

e. **Pathological complexity:** It represents an extremely unstructured code which shows a poor design and hence a suggestion for code's reengineering. A value greater than one indicates poor coding practices like branching into a loop or into a decision. Please note that these conditions are not easy to replicate with modern post 3GL languages.

f. **Branch coverage:** It is a measure of how many branches or decisions in a module have been executed during testing.

If the branch coverage is <95% for new code or 75% for code under maintenance then the test scripts require review and enhancement.

g. **Basis path coverage:** A measure of how many of the basis (cyclomatic, V(G)) paths in a module have been executed. Path coverage is the most fundamental of McCabe design. It indicates how much logic in the code is covered or not covered. This technique requires more thorough testing than branch coverage.

If the path coverage is < 90% for new code or 70% for code under maintenance then the test scripts require review and enhancement.

h. **Boolean coverage:** A technique used to establish that each condition within a decision is shown by execution to independently and correctly affect the outcome of the decision.

The major application of this technique is in safety critical systems and projects.

i. **Combining McCabe metrics:** Cyclomatic complexity is the basic indicator for determining the complexity of logic in a unit of code. It can be combined with other metrics.

1. Code review candidate

If V(G) > 10 and essential complexity/essential density exceeds 4, then the unit needs a review.

2. Code refactoring

If V(G) > 10 and the condition

$$V(G) - EV(g) \leq V(g) \text{ is true}$$

Then, the code is a candidate for refactoring.

3. Inadequate comment content

If the graph between V(G) against comment % (in terms of LOC) does not show a linear increase then the comment content need to be reviewed.

4. Test coverage

If the graph between V(G) against path coverage does not show a linear increase then the test scripts need to be reviewed.

II. OO Metrics

a. **Average V(G) for a class:** If average V(G) > 10 then this metric indicates a high level of logic in the methods of the class which in turn indicates a possible dilution of the original object model. If the average is high, then the class should be reviewed for possible refactoring.

b. **Average essential complexity for a class:** If the average is greater than one then it may indicate a dilution of the original object model.

If the average is high, then the class should be reviewed for possible refactoring.

c. **Number of parents:** If the number of parents for a class is greater than one then it indicates a potentially overly complex inheritance tree.

d. **Response for class (RFC):** RFC is the count of all methods within a class plus the number of methods accessible to an object of this class due to implementation. Please note that the larger the number of methods that can be invoked in response to a message, the greater the difficulty in comprehension and testing of the class. Also, note that low values indicate greater specialization. If the RFC is high then making changes to this class will be increasingly difficult due to the extended impact to other classes (or methods).

e. **Weighted methods for class (WMC):** WMC is the count of methods implemented in a class. It is a strong recommendation that WMC does not exceed the value of 14. This metric is used to show the effort required to rewrite or modify the class. The aim is to keep this metric low.

f. **Coupling between objects (CBO):** It indicates the number of non-inherited classes this class depends on. It shows the degree to which this class can be reused.

For dynamic link libraries (DLLs) this measure is high as the software is deployed as a complete entity.

For executables (.exe), it is low as here reuse is to be encouraged. Please remember this point:

Strong coupling increases the difficulty in comprehending and testing a class. The objective is to keep it less than 6.

g. **Class hierarchy level:** It shows the degree of inheritance used. If it is greater than 6, the increased depth increases the testing effort. If it is less than 2 then the value shows a poor exploitation of OO. So, one should aim for 2 and 3 levels only in our OO-code.

h. **Number of methods (n):** If number of methods (n) > 40 then it shows that the class has too much functionality. So, it can be split into several smaller classes. Please note that ideally one should aim for no more than 20 methods in a class.

i. **Lack of cohesion between methods (LOCM):** It is a metric used to measure the dissimilarity of methods on a class by an instance variable or attribute.

What is to be done?

The percentages of methods in a class using an attribute are averaged and subtracted from 100. This measure is expressed in percentage. Two cases arise:

i. If % is low, it means simplicity and high reusability.

ii. If % is high, it means a class is a candidate for refactoring and could be split into two or more subclasses with low cohesion.

j. **Combined OO metrics:** V(G) can also be used to evaluate OO systems. It is used with OO metrics to find out the suitable candidates for refactoring.

By refactoring, we mean making a small change to the code which improves its design without changing its semantics.

Rules for refactoring:

i. If avg. V(G) > 10 (high) and the number of methods (n) is < 10 (low) then the class requires refactoring.

ii. If avg. V(G) is low and the lack of cohesion is high then the class is a suitable candidate for refactoring into two or more classes.

iii. If avg. V(G) is high and CBO is high then the class is a candidate for refactoring.

iv. If CBO is high and lack of cohesion is high then the class is a candidate for refactoring.

III. Grammar Metrics

a. **Line count:** It is a size indicator. It is used in—

i. Estimation techniques like COCOMO2.

ii. Measuring defects = $\frac{\text{Number of defects}}{\text{1000 LOC}}$

b. **Nesting levels:** Nesting of IF statements, switch, and loop constructs can indicate unnecessarily complex conditions which makes future

modifications quite difficult. So, refactoring may be done. Typical industry standards are 4, 2, and 2 for IF, switch, and loop constructs respectively.

c. **Counts of decision types:** It is used to show single outcome (IF and loop) and multiple outcome decision statements. When used in conjunction with V(G), then its value can determine if a method/procedure/control/section is over complex and, hence, a suitable candidate for refactoring.

d. **Maximum number of predicates:** This measure shows overly complex decision statements which are candidates for refactoring.

e. **Comment lines:** It indicates the level of comments in a unit of code. It shows:

$$\text{Documentation Level (within the code)} = \frac{\text{Comment Lines}}{\text{LOC}} = \frac{\text{Comment Lines}}{\text{V(G)}}$$

Historically, a ratio of 15–25% of comments is adequate to enable any user to understand the code.

5.2.3.2. *Cyclomatic complexity, its properties and its meaning in tabular form*

McCabe's cyclomatic metric, V(G) of a graph G with n vertices and e edges is given by the formula:

$$V(G) = e - n + 2$$

Given a program, we associate it with a directed graph that has unique entry and exit nodes. Each node in the graph, G, corresponds to a block of statements in the program where flow is sequential and the arcs correspond to branches taken in the program. This graph is known as a flow graph.

The cyclomatic complexity, V(G) provides us two things:

1. To find the number of independent paths through the program.
2. To provide an upper bound for the number of test cases that must be executed in order to test the program thoroughly. The complexity measure is defined in terms of independent paths. It is defined as

any path through the program that introduces at least one new set of processing statements or a new condition. See the following steps:

Step 1. Construction of flow graph from the source code or flow charts.
Step 2. Identification of independent paths.
Step 3. Computation of cyclomatic complexity.
Step 4. Test cases are designed.

Using the flow graph, an independent path can be defined as a path in the flow graph that has at least one edge that has not been traversed before in other paths. A set of independent paths that cover all the edges is a basis set. Once the basis set is formed, test cases should be written to execute all the paths in the basis set.

Properties of Cyclomatic Complexity

The following are the properties of cyclomatic complexity represented as V(G):

1. $V(G) \geq 1$.
2. V(G) is the maximum number of independent paths in graph, G.
3. Inserting and deleting functional statements to G does not affect V(G).
4. G has only one path if and only if V(G) = 1. Consider this example Here, V(G) = 1 and that graph, G, has only one path.
5. Inserting a new row in G, increases V(G) by unity.

Meaning of V(G) in Tabular Form

For small programs cyclomatic complexity can be calculated manually, but automated tools are essential as several thousand of lines of code are possible in each program in a project. It will be very difficult to manually create flow graphs for large programs. There are several tools that are available in the market which can compute cyclomatic complexity. Note that calculating the complexity of a module after it has been built and tested may be too late. It may not be possible to redesign a complex module after it has been tested. Thus, some basic complexity checks must be performed on the modules before embarking upon the testing (or even coding) phase. Based on the complexity number that emerges from using the tool, one can conclude what actions need to be taken for complexity measure using the table given below:

TABLE 5.1 What V(G) Means?

Complexity	What it means?
1–10	Well-written code, testability is high, cost/effort to maintain is low.
10–20	Moderately complex, testability is medium, cost/effort to maintain is medium.
20–40	Very complex, testability is low, cost/effort to maintain is high.
> 40	Not testable, any amount of money/effort to maintain may not be enough.

5.2.3.3. BASIS PATH TESTING WITH SOLVED EXAMPLES

Basis path testing helps a tester to compute logical complexity measure, V(G), of the code. This value of V(G), defines the maximum number of test cases to be designed by identifying basis set of execution paths to ensure that all statements are executed at least once. See the following steps:

1. Construct the flow graph from the source code or flow charts.
2. Identify independent paths.
3. Calculate cyclomatic complexity, V(G).
4. Design the test cases.

A program is represented in the form of a flow graph. A flow graph consists of nodes and edges. Using the flow graph, an independent path can be defined as a path in the flow graph that has at least one edge that has not been traversed before in other paths. A set of independent paths that cover all edges is a basis set. Once that basis set is formed, test cases should be written to execute all the paths in the basis set.

Flow Graph Testing

The control flow of a program can be represented using a graphical representation known as a "flow graph." The flow graph is a directed graph in which nodes are either entire statements or fragments of a statement. Edges represent the flow of control. If u and v are nodes in the program graph, there is an edge from node u to node v if the statement (fragment)

corresponding to node v can be executed immediately after the statement (fragment) corresponding to node u,

i.e., u → v.

We next show the basic notations that are used to draw a flow graph:

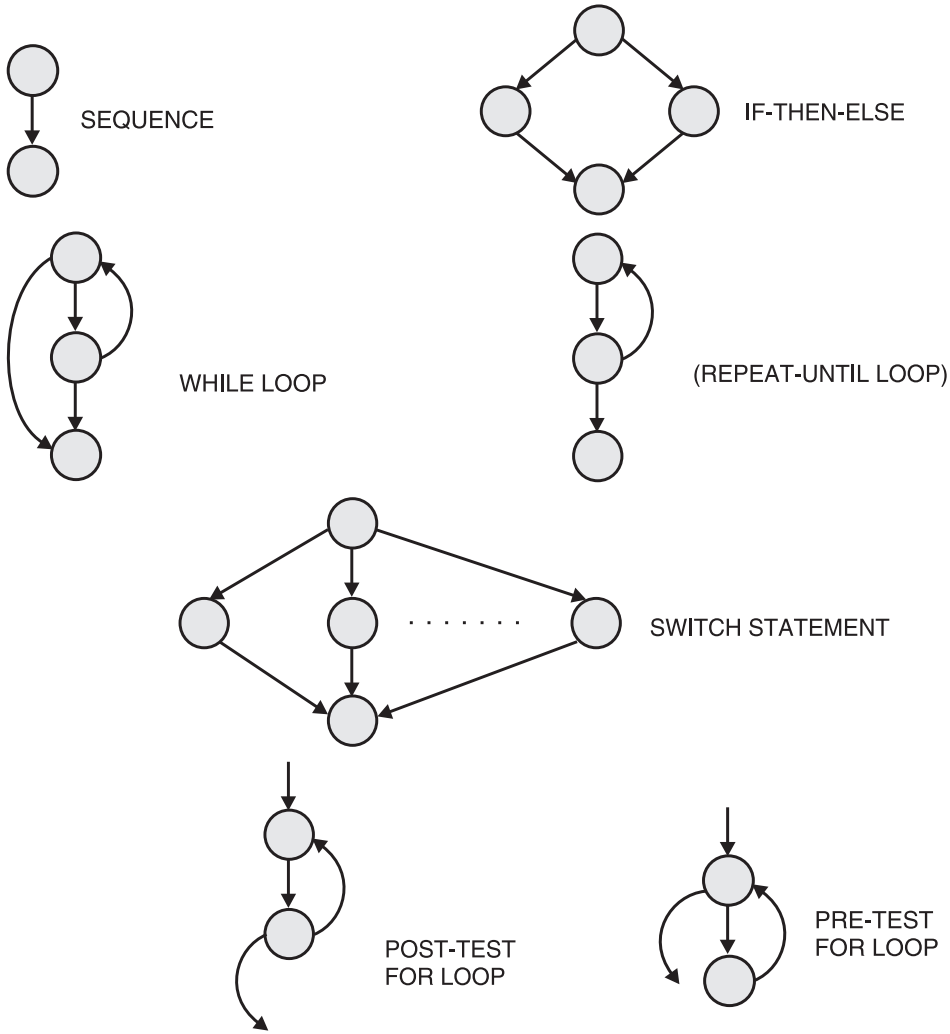


FIGURE 5.3 Notations for Flow Graph.

SOLVED EXAMPLES

EXAMPLE 5.1. Consider the following code:

```
void foo (float y, float a *, int n)
     {
float x = sin (y) ;
if (x > 0.01)
     z = tan (x) ;
else
     z = cos (x) ;
for (int i = 0 ; i < x ; + + i) {
     a[i] = a[i] * z ;
Cout < < a [i] ;
}
```

Draw its flow graph, find its cyclomatic complexity, V(G), and the independent paths.

SOLUTION. First, we try to number the nodes, as follows:

1.
```
void foo (float y, float a *, int n)
{
      float x = sin (y) ;
      if (x > 0.01)
```

2.
```
      z = tan (x) ;
else
```

3.
```
      z = cos (x) ;
```

4.
```
      for (int i = 0; i < n; ++ i)
```

5.
```
      {
```

6.
```
a[i] = a[i] * z ;
cout < < a [i] '
}
```

7.
```
cout < < i ;
}
```

So, its flow graph is shown in Figure 5.4. Next, we try to find V(G) by three methods:

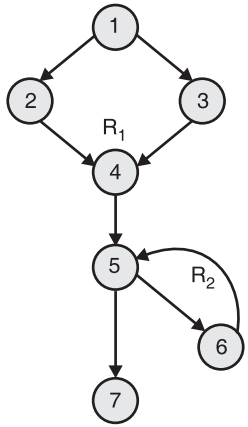


FIGURE 5.4 Flow Graph for Example 5.1.

a. V(G) = e – n + 2 (e – edges, n – nodes)

= 8 – 7 + 2 = 3

b. V(G) = P + 1 (P- predicate nodes without degree = 2)

= 2 + 1 = 3. (Nodes 1 and 5 are predicate nodes)

c. V(G) = Number of enclosed regions + 1

= 2 + 1 = 3

∴ V(G) = 3 and is same by all the three methods.

By V(G) = 3 we mean that it is a well written code, its testability is high, and cost/effort to maintain is low.

Also, it means that there are 3 paths in this program which are independent paths and they form a basis-set. These paths are given below:

Path 1: 1 – 2 – 4 – 5 – 7
Path 2: 1 – 3 – 4 – 5 – 7
Path 3: 1 – 3 – 4 – 5 – 6 – 7

Another basis-set can also be formed:

Path 1: 1 – 2 – 4 – 5 – 7
Path 2: 1 – 3 – 4 – 5 – 7
Path 3: 1 – 2 – 4 – 5 – 6 – 7

This means that we must execute these paths at least once in order to test the program thoroughly. So, test cases can be designed.

EXAMPLE 5.2. Consider the following program that inputs the marks of five subjects of 40 students and outputs average marks and the pass/fail message.

```
a.        # include <stdio.h>
a. (1)    main ( ) {
a. (2)    int num_student, marks, subject, total;
b. (3)    float average ;
e. (4)    num_student = 1;
e. (5)    while (num_student < = 40) {
f. (6)    total = 0 ;
f. (7)    subject = 1;
f. (8)    while (subject < = 5) }
f. (9)    Scanf ("Enter marks: % d", & marks);
f. (10)   total = total + marks ;
f. (11)   subject ++;
f. (12)   }
g. (13)   average = total/5 ;
g. (14)   if (average > = 50)
h. (15)   printf ("Pass... Average marks = % f",
          average);
   (16)       else
i. (17)       print ("FAIL ... Average marks are % f",
              average);
j. (18)       num_student ++;
c. (19)       }
c. (20)       printf ("end of program");
d. (21)       }
```

Draw its flow graph and compute its V(G). Also identify the independent paths.

SOLUTION. The process of constructing the flow graph starts with dividing the program into parts where flow of control has a single entry and exit point. In this program, line numbers 2 to 4 are grouped as one node (marked as "a") only. This is because it consists of declaration and initialization of varia-

bles. The second part comprises of a while loop-outer one, from lines 5 to 19 and the third part is a single printf statement at line number 20.

Note that the second part is again divided into four parts—statements of lines 6 and 7, lines 8 to 12, line 13, and lines 14–17, i.e., if-then-else structure using the flow graph notation, we get this flow graph in Figure 5.5.

Here, "*" indicates that the node is a predicate node, i.e., it has an outdegree of 2.

The statements corresponding to various nodes are given below:

Nodes	Statement Numbers
a	2–4
b	5
e	6–7
f	8
z	9–12
g	13–14
h	15
i	17
j	18
c	19
d	20

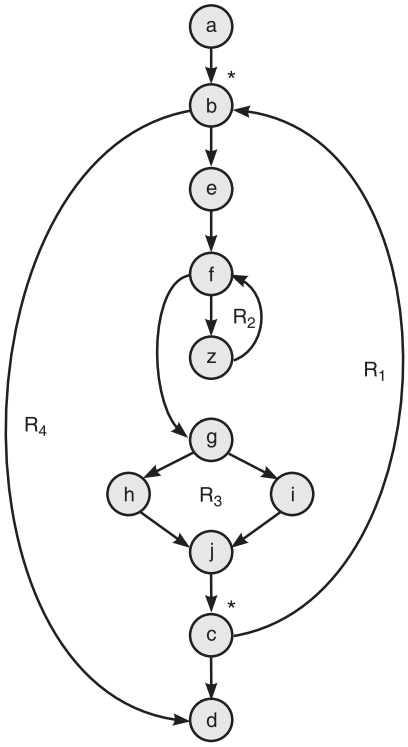


FIGURE 5.5 Flow Graph for Example 5.2.

Next, we compute its cyclomatic complexity, V(G):

a. V(G) = e – n + 2

= 14 – 11 + 2 = 3 + 2 = 5

b. V(G) = Number of predicate nodes (P) + 1

= 4 + 1 = 5

[∵ Nodes b, f, g, and c are predicate nodes with two outgoing edges]

c. V(G) = Number of enclosed regions + 1

= 4 + 1 = 5

[∵ $R_1 - R_4$ are 4 enclosed regions and 1 corresponds to one outer region]

∴ V(G) = 5 by all three methods. Next, we identify a basis-set with five paths:

Path 1: a – b – d – e
Path 2: a – b – d – f – n – b – d – e
Path 3: a – b – c – g – j – k – m – n – b – d – e
Path 4: a – b – c – g – j – l – m – n – b – d – e
Path 5: a – b – c – g – h – i – n – b – d – e

Each of these paths consist of at least one new edge. Please remember that this basis set of paths is NOT unique. Finally, test cases can be designed for the independent path execution that have been identified above. This is to ensure that all statements are executed at least once.

EXAMPLE 5.3. (Quadratic Equation Problem). This program reads a, b, and c as the three coefficients of a quadratic equation $ax^2 + bx + c = 0$. It determines the nature of the roots of this equation. Draw its flow graph and calculate its cyclomatic complexity.

SOLUTION. First, we write its procedure:

```
proc roots
1    int a, b, c;
2    D = b * b - 4 * a * c;
3    if (D < 0)
     4    real = -b/2 * a ;// imaginary roots
          D = - D;
          num = pow ((double) D, (double) 0.5);
          image = num/(2 * a);
5    else if (D = = 0)
     6     root 1 = -b/(2 * a)
           root 2 = root 1;
7    else if (D > 0)
     8     root 1 = (-b + sqrt (d)/2 * a)
           root 2 = (-b - sqrt(d)/2 * a)
9    end
```

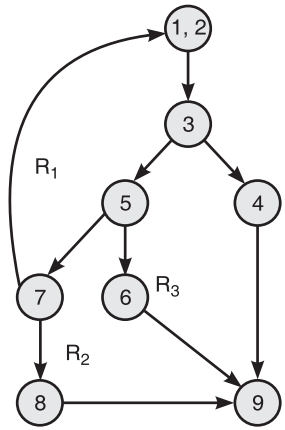


FIGURE 5.6 Flow Graph of Quadratic Equation Problem.

Now, we draw its flow graph as shown in Figure 5.6.

$\therefore$ V(G) = P + 1
= 3 + 1 = 4 [$\because$ Node 3, 5, and 7 are predicate nodes]
V(G) = Number of regions + 1
= 3 + 1 = 4

Also, V(G) = e – n + 2 = 11 – 9 + 2 = 4

$\therefore$ We have 4 independent paths and they are:

Path 1: 1 – 2 – 3 – 4 – 9
Path 2: 1 – 2 – 3 – 5 – 6 – 9
Path 3: 1 – 2 – 3 – 5 – 7 – 8 – 9
Path 4: 1 – 2 – 3 – 5 – 7 – 1 – 2 .....

So, the test cases for each of the path are as follows:

Path 1: test case 1
a, b, c: valid input
expected results: D < 0, imaginary roots

Path 2: test case 2
a, b, c: valid input
expected results: D = 0, equal roots

Path 3: test case 3
a, b, c: valid input
expected results: D > 0, root 1 and root 2 are real

Path 4: test case 4
a, b, c: valid input
expected results: D is not > 0, read a, b, c again

EXAMPLE 5.4. (Triangle Problem): This program reads a, b, c as the three sides of a triangle and determines whether they form an isosceles, equilateral, or scalene triangle. Draw its flow graph and calculate its V(G).

SOLUTION. We draw its flow chart first.

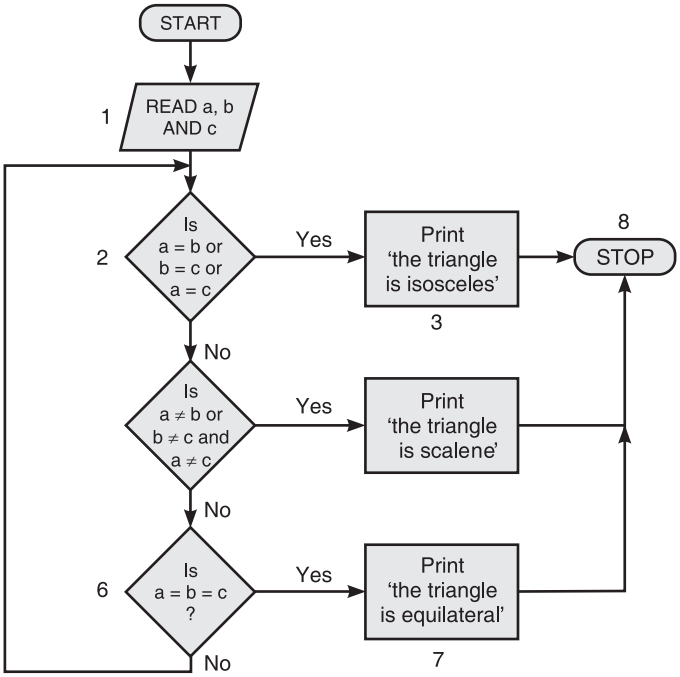


FIGURE 5.7 Flow Chart for Example 5.4.

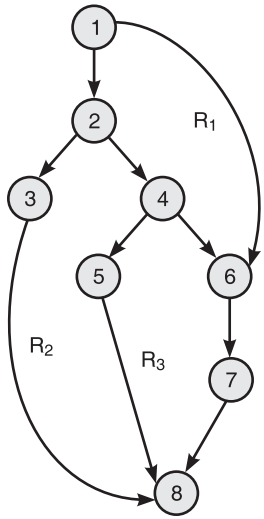


FIGURE 5.8 Flow Graph for the Triangle Problem.

So, we draw its flow graph shown in Figure 5.8.
∴ Its cyclomatic complexity, V(G) by all three methods is

a. V(G) = e – n + 2 = 10 – 8 + 2 = 4

b. V(G) = P + 1 = 3 + 1 = 4 [Nodes 2, 4, 6 are predicate nodes]

c. V(G) = Number of enclosed regions + 1 = 3 + 1 = 4.

∴ 4-test cases need to be designed. They are given below:

```
Path 1: test case 1
    a, b, c: valid input
    Expected results: if a = b or b = c or a = c
             then message 'isosceles triangle' is
             displayed.

Path 2: test case 2
    a, b, c: valid input
    Expected results: if a ≠ b ≠ c then message
    'scalene triangle' is displayed.

Path 3: test case 3
    a, b, c: valid input
    Expected results: if a = b = c then message
    'equilateral triangle' is displayed.
```

```
Path 4: test case 4
    a, b, c: valid float inputs and if a > 0 and b > 0
    and c > 0  Expected results: proper inputs—a, b and c
    and proper results.
```

EXAMPLE 5.5. Consider the following code snippet to search a number using binary search:

```
void search (int key, int n, int a[])
{
     int mid;
1.   int bottom = 0;
2.   int top = n - 1;
3.   while (bottom <= top)
         {
4.       mid = (top + bottom)/2;
5.       if (a[mid]==key)
         {
6.           printf ("\n element is found");
7.           return;
             }
     else
             {
8.           if (a[mid] < key)
9.           bottom = mid + 1;
     else
10.          top = mid - 1;
             }
      }
11.  }
```

a. Draw its flow graph

b. Find V(G) by all 3 methods

c. Derive test cases

SOLUTION. Flow graph is shown in Figure 5.9.

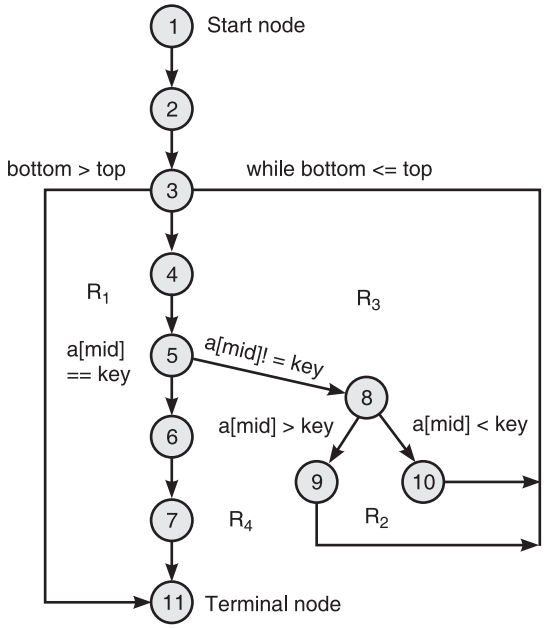


FIGURE 5.9

$\therefore$ V(G) = [Total number of enclosed regions (R_1, R_2, R_3) + 1 outer region where this graph is enclosed]
= 3 + 1 = 4

or V(G) = e – n + 2 = 13 – 11 + 2 = 2 + 2 = 4

or V(G) = P + 1 = 3 + 1 = 4 [$\because$ Nodes 3, 5, and 8 are predicate nodes with outdegree as 2]

Basis set of paths is

Path 1: 1, 2, 3, 4, 5, 6, 7, 11

Path 2: 1, 2, 3, 11

Path 3: 1, 2, 3, 4, 5, 8, 9, 3, …

Path 4: 1, 2, 3, 4, 5, 8, 10, 3, …

$\because$ V(G) = 4

$\Rightarrow$ Paths are also 4 as given above.

$\Rightarrow$ These 4 paths must be executed at least once in order to test this program thoroughly.

$\therefore$ Test cases are:

Test case id	Test case description	Step	Expected results	Actual	Test case studies	Test status (P/F)
1.	Checking bottom and top values of our list	Set bottom = 0 top = n – 1 check if bottom <= top by while loop	Initially bottom <= top is true. true. Lists length gets reduced after every iteration If bottom >= top is reached then return to main		Design	
2.	Checking if middle element of array is equal to key value	set mid = (top + bottom)/2 then compare if a[mid] = key	If a[mid] = key value then print message "even found" and return to main		Design	
3.	If a[mid] < key value	set bottom = mid + 1 and then goto while	search right sublist for key element		Design	
4.	If a[mid] > key value	set top = mid + 1 and go back to while-loop	search left sublist for key element		Design	

EXAMPLE 5.6. Consider the following recursive code to find GCD of two numbers.

```
     GCD (x, y)
     {
0    if x = y
1       ret x;
2       else if x > y
3    GCD (x - y, y);
4       else
5    GCD (x, y - x)
6    }
```

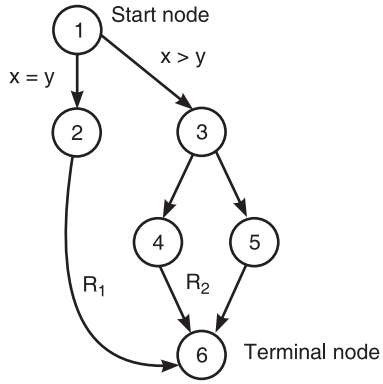


FIGURE 5.10

Draw its flowgraph, find V(G), and basis path set. Derive test cases also.

SOLUTION. The flowgraph for the GCD program is shown in Figure 5.10.

$\therefore$ V(G) = Enclosed regions + 1 = 2 + 1 = 3
V(G) = e – n + 2 = 7 – 6 + 2 = 1 + 2 = 3
= P + 1 = 2 + 1 = 3 ($\because$ Nodes 1 and 3 are predicate nodes)

$\because$ Three independent paths are
1–3–4–6
1–3–5–6
1–2–6

$\because$ Three test cases are:

Test case id	Test case description	Step	Expected results	Actual result	Test case status
1.	x = y	Read 2 nos. such that x = y	GCD = x		
2.	x > y	Read 2 nos. such that x > y	Call recursive routine GCD (x – y, y)		
3.	x < y	Read 2 nos. such that x < y	Call recursive routine GCD (x, y – x)		

NOTES

1. *In test case id–2, we call GCD function recursively with x = x – y and y as it is.*
2. *In test case id–3, we call GCD function recursively with y = y – x and x as it is.*

EXAMPLE 5.7. What is a cyclomatic complexity? Find the following related to given:

```
if(c1){
f1();
}else{
f2();
}
if(c2){
f3();
}else{
f4();
}
```

i. What is a cyclomatic number?

ii. How many test cases are required to achieve complete branch average?

iii. How many test cases are required for complete path coverage?

SOLUTION.

- Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program.
- Cyclomatic complexity has a foundation in graph theory and is computed in one of the three ways:
  - **i.** The number of regions corresponds to the cyclomatic complexity.
  - **ii.** Cyclomatic complexity: $E - N + 2$ (E is the number of edges, and N is number of nodes).
  - **iii.** Cyclomatic complexity: $P + 1$ (P is the number of predicate nodes).

Referring to the flow graph, the cyclomatic number is:

1. The flow graph has three regions.

2. Complexity = 8 edges – 7 nodes + 2 = 3

3. Complexity = 2 predicate nodes + 1 = 3 (Predicate nodes = C_1, C_2)

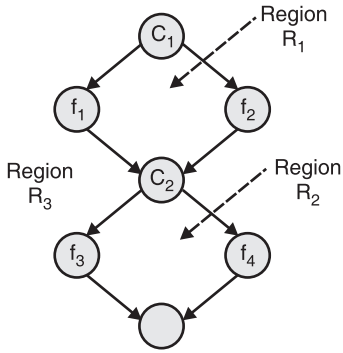


FIGURE 5.11

Two test cases are required for complete branch coverage and four test cases are required for complete path coverage.

Assumptions:

```
c1; if(i%2==0)
f1: EVEN()
f2: ODD()
c2: if(j > 0)
f3: POSITIVE()
f4: NEGATIVE()
i.e.
if(i%2==0){
EVEN();
}else{
ODD();
}
if(j < 0){
POSITIVE();
}else{
NEGATIVE();
}
```

Test cases that satisfy the branch coverage criteria, i.e., <c2, f1, c2, f3> and <c1, f2, c2, f4>.

Sr. No.	Test case: Value of i, j	Decision or Branch	Value of predicate
1.	10, 10	c1 if (i % 2 == 0)	True
		c2 if (j > 0)	True
2.	5, –8	c1 if (i % 2 == 0)	False
		c2 if (j < 0)	False

Test cases that satisfies the path coverage criteria, i.e.,

<c1, f1, c2, f3>
<c1, f2, c2, f4>
<c1, f1, c2, f4>
<c1, f2, c2, f3>

Sr. No.	Test case: Value of i, j	Decision or Branch	Value of predicate
1.	10, 10	c1	True
		c2	True
2.	7, –8	c1	False
		c2	False
3.	10, –8	c1	True
		c2	False
4.	7, 20	c1	False
		c2	True

5.2.3.4. DD PATH TESTING WITH SOLVED EXAMPLES

From a given flow graph, we can easily draw another graph that is known as a decision-to-decision (DD) path graph. Our main concentration now is on the decision nodes only. The nodes of the flow graph are combined into a single node if they are in sequence.

EXAMPLE 5.8. Consider the following pseudo code:

```
0   Premium (age, sex, married)
1        {
2        premium = 500;
3   if ((age < 25) and (sex==male) and (not
    married))
4       premium = premium + 1500;
5   else {
6        if (married or (sex==female))
7            premium = premium-200;
8        if ((age > 45) and (age < 65))
9             premium = premium - 100;
10       }
11   }
```

a. Design its program graph

b. Design its DD path graph

c. List all DD-paths

SOLUTION.

a. Its program graph is as follows:

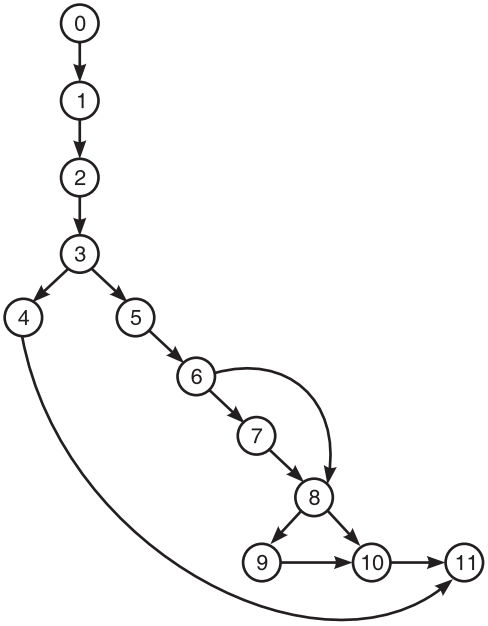


FIGURE 5.12

b. Its DD-path graph is as follows:

Nodes	Lines
A	0, 1, 2
B, C, D	3, 4, 5
E	6
F	7
G, H, I	8, 9, 10
J	11

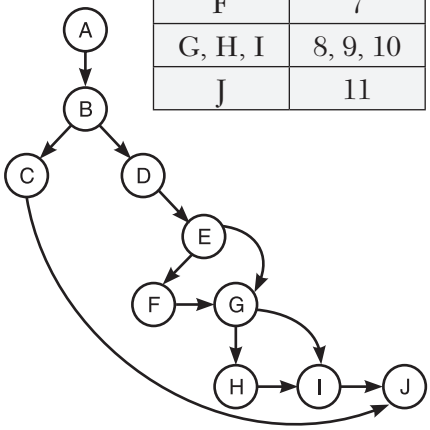


FIGURE 5.13

c. The DD-paths are as follows:

Path–1: A–B–C–J
Path–2: A–B–D–E–G–I–J
Path–3: A–B–D–E–F–G–H–I–J
Path–4: A–B–D–E–G–H–I–J

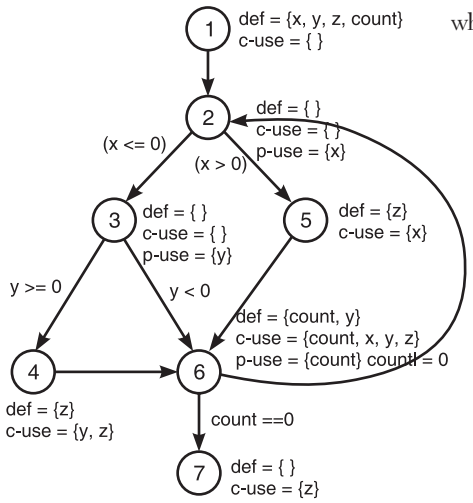
EXAMPLE 5.9. Consider the following code snippet:

```
1   begin
2         float x, y, z = 0.0;
3         int count;
4         input (x, y, count);
5         do {
```

```
6            if (x ≤ 0) {
7            if(y ≥ 0){
8                 z = y*z + 1;
9            }
10       }
11       else {
12           z = 1/x;
13       }
14       y = x * y + z
15       count = count - 1
16       while (count > 0)
17       output (z);
18       end
```

Draw its data-flow graph. Find out whether paths (1, 2, 5, 6) and (6, 2, 5, 6) are def-clear or not. Find all def-use pairs?

SOLUTION. We draw its data flow graph (DD-path graph) first.



where

Nodes	Lines
1	1, 2, 3, 4
2	5, 6
3	7
4	8, 9, 10
5	11, 12, 13
6	14, 15, 16
7	17, 18

FIGURE 5.14

Now, path (1, 2, 5, 6) is definition clear (def-clear) with respect to definition of x at node-1, i.e., d1(x).

So, $d_1(x)$ is live at node 6, i.e., x is being used at node-6 again. But path (1, 2, 5, 6) is not def-clear with respect to $d_1(z)$ because of $d_5(z)$.

Now, consider path (6, 2, 5, 6). It is def-clear with respect to d_6 (count). Variables y and z are used at node 4. Please note that we can easily check for definitions $d_1(y)$, $d_6(y)$, $d_1(z)$, and $d_5(z)$ that they are live at node 4.

What about def-use pairs?

We have 2 types of def-use pairs:

1. That correspond to a definition and its c-use, (dcu). ("c" stands for computational.)
2. That correspond to a definition and its p-use, (dpu). ("p" stands for predicate condition.)

Let us then compute dcu (x, 1), i.e., the set of all c-uses associated with the definition of x at node-1. Also note that there is a c-use of x at node-5 and a def-clear path (1, 2, 5) from node-1 to node-5. So, node-5 is included in dcu (x, 1). Similarly, we can include node-6 also in this set. So, we get:

$$\text{dcu}(x, 1) = \{5, 6\}$$

What is dpu (x, 1)?

As shown in program, there is a p-use of x at node-2 with outgoing edges (2, 3) and (2, 5). The existence of def-clear paths from node-1 to each of these two edges is easily seen in DD-path graph. There is no other p-use of x. So,

$$\text{dpu}(x, 1) = \{(2, 3), (2, 5)\}$$

We can now compute for all other variables as shown below:

Variable	Defined at node (n)	dcu (v, n)	dpu (v, n)
x	1	{5, 6}	{(2, 3), (2, 5)}
y	1	{4, 6}	{(3, 4), (3, 6)}
y	6	{4, 6}	{(3, 4), (3, 6)}
z	1	{4, 6, 7}	{ }
z	4	{4, 6, 7}	{ }
z	5	{4, 6, 7}	{ }
count	1	{6}	{(6, 2), (6, 7)}
count	6	{6}	{(6, 2), (6, 7)}

The spots that are not def-clear are trouble spots.

5.2.3.5. GRAPH MATRICES TECHNIQUE WITH SOLVED EXAMPLES

In graph matrix based testing, we convert our flow graph into a square matrix with one row and one column for every node in the graph. The objective is to trace all links of the graph at least once. The aim is again to find cyclomatic complexity, V(G) and, hence, the independent paths. If the size of graph increases, it becomes difficult to do path tracing manually.

For example, Consider the following flow graph:

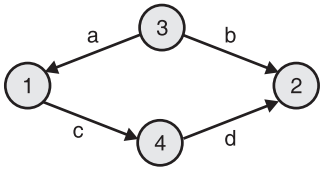


FIGURE 5.15 Flow Graph Example.

Its graph-matrix is:

	1	2	3	4
1			a	c
2				
3		b		
4		d		

FIGURE 5.16 Graph Matrix.

Now, consider another flow graph:

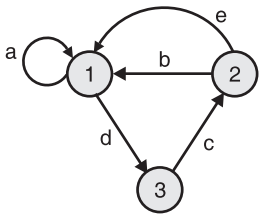


FIGURE 5.17 Flow Graph Example.

∴ Its graph matrix is:

	1	2	3
1	a		d
2	b+e		
3		c	

FIGURE 5.18 Graph Matrix.

Note that if there are several links between two nodes then "+" sign denotes a parallel link.

This is, however, not very useful. So, we assign a weight to each entry of the graph matrix. We use "1" to denote that the edge is present and "0" to show its absence. Such a matrix is known as a *connection matrix*. For the Figure above, we will have the following connection matrix:

	1	2	3	4
1			1	1
2				
3		1		
4		1		

FIGURE 5.19 Connection Matrix.

	1	2	3	4	
1			1	1	2 – 1 = 1
2					
3		1			1 – 1 = 0
4		1			1 – 1 = 0
					1 + 1 = 2 = V(G)

FIGURE 5.20

Now, we want to compute its V(G). For this, we draw the matrix again, i.e., we sum each row of the above matrix. Then, we subtract 1 from each row. We, then, add this result or column of result and add 1 to it. This gives us V(G). For the above matrix, V(G) = 2.

EXAMPLE 5.10. Consider the following flow graph:

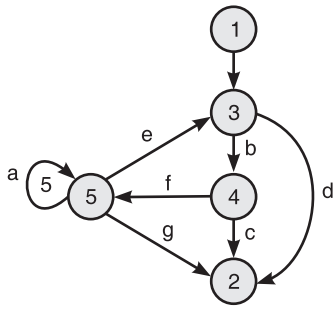


FIGURE 5.21

Draw its graph and connection matrix and find V(G).

SOLUTION.

First, we draw its graph matrix:

	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		h

FIGURE 5.22

Now, its connection matrix is:

	1	2	3	4	5	
1			1			1 – 1 = 0
2						
3		1		1		2 – 1 = 1
4		1			1	2 – 1 = 1
5	0	1	1		1	3 – 1 = 2
						4 + 1 = 5 = V(G)

FIGURE 5.23

Similarly, we can find two link or three link path matrices, i.e., A^2, A^3, ... A^{n-1}. These operations are easy to program and can be used as a testing tool.

5.2.3.6. DATA FLOW TESTING WITH EXAMPLES

Data flow testing uses the sequence of variable access to select paths from a control graph. The main focus of this structural or white-box testing technique is on the usage of variables. With a program data definition faults are nearly as frequent (22% of all faults) as control flow faults (24 percent). Some common data related faults include:

- Using an undefined or uninitialized variable.
- Unreachable code segments.
- Assigning a value to a variable more than once without an intermediate reference.
- Deallocating or re-initializing a variable before its last use.

It is another form of white-box testing. Here the focal point is on the usage of variables. Because data variables and data structures are an important part of any software, so we must check for variables— which have not been defined but used or variables which are defined but not used or a variable that is defined twice. These may be the main culprits or sources of errors in our program. We need to identify them and remove the problems associated with them.

Some terminology used during data flow testing is given below:

1. **Defining node:** A defining node, n, of the variable, v, if the value of the variable, v, is defined at this node, n.

   For example, a = b + c
   ↑
   This is defining node for "a."

2. **Usage node:** A usage node, n, of the variable, v, if the value of the variable, v, is used at this node, n.

   For example, a = b + c
   ↑
   This is the usage node for "b" and "c."

3. **Definition use path (du-path):** The path from a node, m, to node, n, where the variable, v, is initially defined to the point where it is used in the program is called as du-path.

4. **Definition clear path (dc-path):** The path from node, m, to node, n, such that there is no other node in this path that redefines the variable, v.

   Please note that our objective is to find all du-paths and then identify those du-paths which are not dc-paths. We generate test cases for du-paths that are not dc-paths.
   Steps followed for data flow testing are given below:

   Step 1: Draw the program flow graph.
   Step 2: Prepare a table for define/use status of all variables used in your program.
   Step 3: Find all du-paths.
   Step 4: Identify du-paths that are not dc-paths.

   This is data flow testing.

EXAMPLE 5.11. For the following C-like program, give the

1. flow graph.
2. def/use graph.

```
scanf(x, y);
if (y < 0)
    pow = 0 - y;
else
    pow = y;
z = 1.0;
while (pow!=0)
     {
     z = z * x;
     pow = pow-1;
     }
     if (y < 0)
         z = 1.0/z;
     printf(z);
```

SOLUTION.

Let us rewrite the program again for easy drawing of its flow graph.

```
1     scanf(x, y); if (y < 0)
2     pow = 0 - y;
3     else pow = y;
4     z = 1.0;
5     while (pow! = 0)
```

```
6      {z = z*x; pow = pow - 1;}
7      if (y < 0)
8      z = 1.0/z;
9      printf (z);
```

i. The flow graph of a given program is as follows:

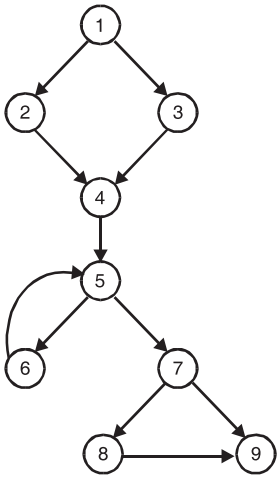


FIGURE 5.24

ii. The def/use graph for this program is as follows:

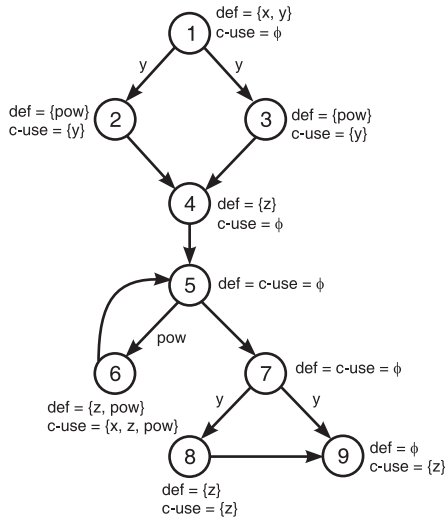


FIGURE 5.25

Now, let us find its dcu and dpu. We draw a table again as follows:

(Node, var)	dcu	dpu
(1, x)	{6}	ϕ
(1, y)	{2, 3}	{(1, 2), (1, 3), (7, 8), (7, 9)}
(2, pow)	{6}	{(5, 6), (5, 7)}
(3, pow)	{6}	{(5, 6), (5, 7)}
(4, z)	{6, 8, 9}	ϕ
(6, z)	{6, 8, 9}	ϕ
(6, pow)	{6}	{(5, 6), (5, 7)}
(8, z)	{9}	ϕ

EXAMPLE 5.12. (The Commission Problem). Consider again the commission problem. The problem domain has already been discussed in Chapter 2. Write its program, draw its flow graph and DD-path graph. Find its V(G) and, hence, independent paths. Also, perform its data flow testing.

SOLUTION. We develop its program first:

```
1  Program commission (Input, Output)
2  Dim locks, stocks, barrels As Integer
3  Dim lockprice, stockprice, barrel price
   As real
4  Dim totallocks, totalstocks, total barrels
   As integer
5  Dim locksales, stocksales, barrelsales As
   real
6  Dim sales, commission As real
7  lockprice = 45.0
8  Stockprice = 30.0
9  barrelprice = 25.0
10 totallocks = 0
11 totalstocks = 0
12 total barrels = 0
13 Input (locks)
14 while NOT clocks = -1)
15       Input (stock, barrels)
16       totallocks = totallocks + locks
17       totalstocks = totalstocks + stocks
18       totalbarrels = totalbarrels + barrels
19       Input (locks)
20 Endwhile
21       Output (“Locks sold:”, total locks)
22       Output (“Stocks sold:”, total stocks)
23       Output (“Berrels sold:”, total
         barrels)
24       locksales = lockprice * totallocks
25       stocksales = stockprice * totalstocks
26       barrelsales = barrelprice *
         totalbarrels
27       sales = locksales + stocksales +
         barrelsales
28       Output (“Total sales:”, sales)
29       If (sales > 1800.0)
```

```
30        Then
31             commission = 0.10 * 1000.0
32             commission = commission + 0.15 *
               800
33             commission = commission + 0.20 *
               (sales - 1800)
34        Else if (sales > 100.0)
35        then
36             commission = 0.10 * 1000.0
37             commission = commission + 0.15 *
               (sales - 1000.0)
38             else commission = 0.10 * sales
39             Endif
40        Endif
41             Output ("Commission is $",
               commission)
42             End commission
```

Now, we draw its flow graph first:

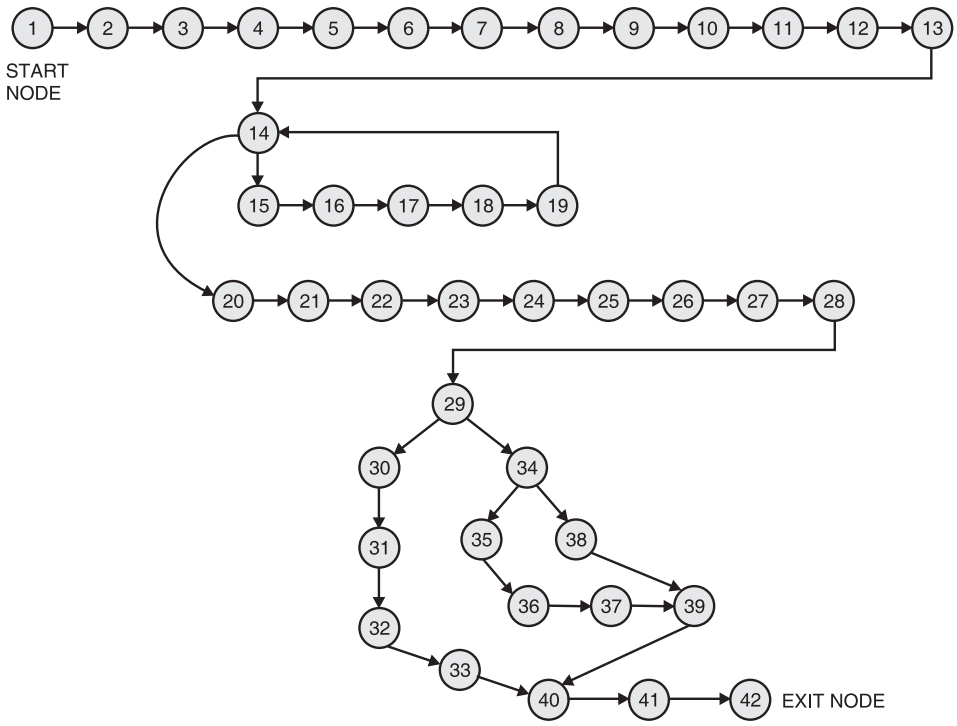


FIGURE 5.26 Flow Graph of Commission Problem.

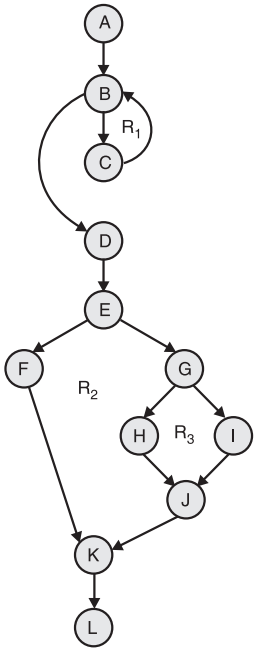


FIGURE 5.27 DD Path Graph.

Next, we draw its DD path graph:

Note that more compression exists in this DD-path graph because of the increased computation in the commission problem.

The table below details the statement fragments associated with DD-paths.

DD-path	Nodes
A	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13
B	14
C	15, 16, 17, 18, 19
D	20, 21, 22, 23, 24, 25, 26, 27, 28
E	29
F	30, 31, 32, 33
G	34
H	35, 36, 37
I	38
J	39
K	40
L	41, 42

∴ We now compute its V(G):

a. V(G) = Number of enclosed regions + 1 = 3 + 1 = 4

b. V(G) = e – n + 2 = 14 – 12 + 2 = 2 + 2 = 4

c. V(G) = P + 1 = 3 + 1 = 4 [∵ Nodes B, E, and G are predicate nodes]

∴ There must exist four independent paths. They are given below.

Path 1: A – B – C – B – ...
Path 2: A – B – D – E – F – K – L
Path 3: A – B – D – E – G – H – J – K – L
Path 4: A – B – D – E – G – I – J – K – L

Next, we prepare the define/use nodes for variables in the commission problem.

Variables	Defined at node	Used at node
lockprice	7	24
stockprice	8	25
barrelprice	9	26
total locks	10, 16	16, 21, 24
total stocks	11, 17	17, 22, 25
total barrels	12, 18	18, 23, 26
locks	13, 19	14, 16
stocks	15	17
barrels	15	18
lock sales	24	27
stock sales	25	27
barrel sales	26	27
sales	27	28, 29, 33, 34, 37, 38
commission	31, 32, 33, 36, 37, 38	32, 33, 37, 41

Next, to find du-paths, we prepare the table:

Variable	Path (beginning, end) nodes	Definition clear
lockprice	7, 24	yes
stockprice	8, 25	yes
barrelprice	9, 26	yes
total stocks	11, 17	yes
total stocks	11, 22	no
total stocks	17, 25	no
total stocks	17, 17	yes
total stocks	17, 22	no
total stocks	17, 25	no
locks	13, 14	yes
locks	19,14	yes
locks	13, 16	yes
locks	19, 16	yes

(*Continued*)

Variable	Path (beginning, end) nodes	Definition clear
sales	27, 28	yes
sales	27, 29	yes
sales	27, 33	yes
sales	27, 34	yes
sales	27, 37	yes
sales	27, 38	yes
commission	31, 32	yes
commission	31, 33	no
commission	31, 37	not possible
commission	31, 41	no
commission	32, 32	yes
commission	32, 33	yes
commission	32, 37	not possible
commission	32, 41	no
commission	33, 32	not possible
commission	33, 33	yes
commission	33, 37	not possible
commission	33, 41	yes
commission	36, 32	not possible
commission	36, 33	not possible
commission	36, 37	yes
commission	36, 41	no
commission	37, 32	not possible
commission	37, 33	not possible
commission	37, 37	yes
commission	37, 41	yes
commission	38, 32	not possible
commission	38, 33	not possible
commission	38, 37	not possible
commission	38, 41	yes

The initial value definition for the variable, "titalstocks" occurs at node-11 and it is first used at node-17. Thus, the path (11, 17) which consists of the node sequence <11, 12, 13, 14, 15, 16, 17>, is definition clear.

The path (11, 22), which consists of the node sequence <11, 12, 13, 14, 15, 16, 17, 18, 19, 20> * & <21, 22> is not definition clear because the values of total stocks are defined at node-11 and at node-17. The asterisk, *, is used to denote zero or more repetitions.

Thus, out of 43 du-paths, 8-paths, namely, (11, 22), (17, 25), (17, 22), (17, 25), (31, 33), (31, 41), (32, 41), and (36, 41), are not definition clear. These 8-paths are the main culprits and thus the cause of the error.

Problem for Practice

1. Consider the problem of finding out the roots of a given quadratic equation, $ax^2 + bx + c = 0$.

   Write its program in C and perform the following:

   a. Draw the flow graph.

   b. Draw the DD path graph.

   c. Find V(G) by the three methods.

   d. Perform its data flow testing and find all du- and dc-paths.

How to Develop Data Flow Test Cases?

The steps that are usually followed to develop data flow test cases are as follows:

1. Layout a control graph for the program.
2. For each variable, note the defined (d), used (u), and killed (k) actions by node.
3. Trace all possible du- and dk-paths for each variable.
4. Merge the paths.
5. Test/check each path for testability.
6. Select test data for feasible paths.

We shall now apply these steps to the following code:

```
1.  void MyClass :: xyz (int x, int y, int z)
         {
         if (x>=10) {
2.  x = x + 2;
         y = y - 4;
         if (x > z) {
3.       y = x - z;
         else
4.  y = 7;
         }
         else {
5.  y = z + x;
    }
6.  while (x > y) {
7.  y = y + 1;
        z = z - y;
        }
8.  if (x > z)
9.      cout << x<<z;
10. cout << x<<y;
    }
```

Step 1: Its flow graph is:

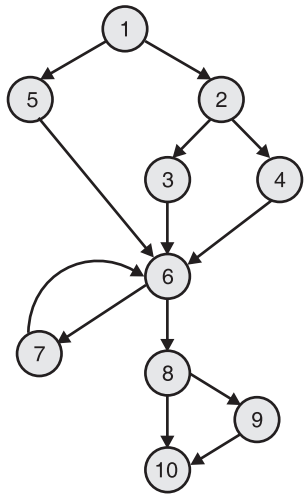


FIGURE 5.28 Flow Graph for Example.

Step 2: Next, we tabulate data actions. The D and U actions in xyz-class can be read from the code.

Block	Source code	x	y	z

1.	MyClass:: xyz
	(int x; int y; int z) {
	if (x >= 10) {
		
		D
		U
			
			D
				
				D

2.	x = x + 2;
	y = y – 4;
	if (x > z) {
		UD
		
		U
			
			UD
				
				U

3.	y = x – z; else	U	D	U

4.	y = 7;
	}
	else
			D	

5.	y = z + x; }	U	D	U
6.	while (x > y) {	U		U

7.	y = y + 1;
	z = z – y;
	}
			UD
			U
				
				UD

8.	if (x > z)	U		U
9.	cout << x, z;	U		U
10.	cout << x, y; }	U	U	

Step 3: Trace data flow for X

There will be a set of paths for each d. Mark the d nodes for the variable on the graph and trace the paths. The baseline trace method can be adapted for this tracing. Tabulate the paths as you trace them. So, define/use paths for "x" are shown in Figure 5.29.

Note that the variable-x has d actions at nodes-1 and -2.

∴ Paths from node-1:

1–5–6–8–9–10	DU–
1–5–6–8–10	DU–
1–2	DUD

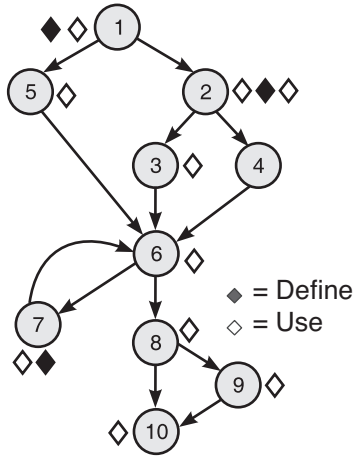


FIGURE 5.29 Trace Data Flow Graph for "X."

And paths from node-2:

2–3–6–8–9–10	DU–
2–4–6–8–9–10	DU–
2–3–6–8–10	DU–

Similarly, trace data flow for y is as follows: Define/Use paths for y are shown in Figure 5.30. Variable "y" has d actions are nodes 1, 2, 3, 4, 5, 7.

Paths from node-1:

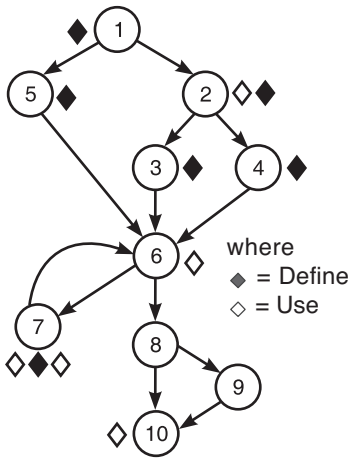


FIGURE 5.30 Trace Data Flow for "y."

1–5	DUD
1–2	DUD

Paths from node-2:

2–3	DD
2–4	DD

Paths from node-3:

3–6–7	DUD
3–6–8–10	DU–

Paths from node-4:

4–6–7	DUD
4–6–8–10	DU–

Paths from node-5:

5–6–7	DUD
5–6–8–10	DU–

Paths from node-7:

5–6–7	DUD
5–6–8–10	DU–

Similarly, the trace data flow for "Z" is as follows: Define/Use Paths for Z are shown in Figure 5.31. The variable Z has d actions at nodes-1 and-7.

∴ Paths from node-1 are:

1–5–6–7	DUD
1–5–6–8–9	DU–
1–5–6–8	DU–
1–2–3–6–7	DUD
1–2–4–6–7	DUD
1–2–3–6–8–9	DU–
1–2–3–6–8	DU–

Paths from node-7:

7–6–8–9	DU–
7–6–7	DU–
7–6–8	DU–

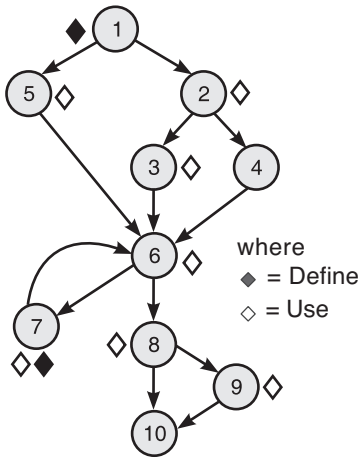


FIGURE 5.31 Trace Data Flow for "Z."

Step 4: Merge the paths

a. **Drop sub-paths:** Many of the tabulated paths are sub-paths. For example,

{1 2 3 6 8} is a sub-path of {1 2 3 6 8 9}.

So, we drop the sub-path.

b. **Connect linkable paths:** Paths that end and start on the same node can be linked. For example,

{1 5} {5 6 8 10}
becomes {1 5 6 8 10}

We cannot merge paths with mutually exclusive decisions. For example,

{1 5 6 8 9} and {1 2 3 6 7}

cannot be merged because they represent the predicate branches from node-1.
So, merging all the traced paths provides the following set of paths:

Path	Nodes visited
1	1–5–6–8–10
2	1–5–6–8–9–10
3	1–5–6 (7 6)* 8 10
4	1–5–6 (7 6)* 8 9 10
5	1–2–3–6–8–10
6	1–2–3–6–8–9–10
7	1–2–3–6 (7 6)* 8–10
8	1–2–3–6 (7 6)* 8–9–10
9	1–2–4–6–8–10
10	1–2–4–6–8–9–10
11	1–2–4–6 (7 6)* 8–10
12	1–2–4–6 (7 6)* 8–9–10

The (7 6)* means that path 7–6 (the loop) can be iterated. We require that a loop is iterated at least twice.

Step 5: Check testability

We need to check path feasibility.
Try path 1: {1–5–6–8–10}

Condition	Comment
1. $x \le 10$	Force branch to node-5
2. $y' = z + x$	Calculation on node-5
3. $x \le z + x$	Skip node-7, proceed to node-8
4. $x \le z$	Skip node-9, proceed to node-10

The values that satisfy all these constraints can be found by trial and error, graphing, or solving the resulting system of linear inequalities.

The set x = 0, y = 0, z = 0 works, so this path is feasible.
Try path 5: {1–2–3–6–8–10}

Condition	Comment
1. $x > 10$	Force branch to node-2
2. $x' = x + 2$	Calculation on node-2
3. $y' = y - 4$	Calculation on node-2
4. $x' > z$	Force branch to node-3
5. $x' \leq y'$	Skip node-7, proceed to node-8
6. $x' \leq z$	Skip node-9, proceed to node-10

By inspection, conditions 4 and 6 cannot both be true, so path 5 is dropped.

Step 6: Select test data

Path feasibility analysis also provides domain information for each path. This can be used in conjunction with other domain constraints (from requirements, for example) to develop test data for each path.

TEST DATA

Path	Nodes visited	Input test data			Output	
		x	y	z	B9	B10
1.	1–5–6–8–10	0	0	0	–	0
		9	0	9	–	18
2.	1–5–6–8–9–10	9	0	8	17	25
		9	0	0	9	18
3.	1–5–6–(7 6)* 8–10	–1	0	–1	–	–2
4.	1–5–6–(7–6)* 8–9–10	0	0	1	–1	–1
		9	0	–1	–1	18
7.	1–2–3–6–(7–6)* 8–10	10	0	62	–	24
8.	1–2–3–4–(7–6)*8–9–10	10	0	12	–26	24
		10	0	61	23	24
10.	1–2–3–6–8–9–10	10	0	0	12	24
12.	1–2–3–6–(7–6)*–8–9–10	10	0	1	1	24

5.3. MUTATION TESTING VERSUS ERROR SEEDING — DIFFERENCES IN TABULAR FORM

In the error-seeding technique, a predefined number of artificially generated errors is "sown" in the program code. After that, test runs are used to detect errors and to examine the ratio between actual and artificial errors based on the total number of detected errors. The artificially generated errors are not known to the testers. Error seeding and mutation testing are both error-oriented techniques and are generally, applicable to all levels of testing. We shall now tabulate the differences between these two techniques of white-box testing. They are as follows:

Error seeding		Mutation testing	
1.	No mutants are present here.	1.	Mutants are developed for testing.
2.	Here source code is tested within itself.	2.	Here mutants are combined, compared for testing to find errors introduced.
3.	Errors are introduced directly.	3.	Special techniques are used to introduce errors.
4.	Test cases which detect errors are used for testing.	4.	Here, test cases which kill mutants are used for testing.
5.	It is a less efficient error testing technique.	5.	It is more efficient than error seeding.
6.	It requires less time.	6.	It is more time consuming.
7.	It is economical to perform.	7.	It is expensive to perform.
8.	It is a better method for bigger problems.	8.	It is a better method for small size programs.

This mutation testing scheme was proposed by DeMillo in 1978. In this testing technique, we mutate (change) certain statements in the source code and check if the test code is able to find the errors. It is a technique that is used to assess the quality of the test cases, i.e., whether they can reveal certain types of faults.

For example, SDLC is viewed as a "root-finding" procedure. That is, a designer submits an initial guess which is then iteratively tested for validity until a correct solution is obtained. This is the *principle of meta-induction.*

These mutants are run with an input data from a given test set. If a test set can distinguish a mutant from the original program, i.e., it produces a different execution result, the mutant is said to be killed. Otherwise, the mutant is called as a live mutant. Please note that a mutant remains live because it is equivalent to the original program, i.e., it is functionally identical to the original program or the test data is inadequate to kill the mutant. If a test data is inadequate, it can be improved by adding test cases to kill the live mutant. A test set which can kill all non equivalent mutants is said to be adequate (mutation score). Now, the adequacy of a test set is measured as follows:

$$\text{Adequacy of Test Set} = \frac{\text{No. of Killed Mutants}}{\text{No. of Non equivalent Mutants}}$$

Advantages of Mutation Testing

1. It can show the ambiguities in code.
2. It leads to a more reliable product.
3. A comprehensive testing can be done.

Disadvantages of Mutation Testing

1. It is difficult to identify and kill equivalent mutants.
2. Stubborn mutants are difficult to kill.
3. It is a time consuming technique, so automated tools are required.
4. Each mutation will have the same size as that of the original program. So, a large number of mutant programs may need to be tested against the candidate test suite.

5.4. COMPARISON OF BLACK-BOX AND WHITE-BOX TESTING IN TABULAR FORM

Functional or black-box testing	Structural or white-box testing or glass-box testing
1. This method focus on *functional requirements* of the software, i.e., it enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program.	**1.** This method focuses on *procedural details,* i.e., internal logic of a program.
2. It is NOT an alternative approach to white-box technique rather is a complementary approach that is likely to uncover a different class of errors.	**2.** It concentrates on internal logic, mainly.
3. Black-box testing is applied during *later stages* of testing.	**3.** Whereas, white-box testing is performed *early* in the testing process.
4. It attempts to find errors in the following categories: **a.** Incorrect or missing functions **b.** Interface errors **c.** Errors in data structures or external database access **d.** Performance errors **e.** Initialization and termination errors	**4.** Whereas, white-box testing attempts errors in following cases: **a.** Internal logic of your program **b.** Status of program
5. It disregards *control structure* of procedural design (i.e., what is the control structure of our program, we do not consider here).	**5.** It *uses control structure* of the procedural design to derive test cases.

Functional or black-box testing	Structural or white-box testing or glass-box testing
6. Black-box testing broadens our focus on the information domain and might be called as "testing in the large," i.e., *testing bigger monolithic programs.*	**6.** White-box testing, as described by *Hetzel* is *"testing in small,"* i.e., testing small program components (e.g., modules or small group of modules).
7. Using black-box testing techniques, we derive a set of test cases that satisfy following criteria: **a.** Test cases that reduce (by a count that is greater than 1) the number of additional test cases that must be designed to achieve reasonable testing. **b.** Test cases that tell us something about the presence or absence of classes of errors rather than an error associated only with the specific tests at hand.	**7.** Using white-box testing, the software engineer can desire test cases that: **a.** guarantee that all independent paths within a module have been exercised at least once. **b.** exercise all logical decisions on their true and false sides. **c.** execute all loops at their boundaries and within their operational bounds. **d.** exercise internal data structures to ensure their validity.
8. It includes the tests that are conducted at the software interface.	**8.** A close examination of procedural detail is done.
9. Are used to uncover errors.	**9.** Logical paths through the software are tested by providing test cases, that exercise specific sets of conditions or loops.
10. To demonstrate that software functions are operational, i.e., input is properly accepted and output is correctly produced. Also, the integrity of external information (e.g., a data base) is maintained.	**10.** A limited set of logical paths be examined.

5.5. PRACTICAL CHALLENGES IN WHITE-BOX TESTING

The major challenges that we face during white-box testing are as follows:

1. Difficult for Software Developer to Pin-Point Defects From His Own Creations: As discussed earlier, no one would like to point out errors from their own creations. So, does a developer. So, usually we select a different test team.
2. Even a Completely Tested Code May Not Satisfy Real Customer Requirements: Developers do not have a full appreciation and favor to external customer's requirements or the domain knowledge. This means that even after thorough verification and validation, common user scenarios may get left out. So, we must address such scenarios.

5.6. COMPARISON ON VARIOUS WHITE-BOX TESTING TECHNIQUES

Functional testing techniques always result in a set of test cases and structural metrics are always expressed in terms of something countable like the number of program paths, the number of decision-to-decision paths (DD-paths), and so on.

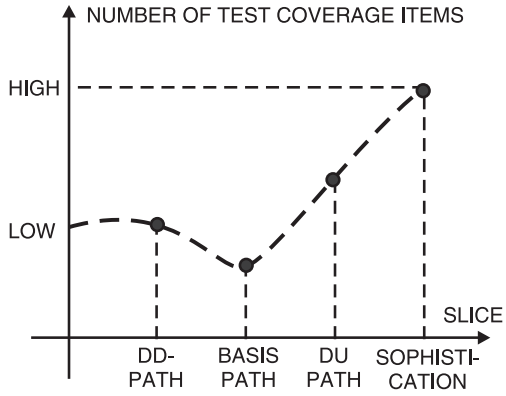


FIGURE 5.32 Shows Trends of Test Coverage Item(s).

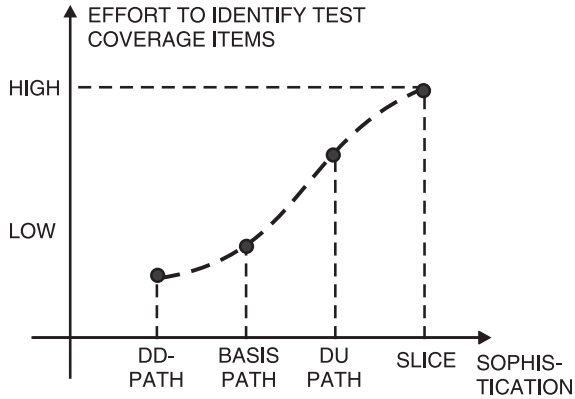


FIGURE 5.33 Shows Trend of Test Method Effort.

Figures 5.32 and 5.33 show the trends for the number of test coverage items and the effort to identify them as functions of structural testing methods, respectively. These graphs illustrate the importance of choosing an appropriate structural coverage metric.

5.7. ADVANTAGES OF WHITE-BOX TESTING

1. White-box testing helps us to identify memory leaks. When we allocate memory using malloc( ) in C, we should explicitly release that memory also. If this is not done then over time there would be no memory available for allocating memory on requests. This can be done using debuggers that can also tally allocated and freed memory.
2. Performance analysis: Code coverage tests can identify the areas of a code that are executed most frequently. Extra efforts can then be made to check these sections of code. To do further performance improvement techniques like caching, coprocessing or even parallel processing can be considered.
3. Coverage tests with instrumented code is one of the best means of identifying any violations of such concurrency constraints through critical sections.
4. White-box testing is useful in identifying bottlenecks in resource usage. For example, if a particular resource like RAM or ROM or even network

is perceived as a bottleneck then instrumented code can help identify where the bottlenecks are and point towards possible solutions.

5. White-box testing can help identify security holes in dynamically generated code. For example, in case of Java, some intermediate code may also be generated. Testing this intermediate code requires code knowledge. This is done by white-box testing only.

SUMMARY

1. White-box testing can cover the following issues:
   - **a.** Memory leaks
   - **b.** Uninitialized memory
   - **c.** Garbage collection issues (in JAVA)
2. We must know about white-box testing tools also. They are listed below:
   - **a.** Purify by Rational Software Corporation
   - **b.** Insure++ by ParaSoft Corporation
   - **c.** Quantify by Rational Software Corporation
   - **d.** Expeditor by OneRealm Inc.

MULTIPLE CHOICE QUESTIONS

1. A testing which checks the internal logic of the program is
   - **a.** Black-box testing.
   - **b.** White-box testing.
   - **c.** Both (a) and (b)
   - **d.** None of the above.
2. The cyclomatic complexity, V(G) was developed by:
   - **a.** Howard
   - **b.** McCabe
   - **c.** Boehm
   - **d.** None of the above.
3. A node with indegree = 0 and outdegree ≠ 0 is called
   - **a.** Source node.
   - **b.** Destination node.
   - **c.** Transfer node.
   - **d.** None of the above.

4. A node with indegree ≠ 0 and outdegree = 0 is called
   - **a.** Source node.
   - **b.** Destination node.
   - **c.** Predicate node.
   - **d.** None of the above.
5. V(G) is given by which formula?
   - **a.** $V(G) = e - n + 2$
   - **b.** $V(G) = e - 2n + P$
   - **c.** $V(G) = e - 2n$
   - **d.** None of the above.
6. A predicate node is one which has
   - **a.** Two outgoing edges.
   - **b.** No outgoing edges.
   - **c.** Three or more outgoing edges.
   - **d.** None of the above.
7. An independent path is one
   - **a.** That is a complete path from source to destination node.
   - **b.** That is a path which introduces at least one new set of processing statements.
   - **c.** That is a path which introduces at most one new set of processing statements.
   - **d.** None of the above.
8. The size of the graph matrix is the
   - **a.** Number of edges in the flow graph.
   - **b.** Number of nodes in the flow graph.
   - **c.** Number of paths in the flow graph.
   - **d.** Number of independent paths.

9. In data flow testing, the objective is to find

a. All dc-paths that are not du-paths.

b. All du-paths.

c. All du-paths that are not dc-paths.

d. All dc-paths.

10. Mutation testing is related to

a. Fault seeding.

b. Functional testing.

c. Fault checking.

d. None of the above.

ANSWERS

1. b.	**2.** b.	**3.** a.	**4.** b.
5. a.	**6.** a.	**7.** b.	**8.** b.
9. c.	**10.** a.		

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

Q. 1. Write a short paragraph on test coverage analyzers?

Ans. Coverage analyzers are a class of test tools that offer automated support for this approach to testing management. With this tool, the tester runs a set of test cases on a program that has been instrumented by the coverage analyzer. The analyzer then uses the information produced by the instrumentation code to generate a coverage report. For example, in case of DD-path coverage, it identifies and labels all DD-paths in the original program. When the instrumented program is executed with test cases, the analyzer tabulates the DD-paths traversed by each test case. So, a tester can experiment with different sets of test cases to determine the coverage of each set.

Q. 2. What is slice-based testing?

Ans. A program slice is a set of program statements that contribute to or affect a value for a variable at some point in the program. This is an informal definition.

Let us see more formal definitions of a program slice.

"Given a program, P and a set of V of variables in P, a slice on the variable set V at statement n, written as S(V, n), is the set of all statements in P that contribute to the values of variables in V."

Note: Listing elements in a slice S(V, n) will be cumbersome because the elements are the program statement fragments.
We can further refine the definition of a program slice.

"Given a program P and a program graph G(P) in which statements and statement fragments are numbered and a set V of variables in P, the slice on the variable set V at statement fragment n, written as S(V, n), is the set of node numbers of all statement fragments in P prior to n that contribute to the values of variables in V at statement fragment n."

The basic idea is to separate a program into components (slices) that have some useful/functional meaning.
Please note that we have used the words "prior to" (in the 2nd definition) in the dynamic sense. So, a slice captures the execution time behavior of a program with respect to the variables in the slice. This develops a lattice, or a directed, acyclic graph of slices in which nodes are slices and edges correspond to the subset relationship.
We have also used a word, "contribute," which means that data declaration statements have an effect on the value of a variable. We simply exclude all non-executable statements.
We have five forms of usages:

1. P-use — used in a predicate (decision)
2. C-use — used in computation
3. O-use — used for output
4. L-use — used for location (pointers, subscripts)
5. I-use — used for iterations (counters, loops)

We also identify two forms of definitions nodes:

1. I-def: defined by input
2. A-def: defined by assignment

Now assume that the slice S(v, n) is a slice on one variable, i.e., the set V consists of a single variable, v. If the statement fragment "n" is a defining node for v, then n is included in the slice. Also, if the

statement fragment "n" is a usage node for v, then n is not included in the slice. P-uses and C-uses of other variables, i.e., not the v in the slice set V, are included to the extent that their execution affects the value of variable, v.

Tip: If the value of v is the same whether a statement fragment is included or excluded, then exclude the statement fragment.

Also, O-use, L-use, and I-use nodes are excluded from the slices as L-use and I-use variables are typically invisible outside their modules.

Q. 3. What is slice splicing?

Ans. If we decide to develop a program in terms of compatible slices then we could code a slice and immediately test it. We can then code and test other slices and merge them into a fair solid program. This is known as *slice splicing*.

Q. 4. What are the different sources of knowledge for white-box testing?

Ans. The following are the knowledge sources for white-box or structural testing:

a. High-level design
b. Detailed design
c. Control flow graphs

Q. 5. What is stress testing?

Ans. Stress testing is a testing technique used to determine if the system can function when subjected to large volumes of data. It includes areas like

a. Input transactions
b. Internal tables
c. Disk space
d. Output
e. Communications
f. Computer capacity
g. Interaction with users

If the application functions properly with stressed data then it is assumed that it will function properly with normal volumes of work.

Objectives of stress testing: To simulate a production environment for determining that

a. Normal or above-normal volumes of transactions can be processed through the transaction within the available time frame.

b. The system is able to process large volumes of data.
c. Enough system capacity is available.
d. Users can perform their assigned tasks.

Q. 6. How to use stress testing?

Ans. Online systems should be stress tested by allowing people to enter transactions of a normal or above normal pace.

Batch systems can be stress tested with large input batches. Please note that the error conditions should be included in tested transactions. Transactions that are used in stress testing can be obtained from test data generators, test transactions created by the test group, or the transactions previously processed in the production environment.

In stress testing, the system should be run as it would in the production environment. Operators should use standard documentation and the people entering transactions or working with the system should be the clerical personnel that will use the system.

Examples of stress testing:

i. Enter transactions that determine that sufficient disk space has been allocated to the application.
ii. Overloading the communication network with transactions.
iii. Testing system overflow conditions by entering more transactions that can be accommodated by tables, queues, and internal storage facilities, etc.

Q. 7. What is execution testing? Give some examples.

Ans. Execution testing is used to determine whether the system can meet the specific performance criteria. It includes the following:

a. Verifying the optimum use of hardware and software.
b. Determining the response time to online user requests.
c. Determining transaction processing turnaround time.

Execution testing can be done in any phase of SDLC. It can evaluate a single aspect of the system like a critical routine in the system. We can use hardware and software monitors or create quick and dirty programs to evaluate the approximate performance of a completed system. This testing may be executed onsite or offsite for the performance of the test. Please note that the earlier the technique is used, the higher is the assurance that the completed application will meet the performance criteria.

Q. 8. What is recovery testing? Give some examples.

Ans. Recovery testing is used to ensure that operations can be continued even after a disaster.

It not only verifies the recovery process but also the effectiveness of the component parts of that process. Its objectives are:

a. Document recovery procedures
b. Preserve adequate backup data
c. Training recovery personnel

Examples of recovery testing:

i. Inducing failures into one of the application system programs during processing.
ii. Recovery could be conducted from a known point of integrity to ensure that the available backup data was adequate for the recovery process.

Q. 9. What is operations testing? Give some examples.

Ans. Operations testing is designed to determine whether the system is executable during normal system operations. It evaluates both the process and the execution of the process. During different phases of SDLC, it can be used as follows:

Phase 1: Requirements Phase
During this phase, operational requirements can be evaluated to determine the reasonableness and completeness of these requirements.

Phase 2: Design Phase
During this phase, the operating procedures should be designed and thus can be evaluated.

Examples of operations testing:

i. Verifying that the file labeling and protection procedures function properly.
ii. Determining that the OS supports features and performs the predetermined tasks.

Q. 10. What is security testing?

Ans. A testing used to identify defects that are very difficult to identify. It involves determining that adequate attention is devoted to identifying security risks and determining that sufficient expertise exists to perform adequate security testing.

Examples of security testing:

i. Determining that the resources being protected are identified.
ii. Unauthorized access on online systems to ensure that the system can identify such accesses.

Q. 11. Draw a flowchart to show overall mutation testing process.
Ans.

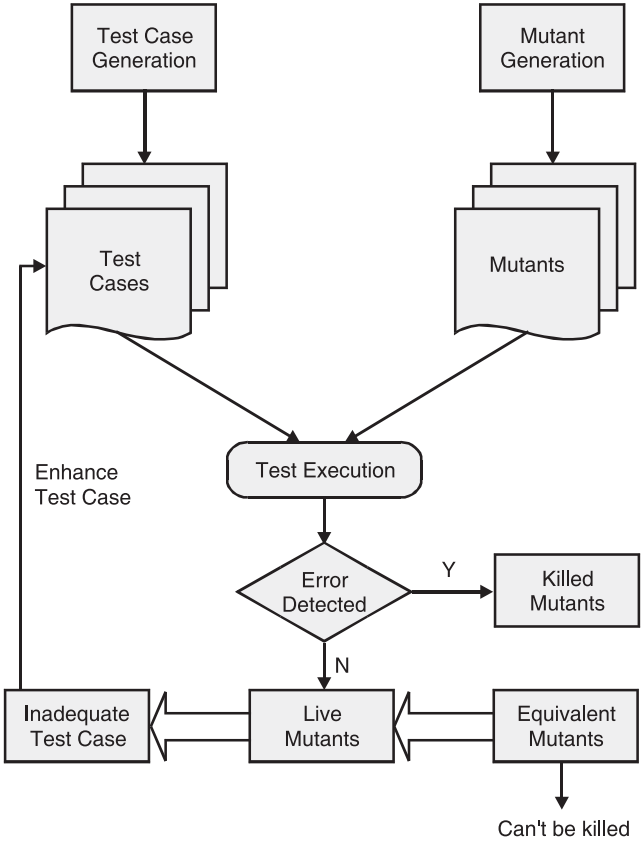


FIGURE 5.34

Q. 12. How is FTR different from the management review?
Ans. We tabulate the differences between the two.

FTR	Management review
1. It is done to examine the product.	**1.** It is done to evaluate a project plan.
2. Its purpose is to evaluate the software elements like requirements and design.	**2.** Its purpose is to ensure adequacy and completeness of each planning document for meeting project requirements.
3. It is done to ensure conformity to specifications.	**3.** It is done to ensure that project activities are progressing per the planning documents.
4. The results of FTRs are given in technical review reports (TRR).	**4.** The results of the management reviews are summarized in a management review report (MRR).
5. It is done to ensure the integrity of changes to the software elements.	**5.** The purpose here is to ensure proper allocation of resources.

REVIEW QUESTIONS

1. White-box testing is complementary to black-box testing, not alternative. Why? Give an example to prove this statement.
2. **a.** What is a flow graph and what it is used for?
   b. Explain the type of testing done using flow graph?
3. Perform the following:
   a. Write a program to determine whether a number is even or odd.
   b. Draw the paths graph and flow graph for the above problem.
   c. Determine the independent path for the above.
4. Why is exhaustive testing not possible?
5. **a.** Draw the flow graph of a binary search routine and find its independent paths.

b. How do you measure

1. Test effectiveness?

2. Test efficiency?

6. Write short paragraphs on:

a. Mutation testing.

b. Error-oriented testing.

7. Differentiate between structural and functional testing in detail.

8. Will exhaustive testing guarantee that the program is 100% correct?

9. **a.** What is cyclomatic complexity? How can we relate this to independent paths?

b. Explain the usefulness of error guessing-testing technique.

10. Differentiate between functional and structural testing.

11. Discuss the limitations of structural testing. Why do we say that complete testing is impossible?

12. Explain define/use testing. Consider any example and show du-paths. Also identify those du-paths that are not dc-paths.

13. Find the cyclomatic complexity of the graph given below:

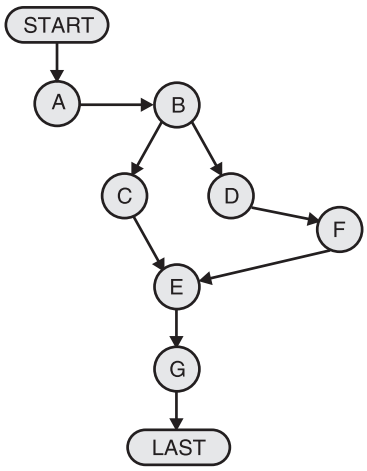


FIGURE 5.35

Find all independent paths.

14. What is data flow testing? Explain du-paths. Identify du- and dc-paths of any example of your choice. Show those du-paths that are not dc-paths.
15. Write a short paragraph on data flow testing.
16. Explain the usefulness of error guessing testing technique.
17. Discuss the pros and cons of structural testing.
18. **a.** What are the problems faced during path testing? How they can be minimized?

    b. Given the source code below:

```
void foo (int a, b, c, d, e) {
     if (a = = 0) {
     return;
     }
     int x = 0;
     if ((a = = b)     or (c = = d)) {
          x = 1;
          }
          e = 1/x;
          }
```

    List the test cases for statement coverage, branch coverage, and condition coverage.
19. Why is error seeding performed? How it is different from mutation testing?
20. **a.** Describe all methods to calculate the cyclomatic complexity.

    b. What is the use of graph matrices?
21. Write a program to calculate the average of 10 numbers. Using data flow testing design all du- paths and dc-paths in this program.
22. Write a short paragraph on mutation testing.
23. Write a C/C++ program to multiply two matrices. Try to take care of as many valid and invalid conditions are possible. Identify the test data. Justify.

24. Discuss the negative effects of the following constructs from the white-box testing point of view:

a. GO TO statements

b. Global variables

25. Write a C/C++ program to count the number of characters, blanks, and tabs in a line. Perform the following:

a. Draw its flow graph.

b. Draw its DD-paths graph.

c. Find its V(G).

d. Identify du-paths.

e. Identify dc-paths.

26. Write the independent paths in the following DD-path graph. Also calculate mathematically. Also name the decision nodes shown in Figure 5.36.

27. What are the properties of cyclomatic complexity?

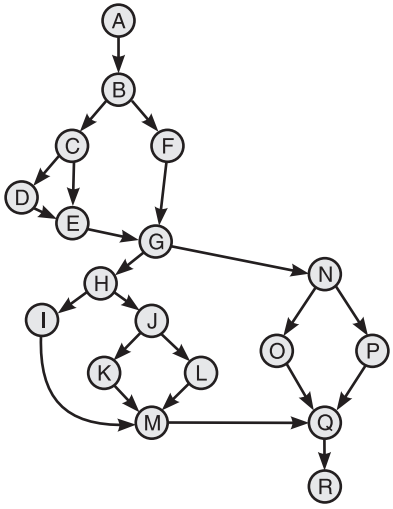


FIGURE 5.36

28. Explain in detail the process to ensure the correctness of data flow in a given fragment of code.

```
main
{
      int K = 35, Z;
      Z = check (K);
      printf ("\n%d", Z);
      }
check (m)
      {
      int m;
      if (m > 40)
            return (1);
      else
            return (0);
      }
```

29. Write a C program for finding the maximum and minimum out of three numbers and compute its cyclomatic complexity using all possible methods.

30. Consider the following program segment:

```
void sort (int a[], int n)
      {
      int i, j;
for (i = 1; i < n; i++)
for (j = i + 1, j < n; j++)
      if (a[i] > a[j])
      {
      temp = a[i];
      a[i] = a[j];
      a[j] = temp;
      }
```

i. Draw the control flow graph for this program segment.

ii. Determine the cyclomatic complexity for this program (give all intermediate steps).

iii. How is cyclomatic complexity metric useful?

31. Explain data flow testing. Consider an example and show all "du" paths. Also identify those "du" paths that are not "dc" paths.

32. Consider a program to find the roots of a quadratic equation. Its input is a triplet of three positive integers (say a, b, c) from the interval [1, 100]. The output may be one of the following words—real roots, equal roots, imaginary roots. Find all du-paths and those that are dc-paths. Develop data flow test cases.

33. If the pseudocode below were a programming language, list the test cases (separately) required to achieve 100% statement coverage and path coverage.

```
1.   If x = 3 then
2.      Display_message x;
3.      If y = 2 then
4.      Display_message y;
5.   Else
6.      Display_message z;
7.   Else
8.      Display_message z;
```

34. Consider a program to classify a triangle. Draw its flow graph and DD-path graph.

CHAPTER 6

GRAY-BOX TESTING

Inside this Chapter:

6.0. Introduction to Gray-Box Testing
6.1. What Is Gray-Box Testing?
6.2. Various Other Definitions of Gray-Box Testing
6.3. Comparison of White-Box, Black-Box, and Gray-Box Testing Approaches in Tabular Form

6.0. INTRODUCTION TO GRAY-BOX TESTING

Code coverage testing involves "dynamic testing" methods of executing the product with pre-written test cases and finding out how much of the code has been covered. If a better coverage of the code is desired, several iterations of testing may be required. For each iteration, one has to write a new set of test cases for covering those portions of code that were not covered by earlier test cases. To do such types of testing, not only does one need to understand the code and logic but one also needs to understand how to write effective test cases that can cover good portions of the code. Understanding of code and logic means white-box or structural testing whereas writing effective test cases means black-box testing. So, we need a combination of white-box and black-box techniques for test effectiveness. This type of testing is known as "*gray-box testing*." We must then understand that:

WHITE + BLACK = GRAY

6.1. WHAT IS GRAY-BOX TESTING?

Black-box testing focuses on software's external attributes and behavior. Such testing looks at an application's expected behavior from the user's point of view. White-box testing/glass-box testing, however, tests software with knowledge of internal data structures, physical logic flow, and architecutre at the source code level. White-box testing looks at testing from the developer's point of view. Both black-box and white-box testing are critically important complements of a complete testing effort. Individually, they do not allow for balanced testing. Black-box testing can be less effective at uncovering certain error types such as data-flow errors or boundary condition errors at the source level. White-box testing does not readily highlight macro-level quality risks in operating environment, compatibility, time-related errors, and usability.

6.2. VARIOUS OTHER DEFINITIONS OF GRAY-BOX TESTING

Gray-box testing incorporates the elements of both black-box and white-box testing. It considers the outcome on the user end, system-specific technical knowledge, and the operating environment. It evaluates the application design in the context of the inter-operability of system components. The gray-box testing approach is integral to the effective testing of web applications that comprise numerous components, both software and hardware. These components must be tested in the context of system design to evaluate their functionality and compatibility. Listed below are some more definitions of gray-box testing:

"Gray-box testing consists of methods and tools derived from the knowledge of the application internals and the environment with which it interacts, that can be applied in black-box testing to enhance testing productivity, bug finding and bug analyzing efficiency."

—Nguyen H.Q.

OR

"Gray-box testing is using inferred or incomplete structural or design information to expand or focus black-box testing."

—Dick Bender

OR

"Gray-box testing is designing of the test cases based on the knowledge of algorithms, interval states, architectures or other high level descriptions of program behavior."

—Dong Hoffman

OR

"Gray-box testing involves inputs and outputs, but test design is educated by information about the code or the program operation of a kind that would normally be out of scope of view of the tester."

—Cem Kaner

Gray-box testing is well suited for web application testing because it factors in high level design environment and the inter-operability conditions. It will serve problems that are not as easily considered by a black-box or white-box analysis, especially problems of end-to-end information flow and distributed hardware/software system configuration and compatibility. Context-specific errors that are germane to web systems are commonly uncovered in this process.

6.3. COMPARISON OF WHITE-BOX, BLACK-BOX, AND GRAY-BOX TESTING APPROACHES IN TABULAR FORM

Before tabulating the differences between black-box, gray-box, and white-box testing techniques, we must first understand that when we say test granularity, we mean the level of details. And when we say the highest, it means that all internals are known.

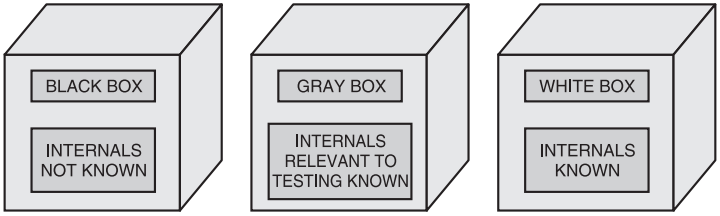


FIGURE 6.1 Comparison of Black-Box, Gray-Box, and White-Box Techniques.

Next, we tabulate the points of differences between them.

Black-box testing	Gray-box testing	White-box testing
1. Low granularity	Medium granularity	High granularity
2. Internals NOT known	Internals partially known	Internals fully known
3. Internals not required to be known	Internals relevant to testing are known	Internal code of the application and database known
4. Also known as • Opaque-box testing • Closed-box testing • Input-output testing • Data-driven testing • Behavioral • Functional testing	Also known as translucent-box testing	Also known as • Glass-box testing • Clear-box testing • Design-based testing • Logic-based testing • Structural testing • Code-based testing.
5. It is done by end-users (user acceptance testing). Also done by tester, developers.	It is done by end-users (user acceptance testing). Also done by testers, developers.	Normally done by testers and developers.
6. Testing method where • System is viewed as a black-box • Internal behavior of the program is ignored • Testing is based upon external specifications	Here, internals partly known (gray), but not fully known (white). Test design is based on the knowledge of algorithm, interval states, architecture, or other high level descriptions of the program behavior.	Internals fully known
7. It is likely to be least exhaustive of the three.	It is somewhere in between.	Potentially most exhaustive of the three.
8. Requirements based. Test cases based on the functional specifications, as internals not known.	Better variety/depth in test cases on account of high level knowledge of internals.	Ability to exercise code with relevant variety of data.
9. Being specification based if would not suffer from the deficiency as described for white-box testing.	It would also not suffer from the deficiency as described for white-box testing.	Because test cases are written based on the code, specifications missed in coding would not be revealed.
10. It is suited for functional/ business domain testing.	It is suited for functional/ business domain testing bit in depth.	It is suited for all.

(*continued*)

Black-box testing	Gray-box testing	White-box testing
11. Not suited to algorithm testing.	Not suited to algorithm testing.	Appropriate for algorithm testing.
12. It is concerned with validating outputs for given inputs, the application being treated as a black-box.	Here in, we have a better variety of inputs and the ability to extract test results from database for comparison with expected results.	It facilitates structural testing. It enables logic coverage, coverage of statements, decisions, conditions, path, and control flow within the code.
13. It can test only by trial and error data domains, internal boundaries, and overflow.	It can test data domains, internal boundaries, and overflow, if known.	It can determine and therefore test better: data domains, internal boundaries, and overflow.

SUMMARY

1. As testers, we get ideas for test cases from a wide range of knowledge areas. This is partially because testing is much more effective when we know what types of bugs we are looking for. As testers of complex systems, we should strive to attain a broad balance in our knowledge, learning enough about many aspects of the software and systems being tested to create a battery of tests that can challenge the software as deeply as it will be challenged in the rough and tumble day-to-day use.
2. Every tester in a test team need not be a gray-box tester. More is the mix of different types of testers in a team, better is the success.

MULTIPLE CHOICE QUESTIONS

1. When both black-box and white-box testing strategies are required to test a software then we can apply
   - **a.** Gray-box testing.
   - **b.** Mutation testing.
   - **c.** Regression testing.
   - **d.** None of the above.

2. Which of the following is true?
   a. There is no such testing named gray-box testing.
   b. Gray-box testing is well suited for web applications.
   c. Gray-box is same as white-box only.
   d. None of the above.
3. Gray-box testing is also known as
   a. Opaque-box testing.
   b. Clear-box testing.
   c. Translucent testing.
   d. None of the above.
4. Gray-box testing involves
   a. Low granularity.
   b. Medium granularity.
   c. High granularity.
   d. None of the above.
5. Gray-box testing is done by
   a. End users.
   b. Testers and developers.
   c. Both (a) and (b)
   d. None of the above.

ANSWERS

1. a. **2.** b. **3.** c. **4.** b. **5.** c.

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

Q. 1. Gray-box testing is based on requirement-based test case generation. Explain.

Ans. Gray-box testing uses assertion methods to preset all the conditions required, prior to a program being tested. Formal testing is one of the commonly used techniques for ensuring that a core program is correct to a very large degree. If the requirement specification language is being used to specify the requirement, it would be easy to execute the requirement stated and verify its correctness. Gray-box testing will use the predicate and verifications defined in requirement specification language as inputs to the requirements based test case generation phase.

Q. 2. Why is gray-box testing especially useful for web and Internet applications?

Ans. It is useful because the Internet is built around loosely integrated components that connected via relatively well-defined interfaces. It factors in high-level design, environment and inter-operability conditions. It reveals problems of end-to-end information flow and distributed hardware/software system configuration and compatibility. Please note that context specific errors are germane to web systems and it is our gray-box testing only that uncovers these errors.

Q. 3. Can we have only gray-box testers?

Ans. No, because a high level of success lies in the fact that we can have a mix of different types of testers with different skill sets—like database expert, security expert, test automation expert, etc.

Q. 4. What is the gray-box assumption for object-oriented software?

Ans. Typically, a reference state model for SUT is assumed. The testing problem is to identify failure to conform to the reference model. The tester must determine which state is actually obtained by applying a distinguishing sequence and observing the resulting output. This increases the number of tests by a large amount. With object-oriented software, we assume that the internal state can be determined by

a. A method of activation state where activation is equated with state.

b. State reporting capability in the class-under-test (CUT).

c. Built-in test reporting in the CUT.

This is the *Gray-Box assumption*.

REVIEW QUESTIONS

1. Define "gray-box testing."
2. Give various other definitions of gray-box testing.
3. Compare white-box, black-box and gray-box testing approaches?
4. How is gray-box testing related to white- and black-box testing?

5. Gray-box testing is well suited for web applications testing. Why?
6. Assume that you have to build the real-time large database connectivity system software. Then what kinds of testing do you think are suitable? Give a brief outline and justification for any four kinds of testing.

CHAPTER 7

LEVELS OF TESTING

Inside this Chapter:

7.0. Introduction
7.1. Unit, Integration, System, and Acceptance Testing Relationship
7.2. Integration Testing

7.0. INTRODUCTION

When we talk of levels of testing, we are actually talking of three levels of testing:

1. Unit testing
2. Integration testing
3. System testing

The three levels of testing are shown in Figure 7.1.

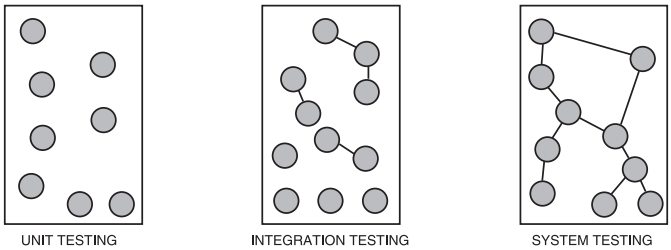


FIGURE 7.1 Levels of Testing.

Generally, system testing is functional rather than structural testing. We will now study these testing techniques one-by-one.

7.1. UNIT, INTEGRATION, SYSTEM, AND ACCEPTANCE TESTING RELATIONSHIP

We can categorize testing as follows:

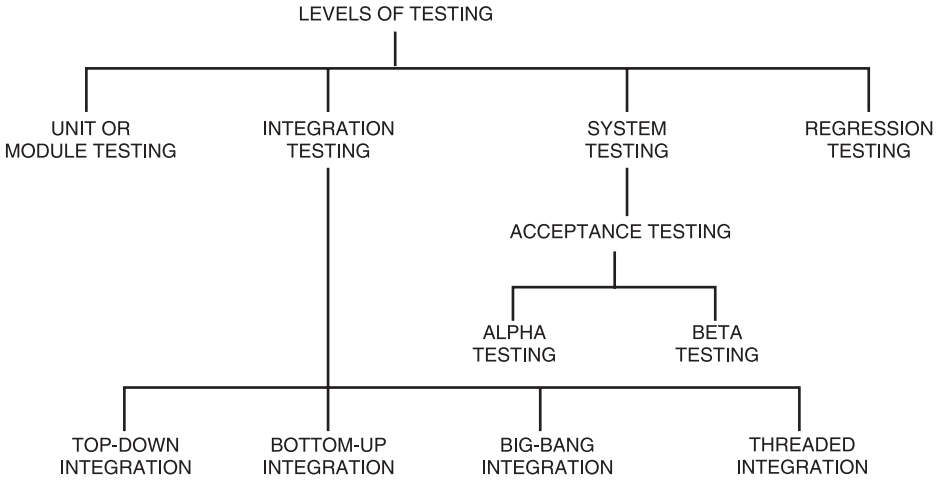


FIGURE 7.2

We will explain unit or module testing next. Consider the following diagram of unit testing as shown in Figure 7.3.

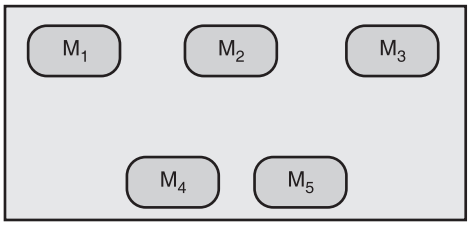


FIGURE 7.3 Unit Testing.

Unit (or module) testing "*is the process of taking a module (an atomic unit) and running it in isolation from the rest of the software product by using prepared test cases and comparing the actual results with the results predicted by the specification and design module*." It is a white-box testing technique.

Importance of unit testing:

1. Because modules are being tested individually, testing becomes easier.
2. It is more exhaustive.
3. Interface errors are eliminated.

TEST is one of the CASE tools for unit testing (Parasoft) that automatically tests classes written in MS.NET framework. The tester need not write a single test or a stub. There are tools which help to organize and execute test suites at command line, API, or protocol level. Some examples of such tools are:

S. No.	Kind of tool	Software description	Platforms
1.	Deja Gnu testing framework for interactive or batch-oriented applications.	Framework designed for regression testing and embedded system testing.	UNIX machines
2.	E-SIM-Embedded software simulation and testing environment.	E-SIM is a native simulator for embedded software.	Win 32, Solaris 5, and LINUX
3.	SMARTS-Automated Test Management Tool.	It is the software maintenance and regression test system. SMARTs automates and simplifies testing process.	Sun OS, Solaris, MS WINDOWS 95/98/NT/ 2000

7.2. INTEGRATION TESTING

A system is composed of multiple components or modules that comprise hardware and software. Integration is defined as the set of interactions among components. Testing the interaction between the modules and interaction with other systems externally is called integration testing.

It is both a type of testing and a phase of testing. The architecture and design can give the details of interactions within systems; however, testing the interactions between one system and another system requires a detailed understanding of how they work together. This knowledge of integration depends on many modules and systems. These diverse modules could have different ways of working when integrated with other systems. This introduces complexity in procedures and in what needs to be done recognizing this complexity, a phase in testing is dedicated to test their interactions, resulting in the evolution of a process. This ensuing phase is called the *integration testing phase*.

7.2.1. CLASSIFICATION OF INTEGRATION TESTING

As shown in Figure 7.4 integration testing is classified as follows:

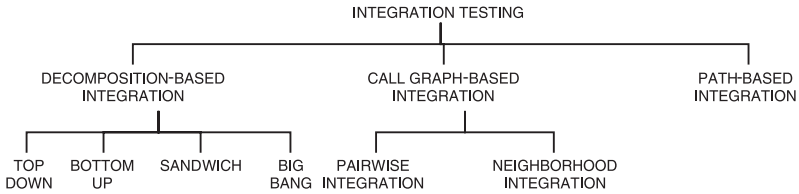


FIGURE 7.4

We will discuss each of these techniques in the following sections.

7.2.2. DECOMPOSITION-BASED INTEGRATION

When we talk of decomposition-based integration testing techniques, we usually talk of the *functional decomposition* of the system to be tested which is represented as a tree or in textual form. It is further classified as top-down, bottom-up, sandwich, and big bang integration strategies. All these integration orders presume that the units have been separately tested, thus, the goal of decomposition-based integration is to test the interfaces among separately tested units.

7.2.2.1. TYPES OF DECOMPOSITION-BASED TECHNIQUES TOP-DOWN INTEGRATION APPROACH

It begins with the main program, i.e., the root of the tree. Any lower-level unit that is called by the main program appears as a "stub." A stub is a piece of throw-away code that emulates a called unit. Generally, testers have to develop the stubs and some imagination is required. So, we draw.

Where "M" is the main program and "S" represents a stub from the figure, we find out that:

$$\text{Number of Stubs Required} = (\text{Number of Nodes} - 1)$$

Once all of the stubs for the main program have been provided, we test the main program as if it were a standalone unit.

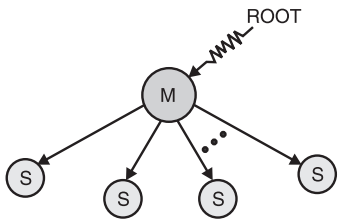


FIGURE 7.5 Stubs.

We could apply any of the appropriate functional and structural techniques and look for faults. When we are convinced that the main program logic is correct, we gradually replace the stubs with the actual code. Top-down integration follows a breadth-first traversal (bfs) of the functional decomposition tree.

7.2.2.2. BOTTOM-UP INTEGRATION APPROACH

It is a mirror image to the top-down order with the difference that stubs are replaced by driver modules that emulate units at the next level up in the tree. In bottom-up integration, we start with the leaves of the decomposition tree and test them with specially coded drivers. Less throw-away code exists in drivers than there is in stubs. Recall that there is one stub for each child node in the decomposition tree.

Most systems have a fairly high fan-out near the leaves, so in the bottom-up integration order, we will not have as many drivers. This is partially offset by the fact that the driver modules will be more complicated. See Figure 7.6

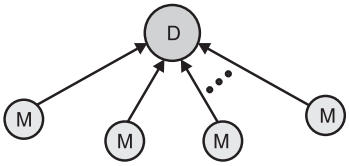


FIGURE 7.6 Drivers.

where "D" represents a driver module and "M" represents other modules.

Number of drivers required = (No. of nodes – No. of leaf nodes)

For Figure 7.6, drivers required = 5 – 4 = 1,

i.e., 1 driver module (D) is required.

7.2.2.3. SANDWICH INTEGRATION APPROACH

It is a combination of top-down and bottom-up integration. There will be less stub and driver development effort. But the problem is in the difficulty of fault isolation that is a consequence of big bang integration.

Because this technique is the combination of the top-down and bottom-up integration approaches, it is also called *bidirectional integration*. It is performed initially with the use of stubs and drivers. Drivers are used to provide upstream connectivity while stubs provide downstream connectivity. A *driver* is a function which redirects the request to some other component and stubs simulate the behavior of a missing component. After the functionality of these integrated components are tested, the drivers and stubs are discarded. This technique then focuses on those components which need focus and are new. This approach is called as *sandwich integration*.

In the product development phase when a transition happens from two-tier architecture to three-tier architecture, the middle tier gets created as a set of new components from the code taken from bottom-level applications and top-level services.

7.2.2.4. BIG-BANG INTEGRATION

Instead of integrating component by component and testing, this approach waits until all the components arrive and one round of integration testing is done. This is known as *big-bang integration*. It reduces testing effort and removes duplication in testing for the multi-step component integrations. Big-bang integration is ideal for a product where the interfaces are stable with fewer number of defects.

7.2.2.5. PROS AND CONS OF DECOMPOSITION-BASED TECHNIQUES

The decomposition-based approaches are clear (except big-bang integration). They are built with tested components. Whenever a failure is observed, the most recently added unit is suspected. Integration testing progress is easily tracked against the decomposition tree. The top-down and bottom-up terms suggest breadth-first traversals of the decomposition tree but this is not mandatory.

One common problem to functional decomposition is that they are artificial and serve the needs of project management more than the need of software developers. This holds true also for decomposition-based testing. The whole mechanism is that units are integrated with respect to structure. This presumes that correct behavior follows from individually correct units and correct interfaces.

The development effort for stubs and drivers is another drawback to these approaches and this is compounded by the retesting effort. We try to compute the number of *integration test sessions* for a given decomposition tree. A test session is defined as one set of tests for a specific configuration actual code and stubs. Mathematically,

$$\text{Sessions} = \text{Nodes} - \text{Leaves} + \text{Edges}$$

For example, if a system has 42 integration testing sessions then it means 42 separate sets of integration test cases, which is too high.

7.2.2.6. GUIDELINES TO CHOOSE INTEGRATION METHOD AND CONCLUSIONS

S. No.	Factors	Suggested method
1.	Clear requirements and design.	Top-down approach.
2.	Dynamically changing requirements, design, and architecture.	Bottom-up approach.
3.	Changing architecture and stable design.	Sandwich (or bi-directional) approach.
4.	Limited changes to existing architecture with less impact.	Big-bang method.
5.	Combination of above.	Select any one after proper analysis.

7.2.3. CALL GRAPH-BASED INTEGRATION

One of the drawbacks of decomposition-based integration is that the basis is the *functional decomposition tree*. But if we use the call graph-based technique instead, we can remove this problem. Also, we will move in the direction of structural testing. Because call graph is a directed graph thus, we can use it as a program graph also. This leads us to two new approaches to integration testing which are discussed next.

7.2.3.1. PAIRWISE INTEGRATION

The main idea behind pairwise integration is to eliminate the stub/driver development effort. Instead of developing stubs and drivers, why not use the actual code? At first, this sounds like big-bang integration but we restrict a session to only a pair of units in the call graph. The end result is that we have one integration test session for each edge in the call graph. This is not much of a reduction in sessions from either top-down or bottom-up but it is a drastic reduction in stub/driver development. Four pairwise integration sessions are shown in Figure 7.7.

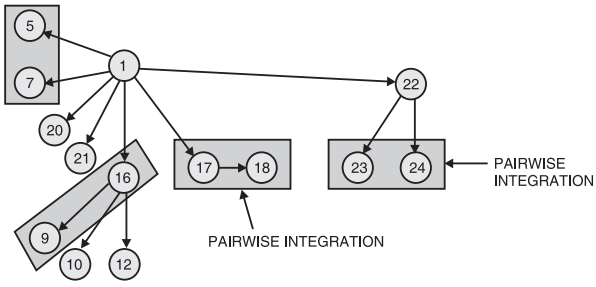


FIGURE 7.7 Pairwise Integration.

7.2.3.2. NEIGHBORHOOD INTEGRATION

The neighborhood of a node in a graph is the set of nodes that are one edge away from the given node. In a directed graph, this includes all of the immediate predecessor nodes and all of the immediate successor nodes. Please note that these correspond to the set of stubs and drivers of the node.

For example, for node-16, neighborhood nodes are 9, 10, and 12 nodes as successors and node-1 as predecessor node.

We can always compute the number of neighbors for a given call graph. Each interior node will have one neighborhood plus one extra in case leaf nodes are connected directly to the root node.

NOTE *An interior node has a non-zero in-degree and a non-zero out-degree.*

So, we have the following formulas:

1. Interior nodes = Nodes – (Source Nodes + Sink Nodes)
2. Neighborhoods = Interior Nodes + Source Nodes

Substituting 1st equation in 2nd equation above, we get:

Neighborhoods = Nodes – Sink Nodes

Neighborhood integration yields a drastic reduction in the number of integration test sessions. It avoids stub and driver development. The end result is that the neighborhoods are essentially the sandwiches that we slipped past in the previous section.

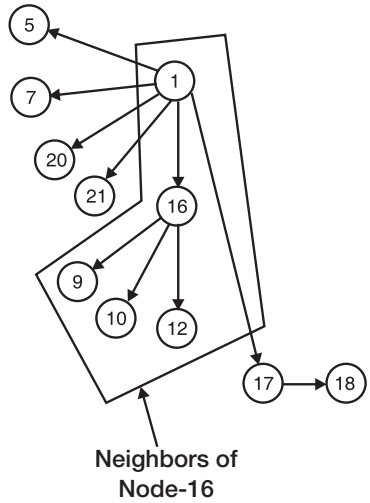


FIGURE 7.8 Neighborhood Integration.

7.2.3.3. PROS AND CONS

The call graph-based integration techniques move away from a purely structural basis toward a behavioral basis. These techniques also eliminate the stub/driver development effort. This technique matches well with the developments characterized by builds. For example, sequences of the neighborhood can be used to define builds.

The biggest drawback to call graph-based integration testing is the fault isolation problem, especially for large neighborhoods. Another problem occurs when a fault is found in a node (unit) that appears in several neighborhoods. Obviously, we resolve the fault but this means changing the unit's code in some way, which in turn means that all the previously tested neighborhoods that contain the changed node need to be retested.

7.2.4. PATH-BASED INTEGRATION WITH ITS PROS AND CONS

We already know that the combination of structural and functional testing is highly desirable at the unit level and it would be nice to have a similar capability for integration and system testing. Our goal for integration testing is: "*Instead of testing interfaces among separately developed and tested units, we focus on interactions among these units*." Here, *cofunctioning* might be a good term. Interfaces are structural whereas interaction is behavioral.

We now discuss some basic terminologies that are used in this technique.

1. **Statement fragment.** It is a complete statement. These are the nodes in the program graph.
2. **Source node.** A source node in a program is a statement fragment at which program execution begins or resumes. The first executable statement in a unit is clearly a source node. Source nodes also occur immediately after nodes that transfer control to other units.
3. **Sink node.** It is a statement fragment in a unit at which program execution terminates. The final executable statement in a program is clearly a sink node, so are the statements that transfer control to other units.
4. **Module execution path (MEP).** It is a sequence of statements that begins with a source node and ends with a sink node with no intervening sink nodes.
5. **Message.** A message is a programming language mechanism by which one unit transfers control to another unit. Depending on the programming language, messages can be interpreted as subroutine invocations, procedure calls, and function references. We follow the convention that the unit that receives a message always eventually returns control to the message source. Messages can pass data to other units.

6. **Module-to-module path (MM-path).** An MM-path is an interleaved sequence of module execution paths (MEPs) and messages.
7. **Module-to-module path graph (MM-path graph).** Given a set of units, their MM-path graph is the directed graph in which nodes are module execution paths and edges correspond to messages and returns from one unit to another. The effect of these definitions is that program graphs now have multiple source and sink nodes. This would increase the complexity of unit testing but the integration testing presumes unit testing is complete. Also, now our goal is to have an integration testing analog of DD-paths, as done earlier.

The basic idea of an MM-path is that we can now describe sequences of module execution paths that include transfers of control among separate units. These transfers are by messages, therefore, MM-paths always represent feasible execution paths and these paths cross unit boundaries. We can find MM-paths in an extended program graph in which nodes are module execution paths and edges are messages.

Consider a hypothetical example as shown in Figure 7.9.

Herein, module-A calls module-B, which in turn calls module-C. Note from Figure 7.9 that MM-path begins with and ends in the main program only. This is true for traditional (or procedural) software.

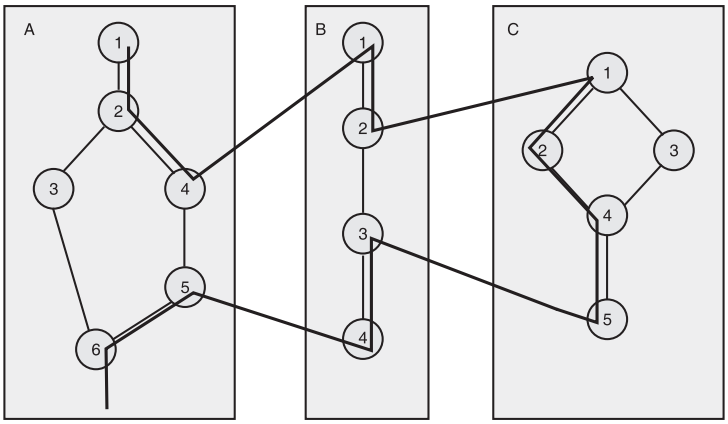


FIGURE 7.9 MM-Path Across Three Units (A, B, and C).

In module-A, nodes 1 and 5 are source nodes and nodes 4 and 6 are sink nodes. Similarly, in module-B, nodes 1 and 3 are source nodes and nodes 2

and 4 are sink nodes. Module-C has a single source node 1 and a single sink node, 5. This can be shown as follows:

Module	Source-node	Sink-node
A	1, 5	4, 6
B	1, 3	2, 4
C	1	5

So, the seven module execution paths are as follows:

MEP (A, 1) = <1, 2, 3, 6>
MEP (A, 2) = <1, 2, 4>
MEP (A, 3) = <5, 6>
MEP (B, 1) = <1, 2>
MEP (B, 2) = <3, 4>
MEP (C, 1) = <1, 2, 4, 5>
MEP (C, 2) = <1, 3, 4, 5>

These are the module execution paths. We can now define an integration testing analog of the DD-path graph that serves unit testing so effectively.

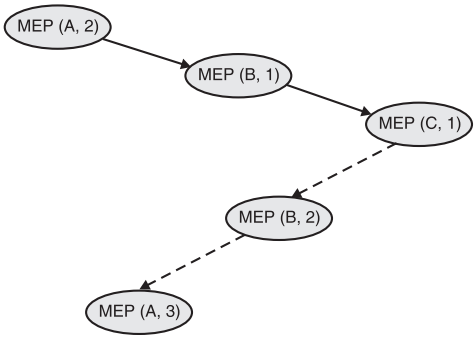


FIGURE 7.10 MM-Path Graph Derived from Figure 7.9.

Now, its MM-path graph is shown in Figure 7.10.

Herein, the solid arrows indicate messages and the dotted arrows represent returns.

Also, note that a program path is a sequence of DD-paths and an MM-path is a sequence of module execution paths.

What Are the Endpoints on MM-Paths?

Two criteria that are observed and are behavioral put endpoints on MM-paths:

1. Message quiescence
2. Data quiescence

Message quiescence occurs when a unit that sends no messages is reached. For example, module-C in Figure 7.9.

Data quiescence occurs when a sequence of processing culminates in the creation of stored data that is not immediately used. This happens in a data flow diagram. Points of quiescence are natural endpoints for an MM-path.

Pros and Cons

1. MM-paths are a hybrid of functional (black-box) and structural (white-box) testing. They are functional because they represent actions with inputs and outputs. As such, all the functional testing techniques are potentially applicable. The structural side comes from how they are identified, particularly the MM-path graph. The net result is that the cross-check of the functional and structural approaches is consolidated into the constructs for path-based integration testing. We therefore avoid the pitfall of structural testing and, at the same time, integration testing gains a fairly seamless junction with system testing. Path-based integration testing works equally well for software developed in the traditional waterfall process or with one of the composition-based alternative life-cycle models. Later, we will also show that the concepts are equally applicable to object-oriented software testing.
2. The most important advantage of path-based integration testing is that it is closely coupled with the actual system behavior, instead of the structural motivations of decomposition and call graph-based integration.
3. The disadvantage of this technique is that more effort is needed to identify the MM-paths. This effort is probably offset by the elimination of stub and driver development.

7.2.5. System Testing

System testing focuses on a complete, integrated system to evaluate compliance with specified requirements. Tests are made on characteristics that are only present when the entire system is run.

7.2.5.1. What is system testing?

The testing that is conducted on the complete integrated products and solutions to evaluate system compliance with specified requirements on functional and non functional aspects is called *system testing*. It is done after unit, component, and integration testing phases.

As we already know, a system is defined as a set of hardware, software, and other parts that together provide product features and solutions. In order to test the entire system, it is necessary to understand the product's behavior as a whole. System testing brings out issues that are fundamental to the design, architecture, and code of the whole product.

System-level tests consist of batteries of tests that are designed to fully exercise a program as a whole and check that all elements of the integrated system function properly. System-level test suites also validate the usefulness of a program and compare end results against requirements.

System testing is the only testing phase that tests both functional and non functional aspects of the product.

On the functional side, system testing focuses on real-life customer usage of the product and solutions. It simulates customer deployments. For a general-purpose product, system testing also means testing it for different business verticals and applicable domains such as insurance, banking, asset management, and so on.

On the non-functional side, it brings into consideration different testing types which are also called quality factors.

7.2.5.2. WHY IS SYSTEM TESTING DONE?

System testing is done to:

1. Provide independent perspective in testing as the team becomes more quality centric.
2. Bring in customer perspective in testing.
3. Provide a "fresh pair of eyes" to discover defects not found earlier by testing.
4. Test product behavior in a holistic, complete, and realistic environment.
5. Test both functional and non functional aspects of the product.
6. Build confidence in the product.
7. Analyze and reduce the risk of releasing the product.
8. Ensure all requirements are met and ready the product for acceptance testing.

Explanation: An independent test team normally does system testing. This independent test team is different from the team that does

the component and integration testing. The system test team generally reports to a manager other than the product-manager to avoid conflicts and to provide freedom to individuals during system testing. Testing the product with an independent perspective and combining that with the perspective of the customer makes system testing unique, different, and effective.

The behavior of the complete product is verified during system testing. Tests that refer to multiple modules, programs, and functionality are included in system testing. This task is critical as it is wrong to believe that individually tested components will work together when they are put together.

System testing is the last chance for the test team to find any leftover defects before the product is handed over to the customer.

System testing strives to always achieve a balance between the objective of finding defects and the objective of building confidence in the product prior to release.

The analysis of defects and their classification into various categories (called as impact analysis) also gives an idea about the kind of defects that will be found by the customer after release. If the risk of the customers getting exposed to the defects is high, then the defects are fixed before the release or else the product is released as such. This information helps in planning some activities such as providing workarounds, documentation on alternative approaches, and so on. Hence, system testing helps in reducing the risk of releasing a product.

System testing is highly complementary to other phases of testing. The component and integration test phases are conducted taking inputs from functional specification and design. The main focus during these testing phases are technology and product implementation. On the other hand, customer scenarios and usage patterns serve as the basis for system testing.

7.2.5.3. Functional versus non functional system testing (in tabular form)

We are now in a position to state an equation:

System testing = Functional testing + Non functional testing

We first tabulate the differences between functional and non functional testing in a tabular form.

Functional testing	Non functional testing
1. It involves the product's functionality.	**1.** It involves the product's quality factors.
2. Failures, here, occur due to code.	**2.** Failures occur due to either architecture, design, or due to code.
3. It is done during unit, component, integration, and system testing phase.	**3.** It is done in our system testing phase.
4. To do this type of testing only domain of the product is required.	**4.** To do this type of testing, we need domain, design, architecture, and product's knowledge.
5. Configuration remains same for a test suite.	**5.** Test configuration is different for each test suite.

Thus, *functional testing* helps in verifying what the system is supposed to do. It aids in testing the product's features or functionality. It has only two results—requirements met or not met. It should have very clear expected results documented in terms of the behavior of the product. It has simple methods and steps to execute the test cases. Functional testing results normally depend on the product and not on the environment. It uses a predetermined set of resources. It requires in-depth customer, product, and domain knowledge to develop different test cases and find critical defects. It is performed in all phases of testing, i.e., unit, component, integration, and system testing.

Non functional testing is performed to verify the *quality factors* such as reliability, scalability, etc. These quality factors are also called non functional requirements. It requires the expected results to be documented in qualitative and quantifiable terms. It requires a large amount of resources and the results are different for different configurations and resources. It is a very complex method as a large amount of data needs to be collected and analyzed. The focus point is to qualify the product. It is not a defect finding exercise. Test cases for non functional testing includes a clear pass/fail criteria.

However, test results are concluded both on pass/fail definitions and on the experiences encountered in running the tests. Non functional test results are also determined by the amount of effort involved in executing them and any problems faced during execution. For example, if a performance test met the pass/fail criteria after 10 iterations, then the experience is bad and

the test result cannot be taken as pass. Either the product or the non functional testing process needs to be fixed here.

Non functional testing requires understanding the product behavior, design, architecture, and also knowing what the competition provides. It also requires analytical and statistical skills as the large amount of data generated requires careful analysis. Failures in non functional testing affect the design and architecture much more than the product code. Because non functional testing is not repetitive in nature and requires a stable product, it is performed in the system testing phase.

The differences listed in the table above are just the guidelines and not the dogmatic rules.

Because both functional and non functional aspects are being tested in the system testing phase so the question that arises is—*what is the ratio of the test-cases or effort required for the mix of these two types of testing?* The answer is here: Because functional testing is a focus area starting from the unit testing phase while non functional aspects get tested only in the system testing phase, it is a good idea that a majority of system testing effort be focused on the non functional aspects. A 70%–30% ratio between non functional and functional testing can be considered good and 50%–50% ratio is a good starting point. However, this is only a guideline and the right ratio depends more on the context, type of release, requirements, and products.

7.2.5.4. FUNCTIONAL SYSTEM TESTING TECHNIQUES

As functional testing is performed at various testing phases, there are two problems that arise. They are:

1. **Duplication:** It refers to the same tests being performed multiple times.
2. **Gray area:** It refers to certain tests being missed out in all the phases.

A small percentage of duplication across phases cannot be avoided as different teams are involved performing cross-reviews (i.e., involving teams from earlier phases of testing) and looking at the test cases of the previous phase before writing system test cases can help in minimizing the duplication. However, a small percentage of duplication is advisable as then different test teams will test the features with different perspectives thus yielding new defects.

Gray-areas in testing happens due to a lack of product knowledge, lack of knowledge of customer usage, and lack of coordination across test teams. These areas (missing of tests) arise when a test team assumes that a particular

test may be performed in the next phase. So, the guideline is—"*A test case moved from a later phase to an earlier phase is a better option than delaying a test case from an earlier phase to a later phase, as the purpose of testing is to find defects as early as possible.*" This has to be done after completing all tests meant for the current phase, without diluting the tests of the current phase.

We are now in a position to discuss various functional system testing techniques in detail. They are discussed one by one.

7.2.5.4.1. Design/Architecture verification

We can compare functional testing with integration testing. They are given in table below:

S. No.	Integration testing	System testing
1.	The test cases are created by looking at interfaces.	The test cases are created first and verified with design and architecture.
2.	The integration test cases focus on interactions between modules or components.	The functional system test focuses on the behavior of the complete product.

In this method of system testing, the test cases are developed and checked against the design and architecture to see whether they are actual product-level test cases. This technique helps in validating the product features that are written based on customer scenarios and verifying them using product implementation.

If there is a test case that is a customer scenario but failed validation using this technique, then it is moved to the component or integration testing phase. Because functional testing is performed at various test phases, it is important to reject the test cases and move them to an earlier phase to catch defects early and avoid any major surprise at later phases.

We now list certain guidelines that are used to reject test cases for system functional testing. They are:

1. Is this test case focusing on code logic, data structures, and unit of the product?
   If yes, then it belongs to *unit testing*.
2. Is this specified in the functional specification of any component?
   If yes, then it belongs to *component testing*.

3. Is this specified in design and architecture specification for integration testing?
   If yes, then it belongs to *integration testing*.
4. Is it focusing on product implementation but not visible to customers?
   If yes, then it is focusing on implementation to be covered in *unit/component/integration testing*.
5. Is it the right mix of customer usage and product implementation?
   If yes, then it belongs to *system testing*.

7.2.5.4.2. Business vertical testing (BVT)

Using and testing the product for different business verticals such as banking, insurance, asset management, etc. and verifying the business operations and usage is called as *business vertical testing*. In this type of testing, the procedure in the product is altered to suit the process in the business. For example, in personal loan processing, the loan is approved first by the senior officer and then sent to a clerk. User objects such as a clerk and officer are created by the product and associated with the operations. This is one way of customizing the product to suit the business. Some operations that can only be done by some user objects is called a role-based operation. BVT involves three aspects. They are:

1. Customization
2. Terminology
3. Syndication

Customization: It is important that the product understands the business processes and includes customization as a feature so that different business verticals can use the product. With the help of this feature, a general workflow of a system is altered to suit specific business verticals.

Terminology: To explain this concept, we consider a common example of e-mail. When an e-mail is sent in a loan processing system than it is called a loan application. An e-mail sent in the insurance context may be called a claim and so on. The users would be familiar with this terminology rather than the generic terminology of "e-mail." So, the user interface should reflect these terminologies rather than use generic terminology e-mails, which may dilute the purpose and may not be understood clearly by the users. An e-mail sent to a blood bank cannot take the same priority as an internal e-mail sent to an employee by another employee. These differentiations need to be made. Some e-mails need to be tracked. For example, an e-mail to a blood

bank service needs a prompt reply. Some mail can be given automated mail replies also. Hence, the terminology feature of the product should call the e-mail appropriately as a claim or a transaction and also associate the profile and properties in a way a particular business vertical works.

Syndication: Not all the work needed for business verticals is done by product development organizations only. Even the solution integrators, service providers pay a license fee to a product organization and sell the products and solutions using their name and image. In this case, the product name, company name, technology names, and copyrights may belong to the latter parties or associations and the former would like to change the names in the product. A product should provide features for those syndications in the product and they are tested as a part of BVT.

How BVT Can Be Done?

BVT can be done in two ways:

1. Simulation
2. Replication

Simulation: In simulation of a vertical test, the customer or the tester assumes requirements and the business flow is tested.

Replication: In replication, customer data and process is obtained and the product is completely customized, tested, and the customized product as it was tested is released to the customer.

NOTES

Business verticals are tested through scenarios. Scenario testing is only a method to evolve scenarios and ideas and is not meant to be exhaustive. It is done from interfaces point of view. Having some business vertical scenarios created by integration testing ensures quick progress in system testing. In the system testing phase, the business verticals are completely tested in real-life customer environment using the aspects such as customization, terminology, and syndication as described above.

7.2.5.4.3. Deployment testing

System testing is the final phase before the product is delivered to the customer. The success or failure of a particular product release is assured on the basis of the how well the customer requirements are met. This type of deployment (simulated) testing that happens in a product development

company to ensure that customer deployment requirements are met is called as *offsite deployment*. Deployment testing is also conducted after the release of the product by utilizing the resources and setup available in customer's locations. This is a combined effort by the product development organization and the organization trying to use the product. This is called *onsite deployment*. Although onsite deployment is not conducted in the system testing phase, it is the system testing team that completes this test. Onsite deployment testing is considered to be a part of acceptance testing.

We will now discuss, onsite deployment testing in detail. It is done in two stages:

I. The first stage (stage-1)

II. The second stage (stage-2)

In the first stage (stage-1), the actual data from the live system is taken and similar machines and configurations are mirrored and the operations from the user are rerun on the mirrored deployment machine (see Figure 7.11).

This gives an idea whether the enhanced or similar product can perform the existing functionality without affecting the user. This also reduces the risk of a product not being able to satisfy existing functionality, as deploying the product without adequate testing can cause major business loss to an organization. Some deployments use "intelligent-recorders" to record the transactions that happen on a live system and commit these operations on a mirrored system and then compare the results against the live system. The objective of the recorder is to help in keeping the mirrored and live system identical with respect to business transactions.

In the second stage (stage-2) after a successful first stage, the mirrored system is made a live system that runs the new product (see Figure 7.12).

Regular backups are taken and alternative methods are used to record the incremental transactions from the time the mirrored system became alive. The recorder that was used in the first stage can also be used here. In this stage, the live system that was used earlier and the recorded transactions from the time mirrored system became live, are preserved to enable going back to the old system if any major failures are observed at this stage. If no failures are observed in this (second) stage of deployment for an extended period (say, 1 month) then the onsite deployment is considered successful and the old live system is replaced by the new system.

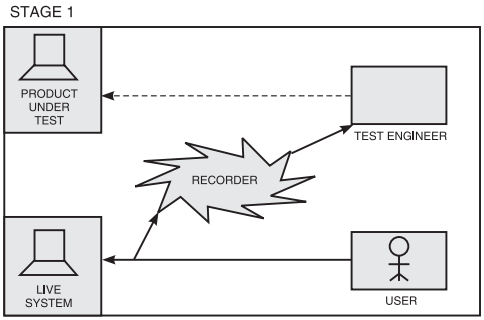


FIGURE 7.11 Stage-1 of Onsite Deployment.

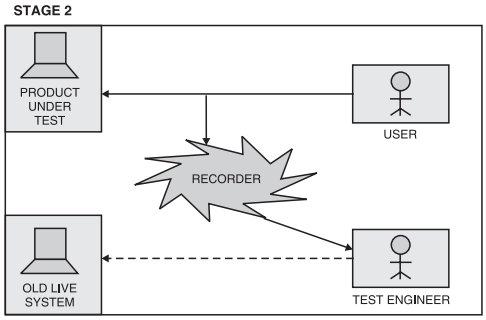


FIGURE 7.12 Stage-2 of the Onsite Deployment.

Please note that in stage-1, the recorder intercepts the user and the live system to record all transactions. All the recorded transactions from the live system are then played back on the product under test under the supervision of the test engineer (as shown by dotted lines). In stage-2, the test engineer records all transactions using a recorder and other methods and plays back on the old live system (as shown again by dotted lines). So, the overall stages are:

Live system → Mirrored system → Live system

7.2.5.4.4. Beta testing

Any project involves a significant amount of effort and time. Customer dissatisfaction and time slippages are very common.

If a customer rejects a project then it means a huge loss to the organization. Several reasons have been found for customer dissatisfaction. Some of them are as follows:

1. **Implicit requirements like ease of use.** If not found in a software then it may mean rejection by the customer.
2. **Customer's requirements keep changing constantly.** Requirements given at the beginning of the project may become obsolete. A failure to reflect changes in the product makes it obsolete.
3. **Finding ambiguous requirements** and not resolving them with the customer results in rejection of the product.

4. Even if understanding of requirements may be correct but their implementation could be wrong. So, design and code need to be reworked. If this is not done in time, it results in product rejection.
5. Poor documentation and difficulty in using of the product may also result in rejection.

To reduce risks, which is the objective of system testing, periodic feedback is obtained on the product. This type of testing in which the product is sent to the customers to test it and receive the feedback is known as beta testing.

During *beta program* (various activities that are planned and executed according to a specific schedule), some of the activities involved are as follows:

1. *Collecting the list of customers and their beta testing requirements* alongwith their expectations on the product.
2. *Working out a beta program schedule* and informing the customers.
3. *Sending some documents* for reading in advance and training the customer on product usage.
4. *Testing the product* to ensure that it meets "beta testing entry criteria" which is prepared by customers and management groups of the vendor.
5. *Sending beta program to the customer* and enable them to carry out their own testing.
6. *Collecting the feedback periodically* from the customers and prioritizing the defects for fixing.
7. *Responding to customer's feedback* with product fixes or documentation changes and closing the communication loop with the customers in a timely fashion.
8. *Analyzing and concluding* whether the beta program met the exit criteria.
9. *Communicate the progress* and action items to customers and formally closing the beta program.
10. *Incorporating the appropriate changes* in the product.

Deciding the timing of beta test poses several conflicts. Sending the product too early, with inadequate internal testing will make the customers unhappy and may create a bad impression on the quality of the product.

Sending the product too late may mean too little a time for beta defect fixes and this one defeats the purpose of beta testing. So, late integration testing phase and early system testing phase is the ideal time for starting a beta program.

We send the defect fixes to the customers as soon as problems are reported and all necessary care has to be taken to ensure the fixes meets the requirements of the customer.

How many beta customers should be chosen?

If the number chosen are too few, then the product may not get a sufficient diversity of test scenarios and test cases.

If too many beta customers are chosen, then the engineering organization may not be able to cope with fixing the reported defects in time. Thus, the number of beta customers should be a delicate balance between providing a diversity of product usage scenarios and the manageability of being able to handle their reported defects effectively.

Finally, the success of a beta program depends heavily on the willingness of the beta customers to exercise the product in various ways.

7.2.5.4.5. Certification, standards, and testing for compliance

A product needs to be certified with the popular hardware, operating system (OS), database, and other infrastructure pieces. This is called certification testing. A product that doesn't work with any of the popular hardware or software may be unsuitable for current and future use. The sale of the product depends on whether it was certified with the popular systems or not. Not only should the product co-exist and run with the current versions of these popular systems, but the product organization should also document a commitment to continue to work with the future versions of the popular systems. This is one type of testing where there is equal interest from the product development organization, the customer, and certification agencies to certify the product. The certification agencies produce automated test suites to help the product development organization.

There are many standards for each technology area and the product may need to conform to those standards. Standards can be like IPv6 in networking and 3G in mobile technology. Tools associated with those open standards can be used cost free to verify the standard's implementation. Testing the product to ensure that these standards are properly implemented is called *testing for standards*. Once the product is tested for a set of standards, they are published in the release documentation for the customer's information so that they know what standards are implemented in the product.

There are many contractual and legal requirements for a product. Failing to meet these may result in business loss and bring legal action against the organization and its senior management.

The terms certification, standards, and compliance testing are used interchangeably. There is nothing wrong in the usage of terms as long as the objective of testing is met. For example, a certifying agency helping an organization meet standards can be called both certification testing and standards testing.

7.2.5.5. Non functional testing techniques

Non functional testing differs from the functional testing in terms of complexity, knowledge requirement, effort needed, and the number of times the test cases are repeated. Because repeating non functional test cases involves more time, effort, and resources, the process for non functional testing has to be stronger than functional testing to minimize the need for repetition. This is achieved by having more stringent entry/exit criteria and better planning.

7.2.5.5.1. Setting up the configuration

Due to the varied types of customers, resources availability, time involved in getting the exact setup, and so on setting up a scenario that is exactly real life is difficult. Due to several complexities involved, simulated setup is used for non functional testing where actual configuration is difficult to get.

In order to create a "near real-life" environment, the details regarding customer's hardware setup, deployment information, and test data are collected in advance. Test data is built based on the sample data given. If it is a new product then information regarding similar or related products is collected. These inputs help in setting up the test environment close to the customer's so that the various quality characteristics of the system can be verified more accurately.

7.2.5.5.2. Coming up with entry/exit criteria

Meeting the entry criteria is the responsibility of the previous test phase, i.e., integration testing phase or it could be the objective of dry-run tests performed by the system testing team, before accepting the product for system testing. The table below gives some examples of how entry/exit

criteria can be developed for a set of parameters and for various types of non functional tests.

Types of test	Parameters	Sample entry criteria	Sample exit criteria
1. Scalability	Maximum limits	Product should scale up to one million records or 1000 users.	Product should scale up to 10 million records or 5000 users.
2. Performance test	▪ Response time ▪ Throughput ▪ Latency	Query for 1000 records should have a response time less than 3 seconds.	Query for 10,000 records should have response time less than 3 seconds.
3. Stress	System when stressed beyond the limits.	25 clients login should be possible in a configuration that can take only 20 clients.	Product should be able to withstand 100 clients logic simultaneously.

7.2.5.5.3. Managing key resources

There are four key resources:

- CPU
- Disk
- Memory
- Network

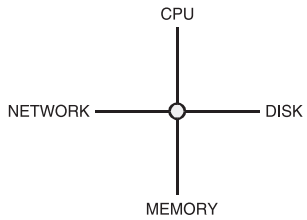


FIGURE 7.13

We need to completely understand their relationship to implement the non functional testing technique. These four resources need to be judiciously balanced to enhance the quality factors of the product. All these resources are interdependent. For example, if the memory requirements in the system are addressed, the need for the CPU may become more intensive. The demand for the resources tends to grow when a new release of the product is produced as software becomes more and more complex. Software is meant not only for computers but also for equipment such as cell phones; hence upgrading the resources is not easy anymore.

Usually, the customers are perplexed when they are told to increase the number of CPUs, memory, and the network bandwidth for better

performance, scalability, and other non functional aspects. When we ask customers to upgrade the resources one important aspect, i.e., ROI (return-on-investment), needs to be justified clearly.

Before conducting non functional testing, some assumptions are validated by the development team and customers. Some of them are given below:

1. When a higher priority job comes in, the CPU must be freed.
2. Until a new job requires memory, the available memory can be completely used.
3. If we can justify the benefits for each new resource that is added then the resources can be added easily to get better performance.
4. The product can generate many network packets as long as the network bandwidth and latency is available. Now, most of the packets generated are for LAN and not for WAN—an assumption. In case of WAN or routes involving multiple hops, the packets generated by the product need to be reduced.
5. More disk space or the complete I/O bandwidth can be used for the product as long as they are available while disk costs are getting cheaper, I/O bandwidth is not.
6. The customer gets the maximum return on investment (ROI) only if the resources such as CPU, disk, memory, and network are optimally used. So, there is intelligence needed in the software to understand the server configuration and its usage.

Without these assumptions being validated, there cannot be any good conclusion that can be made out of non functional testing.

7.2.5.5.4. Scalability testing

This type of testing is done when stakeholders like business owners and operations departments want to know whether a system not only meets its efficiency requirements now but will continue to do so in future.

Please understand that both scalability problems and now scalability requirements typically arise once a system has become operative. So, it is done on systems that are in production. Both technical test analyst and test manager ensure that scalability requirements are captured.

The primary objective is the evaluation of resources like the usage of memory space, disk capacity, and network bandwidth. For interconnected systems, it means testing the network's capacity to handle high data volumes.

For example, a test to find out how many client-nodes can simultaneously log into the server. Failures during scalability test includes the system not responding or system crashing. A product not able to respond to 100 concurrent users while it is supposed to serve 200 users simultaneously is a failure. For a given configuration, the following template may be used:

Given configuration:					RAM (512 MB) Cache (200 MB) No. of users: 100 Scalable parameter		
No. of records	**Start time**	**End time**	**Disk used**	**CPU used**	**Memory**	**Average time to add record**	**Server details**
0–10 thousand records 10–100 thousand records							

This data is collected for several configurations in these templates and analyzed.

In the above template, if disk utilization approaches 100% then another server is set up or a new disk is added. If successful, then repeat tests for a higher number of users but the aim is to find the maximum limit for that configuration. Increasing resources is not a silver bullet to achieve better scalability.

During scalability testing, the resources demand grows exponentially when the scalability parameter is increased. This scalability reaches a *saturation point* beyond which it cannot be improved. This is called as the *maximum capability of a scalability parameter*.

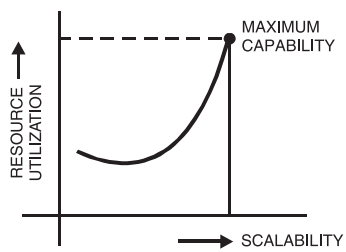


FIGURE 7.14

Guidelines for Scalability Testing:

1. Scalability should increase by 50% when the number of CPUs is doubled. This test is applicable for a CPU-intensive product.
2. Scalability should increase by 40% when memory is doubled. This test is applicable for memory intensive product.
3. Scalability should increase by 30%. When the number of NIC (network interface cards) are doubled. This task is useful for network-intensive products.
4. Scalability should increase by 50% when the I/O bandwidth is doubled. This test is useful for I/O intensive products.

We get a very important deliverable from this scalability testing, i.e., a *sizing guide*. It is a document containing timing parameters like OS parameters, other product parameters like number of open files, number of product threads, etc.

But it is expensive process.

7.2.5.5.5. Reliability testing

Reliability testing is done to find out if the software will work in the expected environment for an acceptable amount of time without degradation.

Executing reliability tests involves repeating the test cases for the defined operational profiles. The tests are executed when specific events occur like:

a. Major software releases

b. Completion of a time-box (time range)

c. At regular time intervals (say, weekly)

The test cases may already be fully specified or the required test data may be generated dynamically prior to execution. Please understand that the conditions for the test like test environment and test data should remain constant during the test. This is done to enable subsequent comparisons between execution cycles to be made. The achieved levels of reliability are reported after each test cycle. Tests relating to reliability growth can benefit from the following tools:

a. Tools for test data generation.

b. Test execution tools.

c. Code coverage tools for fault tolerance testing.

NOTE *These tools help identify the areas of code not yet exercised after performing functional tests.*

Reliability Testing for our Websites Involves:

i. Establish operational profiles for causal browsers and select 50 functional test cases each from an existing *test case database*.

ii. Choose 30 test cases at random from this database.

Probability Testing for Robustness Involves:

i. Perform *software technical reviews (STRs)* of code.

ii. Design *negative tests* with the help of test analysts.

iii. After delivery of each software release, do *exploratory testing*, i.e., to find defects in handling incorrect or unexpected inputs from the users.

iv. Memory leak and log-in and log-out operations are tested here.

Reliability Testing for Backup and Recovery Involves:

i. Verifying that a full backup can be restored within one hour of failure.

ii. Partial backup can be restored with in 30 minutes.

iii. Execute test case and verify that all data inconsistencies can be fully determined.

It is done to evaluate the product's ability to perform its required functions under stated conditions for a specified period of time or for a large number of iterations. For example, querying a database continuously for 48 hours and performing logic operations 10,000 times.

NOTE *The reliability of a product should not be confused with reliability testing.*

7.2.5.5.6. Stress testing

Stress testing is done to evaluate a system beyond the limits of specified requirements or resources to ensure that system does not break. It is done to find out if the product's behavior degrades under extreme conditions and, when it is desired, the necessary resources. The product is over-loaded

deliberately to simulate the resource crunch and to find out its behavior. It is expected to gracefully degrade on increasing the load but the system is not expected to crash at any point of time during stress testing.

It helps in understanding how the system can behave under extreme and realistic situations like insufficient memory, inadequate hardware, etc. System resources upon being exhausted may cause such situations. This helps to know the conditions under which these tests fail so that the maximum limits, in terms of simultaneous users, search criteria, large number of transactions, and so on can be known.

TIPS *Stress testing is a combination of several types of tests like capacity or volume testing, reliability and stability testing, error handling, and so on.*

Spike testing is a special sort of stress testing where an extreme load is suddenly placed on the system. If spikes are repeated with periods of low usage then it forms a bounce test. This is shown in Figure 7.15.

NOTE *Both spike and bounce tests determines how well the system behaves when sudden changes of loads occur.*

Two spikes together form a bounce test scenario. Then, the load increases into the stress area to find the system limits. These load spikes occur suddenly on recovery from a system failure.

There are differences between reliability and stress testing. Reliability testing is performed by keeping a constant load condition until the test case is completed. The load is increased only in the next iteration to the test case.

In stress testing, the load is generally increased through various means such as increasing the number of clients, users, and transactions until and beyond the resources are completely utilized. When the load keeps on increasing, the product reaches a stress point when some of the transactions start failing due to resources not being available. The failure rate may go up beyond this point. To continue the stress testing, the load is slightly reduced below this stress point to see whether the product recovers and whether the failure rate decreases appropriately. This exercise of increasing/decreasing the load is performed two or three times to check for consistency in behavior and expectations (see Figure 7.15).

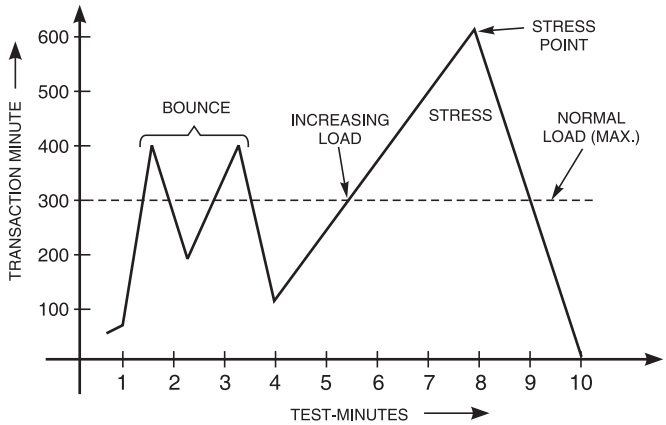


FIGURE 7.15

Sometimes, the product may not recover immediately when the load is decreased. There are several reasons for this. Some of the reasons are

1. Some transactions may be in the wait queue, delaying the recovery.
2. Some rejected transactions may need to be purged, delaying the recovery.
3. Due to failures, some clean-up operations may be needed by the product, delaying the recovery.
4. Certain data structures may have gotten corrupted and may permanently prevent recovery from stress point.

We can show stress testing with variable load in Figure 7.15.

Another factor that differentiates stress testing from reliability testing is mixed operations/tests. Numerous tests of various types run on the system in stress testing. However, the tests that are run on the system to create stress points need to be closer to real-life scenarios.

How to Select Test Cases for Stress Testing?

The following are the guidelines:

1. Execute repeated tests to ensure that at all times the code works as expected.
2. The operations that are used by multiple users are selected and performed concurrently for stress testing.

3. The operations that generate the amount of load needed are planned and executed for stress testing.
4. Tests that stress the system with random inputs (like number of users, size of data, etc.) at random instances and random magnitude are selected and executed as part of stress testing.

Defects that emerge from stress testing are usually not found from any other testing. Defects like memory leaks are easy to detect but difficult to analyze due to varying load and different types/ mix of tests executed. Hence, stress tests are normally performed after reliability testing. To detect stress-related errors, tests need to be repeated many times so that resource usage is maximized and significant errors can be noticed. This testing helps in finding out concurrency and synchronization issues like deadlocks, thread leaks, and other synchronization problems.

7.2.5.5.7. Interoperability testing

This type of testing is done to verify if SUT will function correctly in all the intended target environments. It includes:

a. Hardware
b. Software
c. Middle ware
d. OS
e. Network configurations

Any software is said to be good interoperable if it can be integrated easily with other systems without requiring major changes.

TIPS *The numbers and types of changes required to work in different environments determines the degree of interoperability of any software.*

The degree of interoperability is determined by the use of industry standards like XML. Please understand that the higher the degree of manual effort required to run on a different supported configuration, the lower is the interoperability of the software. Also, note that the so-called plug-and-play devices are good examples of highly interoperable software. This type of testing is done at the *integration level of testing*. Effective interoperability testing requires effective planning of the test lab, equipment, and

configurations. Once the configurations to be tested have been found, some companies use *shot gunning technique*. A method to distribute their test cases across the different configurations.

TIPS *Select those test cases that provide end-to-end functionality and run them.*

Can We Do Test Automations Here?

It is not easy in a multi-configuration environment but it can be done. We need a good test management system that can easily track the environment configuration used for test case execution. Let us solve an example now.

Example: The following testings are to be performed. Name the testing techniques that would be most appropriate for them. They are as follows:

a. Testing interfaces between product modules.

b. Testing information exchange between front-end (say, VB 6.0) and back-end (say, SQL server).

c. Testing product with other infrastructural pieces like OS, database, network.

d. Testing whether the API interfaces work properly.

Answers: These testings belong to:

a. Integration testing

b. Interoperability testing

c. Compatibility testing

d. Integration testing

Some guidelines to improve interoperability testing are as follows:

1. Consistent information flow across systems.
2. Appropriate messages should be given which the user must be aware of.
3. It is a collective responsibility.
4. It should be restricted to qualify the information exchange rather than finding defects and fixing them one by one.

7.2.5.6. ACCEPTANCE TESTING

It is a phase after system testing that is done by the customers. The customer defines a set of test cases that will be executed to qualify and accept the product. These test cases are executed by the customers are normally small in number. They are not written with the intention of finding defects. Testing in detail is already over in the component, integration, and system testing phases prior to product delivery to the customer. Acceptance test cases are developed by both customers and the product organization. Acceptance test cases are black-box type of tests cases. They are written to execute near real-life scenarios. They are used to verify the functional and non functional aspects of the system as well. If a product fails the acceptance test then it may cause the product to be rejected and may mean financial loss or may mean rework of product involving effort and time.

A user acceptance test is:

- A chance to complete test software.
- A chance to completely test business processes.
- A condensed version of a system.
- A comparison of actual test results against expected results.
- A discussion forum to evaluate the process.

The *main objectives* are as follows:

- Validate system set-up for transactions and user access.
- Confirm the use of the system in performing business process.
- Verify performance on business critical functions.
- Confirm integrity of converted and additional data.

The project team will be responsible for coordinating the preparation of all test cases and the acceptance test group will be responsible for the execution of all test cases.

Next, we discuss the acceptance test checklist. It is shown in the table below.

Tester	Component test	Test complete (pass/fail)
Tester #1	List browser/platform to use	
	End-to-end transaction	
	Verify billing	
	Verify log files on server	
	Etc.	
Tester #2	List browser/platform to use	
	End-to-end transaction	
	Help file	
	Remove book from shopping cart	
	Verify that high priority problems found during prior testing have been properly fixed	
Tester #n	List browser/platform to be used	
	Etc.	

7.2.5.6.1. Selecting test cases

The test cases for acceptance testing are selected from the existing set of test cases from different phases of testing.

Guidelines to Select Test Cases for Acceptance Testing

1. Test cases that include the end-to-end functionality of the product are taken up for acceptance testing. This ensures that all the business transactions are completed successfully. Real-life test scenarios are tested when the product is tested end-to-end.
2. Because acceptance tests focus on business scenarios, the product domain tests are included. Test cases that reflect business domain knowledge are included.
3. Acceptance tests reflect the real-life user scenario verification. As a result, test cases that portray them are included.

4. Tests that verify the basic existing behavior of the product are included.
5. When the product undergoes modifications or changes, the acceptance test cases focus on verifying the new features.
6. Some non functional tests are included and executed as part of acceptance testing.
7. Tests that are written to check if the product complies with certain legal obligations are included in the acceptance test criteria.
8. Test cases that make use of customer real-life data are included for acceptance testing.

7.2.5.6.2. Executing test cases

Acceptance testing is done by the customer or by the representative of the customer to check whether the product is ready for use in the real-life environment.

An acceptance test team usually comprises members who are involved in the day-to-day activities of the product usage or are familiar with such scenarios. The product management, support, and consulting teams contribute to acceptance testing definition and execution. A test team may be formed with 90% of them possessing the required business process knowledge of the product and 10% being representatives of the technical testing team.

Acceptance test team members are not aware of testing. Hence, before acceptance testing, appropriate training on the product and the process needs to be provided to the team. The acceptance test team may get the help of team members who developed/tested the software to obtain the required product knowledge. Test team members help the acceptance members to get the required test data, select and identify test cases, and analyze the acceptance test results. Test teams help the acceptance test team report defects. If major defects are identified during acceptance testing, then there is a risk of missing the release date.

All resolution of those defects and the unresolved defects are discussed with the acceptance test team and their approval is obtained for concluding the completion of acceptance testing.

7.2.5.6.3. Types of acceptance testing

There are two types of acceptance tests. They are as follows:

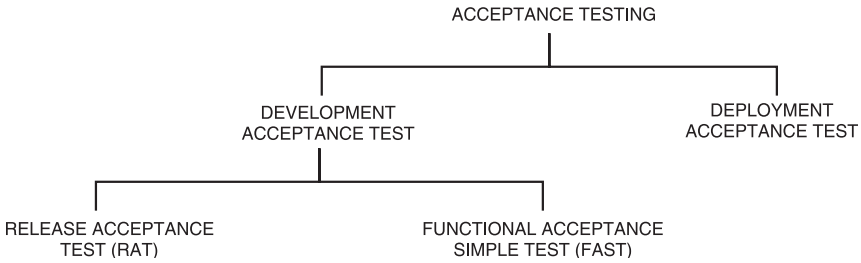


FIGURE 7.16

We shall discuss each of these tests one by one.

I. Development Acceptance Test

(a) Release acceptance test (RAT): It is also known as a build acceptance or smoke test. It is run on each development release to check that each build is stable enough for further testing. Typically, this test suite consists of entrance and exit test cases plus test cases that check mainstream functions of the program with mainstream data. Copies of RAT can be distributed to developers so that they can run the tests before submitting builds to the testing group. If a build does not pass a RAT test, it is reasonable to do the following:

- Suspend testing on the new build and resume testing on the prior build until another build is received.
- Report the failing criteria to the development team.
- Request a new build.

(b) Functional acceptance simple test (FAST): It is run on each development release to check that key features of the program are appropriately accessible and functioning properly on at least one test configuration (preferably minimum or common configuration). This test suite consists of simple test cases that check the lowest level of functionality for each command—to ensure that task-oriented functional tests (TOFTs) can be performed on the program. The objective is to decompose the functionality of a program down

to the command level and then apply test cases to check that each command words as intended. No attention is paid to the combination of these basic commands, the context of the feature that is formed by these combined commands, or the end result of the overall feature. For example, FAST for a File/SaveAs menu command checks that the SaveAs dialog box displays. However, it does not validate that the overall file-saving feature works nor does it validate the integrity of saved files.

Typically, errors encountered during the execution of FAST are reported through the standard issue-tracking process. Suspending testing during FAST is not recommended. Note that it depends on the organization for which you work. Each might have different rules in terms of which test cases should belong to RAT versus FAST and when to suspend testing or to reject a build.

II. Deployment Acceptance Test

The configurations on which the web system will be deployed will often be much different from develop-and-test configurations. Testing efforts must consider this in the preparation and writing of test cases for installation time acceptance tests. This type of test usually includes the full installation of the applications to the targeted environment or configurations. Then, FASTs and TOFTs are executed to validate the system functionality.

7.2.5.6.4. Acceptance testing for critical applications

In case of critical applications where a current system is going to be replaced by a new system, it may be prudent to do acceptance testing in three phases:

1. Acceptance testing: The usual testing for acceptance.
2. Parallel testing: Both the old and the new system are run concurrently, i.e., the old one as the main, live system and new one as the parallel offline system and the results of the two are compared each day. Once these operations are found to be satisfactory for a predetermined period of time, switch over is done to the third phase, i.e., reverse parallel.
3. Reverse parallel: We switch the roles in parallel testing. Make the new system the main, live system and the old system the offline system. Feed data in both. Compare results each day. When the results are found to be satisfactory for a predetermined period of time, discontinue the old system.

7.2.5.7. PERFORMANCE TESTING

The primary goal of performance testing is to develop effective enhancement strategies for mantaining acceptable system performance. It is an information gathering and analyzing process in which measurement data are collected to predict when load levels will exhaust system resources.

Performance tests use actual or simulated workload to exhaust system resources and other related problematic areas, including:

a. Memory (physical, virtual, storage, heap, and stack space)

b. CPU time

c. TCP/IP addresses

d. Network bandwidth

e. File handles

These tests can also identify system errors, such as:

a. Software failures caused by hardware interrupts

b. Memory runtime errors like leakage, overwrite, and pointer errors

c. Database deadlocks

d. Multithreading problems

7.2.5.7.1. Introduction

In this internet era, when more and more of business is transacted online, there is a big and understandable expectation that all applications will run as fast as possible. When applications run fast, a system can fulfill the business requirements quickly and put it in a position to expand its business. A system or product that is not able to service business transactions due to its slow performance is a big loss for the product organization, its customers, and its customer's customer. For example, it is estimated that 40% of online marketing for consumer goods in the US happens in November and December. Slowness or lack of response during this period may result in losses of several million dollars to organizations.

In another example, when examination results are published on the Internet, several hundreds of thousands of people access the educational websites within a very short period. If a given website takes a long time to complete the request or takes more time to display the pages, it may mean a

lost business opportunity, as the people may go to other websites to find the results. Hence, performance is a basic requirement for any product and is fast becoming a subject of great interest in the testing community.

Performance testing involves an extensive planning effort for the definition and simulation of workload. It also involves the analysis of collected data throughout the execution phase. Performance testing considers such key concerns as:

- Will the system be able to handle increases in web traffic without compromising system response time, security, reliability, and accuracy?
- At what point will the performance degrade and which components will be responsible for the degradation?
- What impact will performance degradation have on company sales and technical support costs?

Each of these preceding concerns requires that measurements be applied to a model of the system under test. System attributes, such as response time, can be evaluated as various workload scenarios are applied to the model. Conclusions can be drawn based on the collected data. For example, when the number of concurrent users reaches X, the response time equals Y. Therefore, the system cannot support more than X number of concurrent users. However, the complication is that even when the X number of concurrent users does not change, the Y value may vary due to differing user activities. For example, 1000 concurrent users requesting a 2K HTML page will result in a limited range of response times whereas response times may vary dramatically if the same 1000 concurrent users simultaneously submit purchase transactions that require significant server-side processing. Designing a valid workload model that accurately reflects such real-world usage is no simple task.

7.2.5.7.2. Factors governing performance testing

The testing performed to evaluate the response time, throughput, and utilization of the system, to execute its required functions in comparison with different versions of the same product(s) or different competitive product(s) is called performance testing. There are many factors that govern performance testing:

1. Throughput
2. Response time

3. Latency
4. Tuning
5. Benchmarking
6. Capacity planning

We shall discuss these factors one by one.

1. **Throughput.** The capability of the system or the product in handling multiple transactions is determined by a factor called throughput.

   It represents the number of requests/ business transactions processed by the product in a specified time duration. It is very important to understand that the throughput, i.e., the number of transactions serviced by the product per unit time varies according to the load the product is put under. This is shown in Figure 7.17.

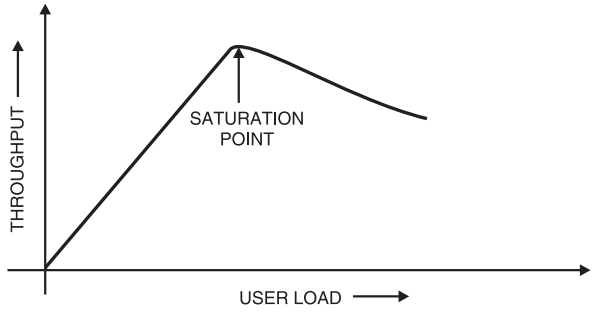


FIGURE 7.17 Throughput of a System at Various Loads.

   From this graph, it is clear that the load to the product can be increased by increasing the number of users or by increasing the number of concurrent operations of the product. Please note that initially the throughput keeps increasing as the user load increases. This is the ideal situation for any product and indicates that the product is capable of delivering more when there are more users trying to use the product. Beyond certain user load conditions (after the bend), the throughput comes down. This is the period when the users of the system notice a lack of satisfactory response and the system starts taking more time to complete business transactions. The "optimum throughput" is represented by the saturation point and is one that represents the maximum throughput for the product.

2. **Response time.** It is defined as the delay between the point of request and the first response from the product. In a typical client-server

environment, throughput represents the number of transactions that can be handled by the server and response time represents the delay between the request and response.

Also, note that it was mentioned earlier that customers might go to a different website or application if a particular request takes more time on this website or application. Hence, measuring response time becomes an important activity of performance testing.

3. **Latency.** It is defined as the delay caused by the application, operating system, and by the environment that are calculated separately. In reality, not all the delay that happens between the request and the response is caused by the product. In the networking scenario, the network or other products which are sharing the network resources can cause the delays. Hence, it is important to know what delay the product causes and what delay the environment causes.

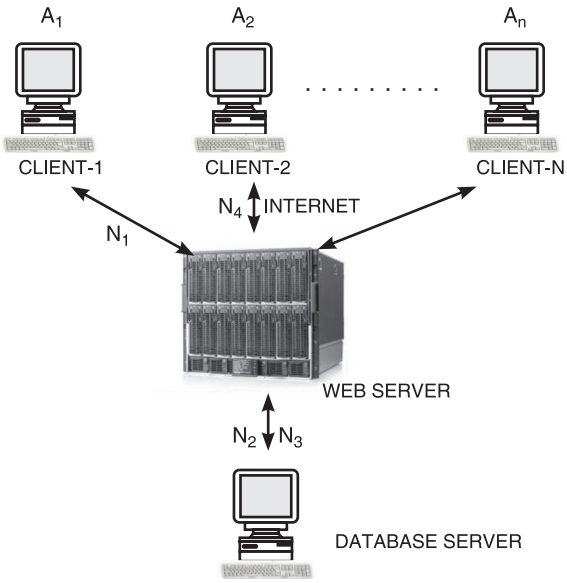


FIGURE 7.18 Latencies in 3-Tier Architecture.

To understand the latency, let us consider an example of a web application providing a service by talking to a web server and a database server connected in the network.

In this case, the latency can be calculated for the product that is running on the client and for the network that represents the

infrastructure available for the product. Thus, from the figure above, we can compute both the latency and the response time as follows:

$$\text{Network latency} = (N_1 + N_2 + N_3 + N_4)$$
$$\text{Product latency} = (A_1 + A_2 + A_3)$$
$$\text{Actual response time} = (\text{Network latency} + \text{Product latency})$$

The discussion about the latency in performance is very important, as any improvement that is done in the product can only reduce the response time by the improvements made in A_1, A_2, and A_3. If the network latency is more relative to the product latency and, if that is affecting the response time, then there is no point in improving the product performance. In such a case, it will be worthwhile to improve the network infrastructure. In those cases where network latency is too large or cannot be improved, the product can use intelligent approaches of caching and sending multiple requests in one packet and receiving responses as a bunch.

4. **Tuning.** Tuning is procedure by which the product performance is enhanced by setting different values to the parameters (variables) of the product, operating system, and other components. Tuning improves the product performance without having to touch the source code of the product. Each product may have certain parameters or variables that can be set at run time to gain optimum performance. The default values that are assumed by such product parameters may not always give optimum performance for a particular deployment. This necessitates the need for changing the values of parameters or variables to suit the deployment or a particular configuration. During performance testing, tuning of the parameters is an important activity that needs to be done before collecting numbers.

5. **Benchmarking.** It is defined as the process of comparing the throughput and response time of the product to those of the competitive products. No two products are the same in features, cost, and functionality. Hence, it is not easy to decide which parameters must be compared across two products. A careful analysis is needed to chalk out the list of transactions to be compared across products. This produces meaniningful analysis to improve the performance of the product with respect to competition.

6. **Capacity planning.** The most important factor that affects performance testing is the availability of resources. A right kind of hardware and software configuration is needed to derive the best results from

performance testing and for deployments. This exercise of finding out what resources and configurations are needed is called capacity planning. The purpose of capacity planning is to help customers plan for the set of hardware and software resources prior to installation or upgrade of the product. This exercise also sets the expectations on what performance the customer will get with the available hardware and software resources.

Summary: Performance testing is done to ensure that a product:

1. processes the required number of transactions in any given interval (throughput).
2. is available and running under different load conditions (availability).
3. responds fast enough for different load conditions (response time).
4. delivers worth while ROI for resources—hardware and software. Also decides what kind of resources are needed for the product for different load conditions (i.e., capacity planning).
5. is comparable to and better than that of the competitors for different parameters (competitive analysis and benchmarking).

We next consider a simple example to understand how increased traffic load and consequently, increased response time can result in lost company revenue.

EXAMPLE 7.1. Suppose your e-commerce site currently handles 300,000 transactions per day. How many transactions are being done per second? How can performance testing be done?

SOLUTION. Transactions per day = 300,000

$$\therefore \quad \text{Transactions per second} = \frac{300{,}000}{24.60.60}$$

$$= 3.47 \text{ transactions per second}$$

After conducting a marketing survey, the findings show that:

a. The transaction response time is of an acceptable level as long as it does not exceed 4 seconds.

b. If the transaction response time is greater than 4 but less than 9 seconds, 30% of users cancel their transactions.

c. If the transaction response time is greater than 8 but less than 10 seconds, 60% of users cancel their transactions.

d. If the transaction response time increases to over 10 seconds, over 90% of users cancel their transactions.

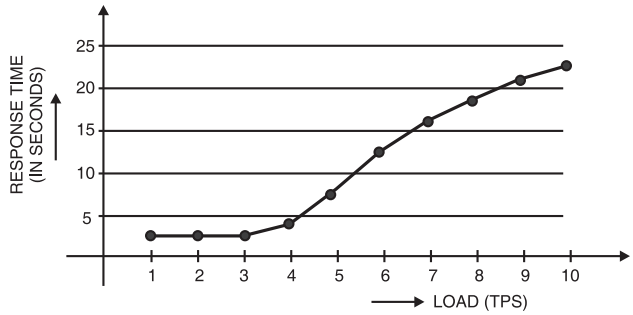


FIGURE 7.19

Suppose in the next 6 months, the number of transactions is expected to rise between 25% and 75% from the current level and the potential revenue for each transaction is $1. Management would like to learn how the performance would impact company revenue as the number of transactions per day increases.

Refer to Figure 7.19 for detailed analysis of traffic, percentage, and dollar amounts.

A performance test is conducted and the findings show that the system cannot handle such increases in traffic without increasing response time, user transaction cancellations, and/or failure results. If the number of transactions increases as expected, the company will face a potential revenue loss of between $112,500 and $472,500 per day.

It takes time, effort, and commitment to plan for and execute performance testing. Performance testing involves individuals from many different departments. A well planned testing program requires the coordinated efforts of members of the product team, upper management, marketing, development, information technology (IT), and testing. Management's main objective in performance testing should be to avoid financial losses due to lost sales, technical support issues, and customer dissatisfaction.

7.2.5.7.3. Steps of performance testing

A methodology for performance testing involves the following steps:

Step 1. Collecting requirements.
Step 2. Writing test cases.
Step 3. Automating performance test cases.

Step 4. Executing performance test cases.
Step 5. Analyzing performance test results.
Step 6. Performance tuning.
Step 7. Performance benchmarking.
Step 8. Capacity planning.

We will explain each of these steps one by one.

(I) Collecting Requirements

Performance testing generally needs elaborate documentation and environment setup and the expected results may not be well known in advance.

Challenges of Performance Testing

a. A performance testing requirement should be testable. All features/functionality cannot be performance tested.
For example, a feature involving a manual intervention cannot be performance tested as the results depend on how fast a user responds with inputs to the product.

b. A performance testing requirement needs to clearly state what factors need to be measured and improved.

c. A performance testing requirement needs to be associated with the actual number or percentage of improvement that is desired.

There are two types of requirements that performance testing focuses on:

1. Generic requirements.
2. Specific requirements.

1. **Generic requirements** are those that are common across all products in the product domain area. All products in that area are expected to meet those performance expectations.

   Examples are time taken to load a page, initial response when a mouse is clicked, and time taken to navigate between screens.

   Specific requirements are those that depend on implementation for a particular product and differ from one product to another in a given domain.

   An example is the time taken to withdraw cash from an ATM.

   During performance testing both generic and specific requirements need to be tested.

   See Table in next page for examples of performance test requirements.

Transaction	Expected response time	Loading pattern or throughput	Machine configuration
1. ATM cash withdrawal	2 sec	Up to 10,000 simultaneous access by users	P-IV/512 MB RAM/ broadband network
2. ATM cash withdrawal	40 sec	Up to 10,000 simultaneous access by users	P-IV/512 MB RAM/ dial-up network
3. ATM cash withdrawal	4 sec	More than 10,000 but below 20,000 simultaneous access by users	P-IV/512 MB RAM/ broadband network

A performance test conducted for a product needs to validate this graceful degradation as one of the requirement.

(II) Writing Test Cases

A test case for performance testing should have the following details defined:

1. List of operations or business transactions to be tested.
2. Steps for executing those operations/transactions.
3. List of product, OS parameters that impact the performance testing and their values.
4. Loading pattern.
5. Resources and their configurations (network, hardware, software configurations).
6. The expected results, i.e., expected response time, throughput, latency.
7. The product versions/competitive products to be compared with related information such as their corresponding fields (Steps 2-6 in the above list).

Performance test cases are repetitive in nature. These test cases are normally executed repeatedly for different values of parameters, different load conditions, different configurations, and so on. Hence, the details of what tests are to be repeated for what values should be part of the test case documentation.

While testing the product for different load patterns, it is important to increase the load or scalability gradually to avoid any unnecessary effort in case of failures.

For example, if an ATM withdrawal fails for ten concurrent operations, there is no point in trying it for 10,000 operations. The effort involved in

testing for 10 concurrent operations may be less than that of testing for 10,000 operations by several times. Hence, a methodical approach is to gradually improve the concurrent operations by say 10, 100, 1000, 10,000, and so on rather than trying to attempt 10,000 concurrent operations in the first iteration itself. The test case documentation should clearly reflect this approach.

Performance testing is a tedious process involving time and effort. All test cases of performance testing are assigned different priorities. Higher priority test cases are to be executed first. Priority may be *absolute* (given by customers) or may be *relative* (given by test team). While executing the test cases, the absolute and relative priorities are looked at and the test cases are sequenced accordingly.

(III) Automating Performance Test Cases

Automation is required for performance testing due to the following characteristics:

1. Performance testing is repetitive in nature.
2. Performance test cases cannot be effective without automation.
3. The results of performance testing need to be accurate. Accuracy will be less if response-time, throughput, etc. are calculated manually.
4. Performance testing involves several factors and their permutations and combinations. It is not easy to remember all of these and use them manually during testing.
5. Performance testing collects various details like resource utilization, log files, trace files, and so on at regular intervals. It is not possible to do book keeping of all this related information manually.

There should not be any hard coded data in automated scripts for performance testing as it affects the repeatability nature of test cases.

The set up required for the test cases, setting different values to parameters, creating different load conditions, setting up and executing the steps for operations/transactions of competitive products, and so on have to be included as the part of the automation script.

While automating performance test cases, it is important to use standard tools and practices. Some of the performance test cases involve comparisons with the competitive product, so the results need to be consistent, repeatable, and accurate.

(IV) Executing Performance Test Cases

Performance testing involves less effort for execution but more effort for planning, data collection, and analysis. As explained earlier, 100% end-to-end automation is desirable for performance testing.

During the execution of performance testing, the following data needs to be collected:

1. Start and end time of test case execution.
2. Log and trace (or audit) files of the product and OS.
3. Utilization of resources like CPU, memory, disk, and network on a periodic basis.
4. Configuration of hardware and software envionmental factors.
5. The response time, throughput, latency, etc., as specified in the test case documentation at regular intervals.

Scenario testing is also done here. A scenario means a set of transaction operations that are usually performed by the user. It is done to ensure whether the mix of operations/transactions concurrently by different users/machines meets the performance criteria.

Configuration performance testing is also done to ensure that the performance of the product is compatible with different hardware. The tests need to be repeated for different configurations. For a given configuration, the product has to give the best possible performance and if the configuration is better, it has to get even better. For example, we show a sample configuration performance test.

TABLE 7.1 Sample Configuration Performance Test.

Transaction	No. of users	Test environment
Querying ATM account balance	20	RAM 512 MB, P-IV dual processor, WIN-NT server (OS)
ATM cash withdrawal	20	RAM 128 MB, P-IV dual processor, WIN-98 (OS)
ATM user profile query	40	RAM 256 MB, P-IV quad processor, WIN-2000 (OS)

The performance test case is repeated for each row in this table and factors such as the response time and throughput are recorded and analyzed.

After the execution of performance test cases, various data points are collected and the graphs are plotted. For example, the response time graph is shown below:

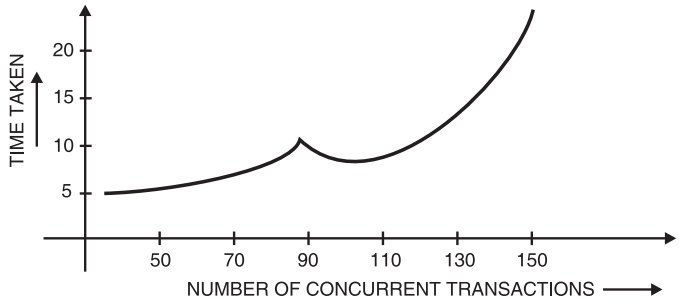


FIGURE 7.20(A) Response Time.

Similarly, for throughput, we can draw:

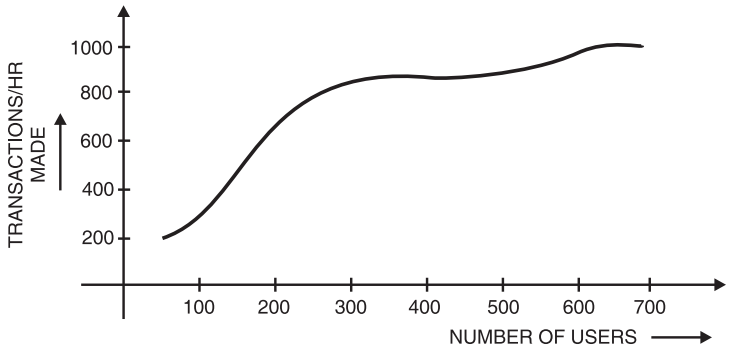


FIGURE 7.20(B) Throughput.

Plotting the data helps in making an easy and quick analysis which is difficult with only raw data.

(V) Analyzing the Performance Test Results

It is a multi-dimensional thinking part of performance testing. The product knowledge, analytical thinking, and statistical background are all absolutely essential.

Before data analysis, some calculations of data and its organization are required. They are:

1. Calculating the mean of the performance test result data.
2. Calculating the standard deviation.

3. Removing noise and replotting and recalculating the mean and standard deviation.

They are explained below:

The performance numbers are to be reproducible for the customers. To ensure that all performance tests are repeated multiple times, the average or mean of those values are taken. Thus, the probability of performance data for reproducibility at a customer site also increases for the same configuration and load condition.

The standard deviation represents how much the data varies from the mean. For example, if the average response time of 100 people withdrawing money from an ATM is 100 seconds and the standard deviation is 2, then there as a greater chance that this performance data is repeatable than in a case where the standard deviation is 30. A standard deviation close to zero means the product performance is highly repeatable and performance values are consistent. The higher the standard deviation, the more the variability of the product performance.

When a set of values is plotted on a chart, one or two values that are out of range may cause the graph to be cluttered and prevent meaningful analysis. Such values can be ignored to produce a smooth curve/graph. The process of removing some unwanted values in a set is called noise removal. Because some values are removed from the set, the mean and standard deviation need to be re-calculated.

Also, note that the majority of client server, Internet, and database applications store data in a local high-speed buffer when a query is made. This is called caching. The performance data need to be differentiated according to where the result is coming from—the server or the cache. So, the data points can be kept as two different sets—one for the cache and one from the server. This helps in the extrapolation of performance data. For example, let us assume that data in a cache can produce a response time of 100 μs and a server access takes 2 μs and 80% of the time a request is satisfied by the cache. Then, the average response time is given by

$$= (0.8) * 100 + 0.2 * 2$$
$$= 80 + 0.4\ \mu s = 80.4\ \mu s$$

Then, the mean response time is computed as a weighted average rather than a simple mean.

Other factors such as garbage collection/defragmentation in the memory of an OS or compiler also affect the performance data.

(VI) Performance Tuning

Once the parameters that affect performance testing are minimized, we can repeat performance test cases to further analyze their effect in getting better performance. This exercise is known as performance tuning and it requires skills in identifying such parameters and their contribution to performance and the relationship among these parameters is equally important for performance tuning.

There are two steps involved here:

Step 1. Tuning the product parameters.
Step 2. Tuning the operating system and parameters.

The first step involves a set of parameters associated with the product where users of the product can set different values to obtain optimum performance. Some of the factors are:

a. Providing a number of forked processes for parallel transactions.

b. Caching.

c. Disabling low-priority operations.

d. Creating background activities.

e. Deferring routine checks to a later point of time.

f. Changing the sequence or clubbing a set of operations and so on.

Setting different values to these parameters enhances the product performance.

Some do's are listed below:

1. Repeat the performance tests for different values of each parameter that impact performance.
2. Repeat the performance tests for a group of parameters and their different values.
3. Repeat the performance tests for the default values of all parameters. This is called factory settings tests.
4. Repeat the performance tests for low and high values of each parameter and combinations.

Please note that performance tuning provides better results only for a particular configuration and for certain transactions. Therefore, tuning may

be counter productive to other situations or scenarios. This side effect of tuning product parameters needs to be analyzed.

The second step involves the tuning of OS parameters. These parameters can be:

1. File system-related parameters like the number of open files allowed.
2. Disk management parameters like simultaneous disk reads/writes.
3. Memory management parameters like virtual memory page size and number of pages.
4. Processor management parameters like enabling/disabling processors in a multiprocessor environment.
5. Network parameters like setting TCP/IP time out.

Please note that the OS parameters need to be tuned only when the complete impact is known to all applications running in the machine. They should be tuned only if they are absolutely necessary.

The results of the performance tuning are published in the form of a guide called the performance tuning guide for customers so that they can benefit from this exercise. It explains in detail the effect of each product and OS parameter on performance. It also gives a set of guideline values for the combination of parameters and what parameter must be tuned in and in which situation.

(VII) Performance Benchmarking

Performance benchmarking is the process of comparing the performance of product transactions to that of the competitors. No two products can have the same architecture, design, functionality, and code. Hence, it will be very difficult to compare two products on those aspects. End-user transactions/scenarios could be one approach for comparison.

In general, an independent test team or an independent organization not related to the organizations of the products being compared does performance benchmarking.

The steps involved in performance benchmaking are:

1. Identify the transactions/scenarios and the test configuration.
2. Compare the performance of different products.
3. Tune the parameters.
4. Publish the results of performance benchmarking.

Generally, the test cases for all products being compared are executed in the same test bed. Next step is to compare the results. A well-tuned

product, X, may be compared with a product B with no parameter tuning to prove that product A performs better than B. It is important in performance benchmarking that all products should be tuned to the same degree.

There could be three outcomes from performance benchmarking.

1. **Positive outcome.** A set of transactions/scenarios out-perform with respect to competition.
2. **Neutral outcome.** A set of transactions are comparable with those of the competition.
3. **Negative outcome.** A set of transactions under-perform compared to those of the competition.

Tuning can be repeated for all situations of positive, neutral, and negative results to derive the best performance results.

The results of performance benchmaking are published. Two types of publications are involved. They are:

a. An internal, confidential publication to product teams containing all three outcomes given above and the recommended set of actions.

b. The positive outcomes are normally published as marketing collateral—which helps as a sales tool for the product.

Also benchmarks conducted by the independent organizations are published as audited benchmarks.

(VIII) Capacity Planning

The planning of load generation capacity does not stop with the specification of required hardware. We should also consider our network's capacity to transport a large number of transactions and a huge amount of data. Both our hardware and network bandwidth must be sufficient. This can cost a bit more. It is the process in which the performance requirements and the performance results are taken as input requirements and the configuration needed to satisfy that set of requirements are derived. Thus, capacity planning is the reverse process.

It necessitates a clear understanding of the resource requirement for transactions/scenarios.

Some transactions of the product associated with certain load conditions could be

a. Disk intensive

b. CPU intensive

c. Network intensive

d. Memory intensive

Some transactions may require a combination of these resources for performing better. This understanding of what resources are needed for each transaction is a prerequisite for capacity planning.

It is critical to consider some requirements during capacity planning. The load can be

a. Short term, i.e., actual requirements of the customer for immediate need.

b. Medium term, i.e., requirements for the next few months.

c. Long-term, i.e., requirements for the next few years.

Capacity planning corresponding to short-, medium-, and long-term requirements are called

1. Minimum required configuration

2. Typical configuration

3. Special configuration

A *minimum-required configuration* denotes that with anything less than this configuration, the product may not even work. Thus, configurations below the minimum-required configuration are usually not supported.

A *typical configuration* denotes that under that configuration the product will work fine for meeting the performance requirements of the required load pattern and can also handle a slight increase in the load pattern.

A *special configuration* denotes that capacity planning was done considering all future requirements.

There are two techniques that play a vital role in capacity planning. They are as follows:

a. Load balancing

b. High availability

Load balancing ensures that the multiple machines available are used equally to service the transactions. This ensures that by adding more machines more load can be handled by the product. Machine clusters are used to ensure availability. In a cluster, there are multiple machines with shared data so that in case one machine goes down, the transactions can be handled by another machine in the cluster.

Capacity planning is based on the performance test data generated in the test lab which is only a simulated environment. In real-life deployment, there could be several other parameters that may impact product performance.

7.2.5.7.4. Tools for performance testing

There are two types of tools.

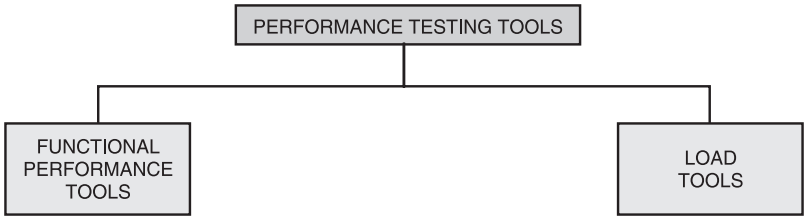


FIGURE 7.21

Functional performance tools help in recording and playing back the transactions and obtaining performance numbers. This test generally involves very few machines.

Load testing tools simulate the load condition for performance testing without having to keep that many users or machines. These tools simplify the complexities involved in creating the load. This is only a simulated load and real-life experience may vary from the simulation. Some popular performance tools are:

I. Functional performance tools
- Win Runner (Mercury-Vendor)
- QA partner (Compuware-Vendor)
- Silk Test from Segue.

II. Load testing tools
- Load Runner (Mercury-Vender)
- QA load (Compuware-Vendor)
- Silk Performer (segue)

These tools can help in getting performance numbers only. The utilization of resources is an important parameter that needs to be collected. For example, "windows task manager" and "top in linux" are such tools. Network performance monitoring tools are available with almost all OS today to collect network data.

7.2.5.7.5. Performance testing challenges

1. The availability of required skills is one of the major problems facing performance testing.
2. The comparison of functional performance numbers with load testing numbers becomes difficult as the build used and time-lines are also different as they were performed in two different phases.

3. Performance testing requires a large number and amount of resources such as hardware, software, effort, time, tools, and people. This is another challenge.
4. It needs to reflect real-life environment and expectations. Adequate care to create a test bed as close to a customer deployment is another expectation for performance tests.
5. Selecting the right tool for the performance testing is another challenge. These tools are expensive. They also expect the test engineers to learn additional meta-languages and scripts.
6. Interfacing with different teams that include a set of customers is another challenge.
7. Lack of seriousness on performance tests by the management and development team is another challenge.

SUMMARY

We can say that we start with unit or module testing. Then we go in for integration testing, which is then followed by system testing. Then we go in for acceptance testing and regression testing. Acceptance testing may involve alpha and beta testing while regression testing is done during maintenance.

System testing can comprise of "n" different tests. That is it could mean:

1. End-to-end integration testing
2. User interface testing
3. Load testing in terms of
   a. Volume/size
   b. Number of simultaneous users
   c. Transactions per minute/second (TPM/TPS)
4. Stress testing
5. Testing of availability (24 × 7)

Performance testing is a type of testing that is easy to understand but difficult to perform due to the amount of information and effort needed.

MULTIPLE CHOICE QUESTIONS

1. Alpha testing involves
   - **a.** Customers
   - **b.** Testers
   - **c.** Developers
   - **d.** All of the above.
2. Site for alpha testing is
   - **a.** Software development company
   - **b.** Installation site
   - **c.** Anywhere
   - **d.** None of the above.
3. Site of beta testing is
   - **a.** Software organization
   - **b.** Customer's site
   - **c.** Anywhere
   - **d.** All of the above.
4. Acceptance testing is done by
   - **a.** Developers
   - **b.** Customers
   - **c.** Testers
   - **d.** All of the above.
5. Integration testing techniques are
   - **a.** Top down
   - **b.** Bottom up
   - **c.** Sandwich
   - **d.** All of the above.
6. Top-down approach is used for
   - **a.** Development
   - **b.** Identification of faults
   - **c.** Validation
   - **d.** Functional testing
7. Which of the following validation activities belong to low-level testing:
   - **a.** Unit testing
   - **b.** Integration testing
   - **c.** System testing
   - **d.** Both (a) and (b).
8. Testing done to ensure that the software bundles the required volume of data is called
   - **a.** Load testing
   - **b.** Volume testing
   - **c.** Performance testing
   - **d.** Stress testing

9. Number of sessions for integration testing is given by

a. Sessions = nodes + leaves – edges

b. Sessions = nodes – leaves – edges

c. Sessions = nodes – leaves + edges

d. None of the above.

10. An MM-path graph refers to

a. Module-to-module path graph

b. Manager-to-manager path graph

c. Many-module path graph

d. None of the above.

ANSWERS

1. a.	**2.** a.	**3.** b.	**4.** b.
5. d.	**6.** b.	**7.** d.	**8.** b.
9. c.	**10.** a.		

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

Q. 1. Some scenarios have a low probability but a high potential impact. How to tackle such scenarios?

Ans. This can be accomplished by adding a weight to each scenario as follows:

Weight	Meaning
+2	Must test, mission/safety critical
+1	Essential functionality, necessary for robust operation
+0	All other scenarios

Q. 2. How will the operational profile maximize system reliability for a given testing budget?

Ans. Testing driven by an operational profile is very efficient because it identifies failures on average, in order of how often they occur. This approach rapidly increases reliability—reduces failure intensity per unit of execution time because the failures that occur most frequently are caused by the faulty operations used most frequently. According to J.D. Musa, users will also detect failures in order of their frequency, if they have not already been found in the test.

Q. 3. What are three general software performance measurements?

Ans. The three general software performance measurements are as follows:

a. **Throughput:** The number of tasks completed per unit time. It indicates how much of work has been done in an interval. It does not indicate what is happening to a single task.

   Example: Transactions per second.

b. **Response time:** The time elapsed between input arrival and output delivery. Here, average and worst-case values are of interest.

   Example: Click to display delay.

c. **Utilization:** The percentage of time a component is busy. It can be applied to processor resources like CPU, channels, storage, or software resources like file-server, transaction dispatcher, etc.

   Example: Server utilization.

Q. 4. What do you understand by system testing coverage?

Ans. System testing is requirements driven. The TC coverage metric reflects the impact of requirements. [IEEE 89a]

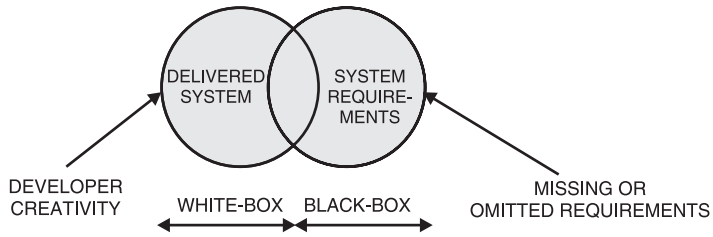


FIGURE 7.22

$$\therefore \quad TC = \left(\frac{\text{Implemented capabilities}}{\text{Required capabilities}}\right) \times \left(\frac{\text{Total components tested}}{\text{Total no. of components}}\right) \times 100$$

Q. 5. What sort of verifications are required during system testing?

Ans. The following tests are suggested:

a. **Conformity test:** One of the major tasks during system testing is to cover all requirements to ensure that all types of system requirements are exercised to ascertain their conformity.

b. **Data coverage test:** In this test, we select test data that cover a broad range of usages. It employs important data values like valid and invalid data and boundary values to test the requirements in the first step.

Q. 6. Compare unit and integration testing in a tabular form.

Ans.

Unit testing	Integration testing
1. It starts from the module specification. **2.** It tests the visibility of code in detail. **3.** It requires complex scaffolding. **4.** It checks the behavior of single modules.	**1.** It starts from the interface specification. **2.** It tests the visibility of the integration structure. **3.** It requires some scaffolding. **4.** It checks the integration among modules.

Q. 7. Compare integration and system testing in a tabular form.

Ans.

Integration testing	System testing
1. It starts from the interface specification. **2.** It tests the visibility of the integration structure. **3.** It requires some scaffolding. **4.** It checks the integration among modules.	**1.** It starts from the requirements specification. **2.** It does not test the visibility of code. **3.** It does not require any scaffolding. **4.** It checks the system functionality.

Q. 8. Consider hypothetical "online railway reservation system." Write suitable scope statement for the system. Write all assumptions and identify two test cases for each of the following:

i. Acceptance testing

ii. GUI testing

iii. Usability and accessibility testing

iv. Ad hoc testing

Document test cases.

Ans. Test cases for online railway reservation system:
Preconditions: Open web browser and enter URL in the address bar. Homepage must be displayed.

Test case ID	Test case name	Test case description	Test steps		Actual result	Test status (P/F)
			Step	Expected result		
AccTest01	Seat Availability	Verify whether the seats are available or not	Enter: • Train number 0340 • Date of journey: July 11, 2009 • Source code: MUM • Destination code; DEL • Class: Sleeper • Quota: General	Should display accommodation availability enquiry		
			Enter: • Train number: a @1% • Date of journey: July 11, 2009 • Source code: MUM • Destination code: DEL • Class: Sleeper • Quota: General	Displays the message: Invalid train number		
	Get Fare details	Verify the working of Get Fare	Click on Get Fare option	System displays a table of fare stating base fare, reservation charges with class		
AccTest0	Reservation	Verify reservation procedure	If seats are available click on Get Availability	System displays reservation form		
			Enter passenger details and train details	System accepts the valid details and displays the confirmation form		

(*Continued*)

Test case ID	Test case name	Test case description	Test steps		Actual result	Test status (P/F)
			Step	Expected result		
GUI01	Verify aesthetic conditions	Is the general screen background the correct color?	Open the browser and type URL of the site	Background color is proper making foreground information visible		
		Is the screen resizable and minimizable?	Open the site, resize and minimize the same	System gets resized and minimized		
GUI02	Verify validation condition	Does a failure of validation of every field cause a sensible user error message?	Open Fare Enquiry for a Train and feed following details: Train no. asd	System displays error message: Invalid train number! Try again!		
Usability_ accessibil-ity01	Verify Usability	Does the homepage load quickly?	Open the browser and type URL of the site	System displays a homepage quickly		
	Interactivity	To verify whether interaction mechanisms (pull-down menus) are easy to use	Open Fare Enquiry for a Train and check different categories of train class	Class pull-down menu displays various categories of class		
Usability_ accessibil-ity01	Navigation	Does each hyperlink work on each page?	Open the homepage and try the navigations for: • Train schedule • Trains • Fare • Seat Availability	System redirects to respective pages		
Adhoc01	Verify the functionality	Functionality of Train schedule	Open the train schedule page and type train number	System displays the proper train route from source to destination along with arrival and departure time		

(*Continued*)

Test case ID	Test case name	Test case description	Test steps		Actual result	Test status (P/F)
			Step	Expected result		
			Open the train schedule page and type train number and change in the train name	System does not allow the user to change the train name		
Adhoc02	Navigation	Is a nagivational bar and link to home present on every screen?	Open the homepage and navigate the system	Navigational bar and home link is present on every screen		
		Are fonts too large or small?	Load the homepage	Fonts are proper		

REVIEW QUESTIONS

1. Differentiate between alpha and beta testing?
2. Explain the following: Unit and Integration testing?
3. a. What would be the test objective for unit testing? What are the quality measurements to ensure that unit testing is complete?
   b. Put the following in order and explain in brief:
      i. System testing
      ii. Acceptance testing
      iii. Unit testing
      iv. Integration testing
4. Explain integration and system testing.
5. Write a short paragraph on levels of software testing.

6. **a.** Why is threaded integration considered an incremental testing technique?

   b. Given the module order in call graph:

   $A \rightarrow B, C$
   $B \rightarrow D, E, F$
   $C \rightarrow F, G$
   $F \rightarrow H$

   Give the integration testing order for bottom-up testing.

7. Describe all methods of integration testing.

8. What do you mean by "big bang" integration strategy?

9. What do you mean by unit testing? Discuss how you would perform unit testing of a module that implements bounded stack of 100 integers that you have designed and coded. Assume that the stack functions supported are push and pop.

10. What do you understand by the term integration testing? What are its different methods? Compare the relative merits and demerits of these different integration testing strategies.

11. What are drivers and stubs? Why are they required?

12. Consider a piece of an embedded software that is part of a TV. Which of the types of system testing discussed would you choose to apply and at what times?

13. Each product has different tuning parameters. For each of the following cases, identify the important tuning parameters:

    a. OS (e.g., windows XP)

    b. Database (e.g., oracle 11i)

    c. A network card (e.g., a wireless LAN card)

14. What are the differences in the requirements gathering for performance testing and for functional testing like black-box testing? Discuss on the basis of sources used, methods used, tools used, and skill sets required.

15. "Staffing people for performance testing is most difficult." Justify.

16. **a.** Explain how you test the integration of two fragment codes with suitable examples.

    b. What are the various kinds of tests we apply in system testing? Explain.

17. Assume that you have to build the real-time multiperson computer game. What kinds of testing do you suggest or think are suitable. Give a brief outline and justification for any four kinds of tests.

18. Discuss some methods of integration testing with examples.

19. **a.** What is the objective of unit and integration testing? Discuss with an example code fragment.

    b. You are a tester for testing a large system. The system data model is very large with many attributes and there are many interdependencies within the fields. What steps would you use to test the system and what are the effects of the steps you have taken on the test plan?

20. What is the importance of stubs? Give an example.

21. **a.** Explain BVT technique.

    b. Define MM-path graph. Explain through an example.

    c. Give the integration order of a given call graph for bottom-up testing.

    d. Who performs offline deployment testing? At which level of testing it is done?

    e. What is the importance of drivers? Explain through an example.

22. Which node is known as the transfer node of a graph?

23. **a.** Describe all methods of integration testing.

    b. Explain different types of acceptance testing.

24. Differentiate between integration testing and system testing.

25. **a.** What are the pros and cons of decomposition based techniques?

    b. Explain call graph and path-based integration testing. Write advantages and disadvantages of them.

    c. Define acceptance testing.

    d. Write a short paragraph on system testing.

CHAPTER 8

QUALITY ASSURANCE

Inside this Chapter:

8.0. Introduction
8.1. Quality Planning
8.2. Quality Plan Objectives
8.3. Planning Process Overview
8.4. Business Plan and Quality Plan
8.5. TQM
8.6. TQM Concepts
8.7. Zero Defect Movement

8.0. INTRODUCTION

The term *Quality Assurance* or *QA* means to ensure that the software conforms to prescribed technical requirements. It is essentially an audit function. It does not prescribe good or complete testing. QA is more than finding bugs during the time left until the deadline following exactly some detailed test plans. It is more than defect detection. *Software quality assurance (SQA)* is a planned and systematic pattern of actions required to ensure quality in software. The objective of SQA is to improve software quality by monitoring both software and the development process to ensure full compliance with the established standards and procedures. SQA benefits the software development project by saving time and money.

Actually, QA and, hence, total quality management have evolved as shown in Figure 8.1.

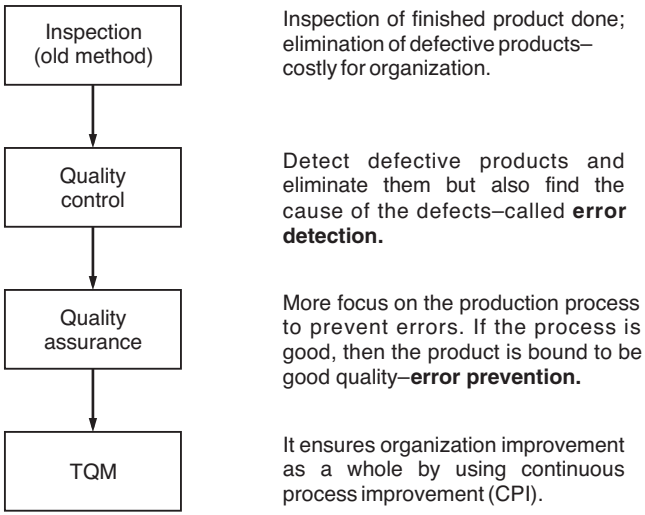


FIGURE 8.1 Evolution of QA.

8.1. QUALITY PLANNING

The relevant quality standards are determined for the project. This is an important aspect of project planning. Based on the inputs captured in the previous steps such as the project scope, requirements, deliverables, etc. Various factors influencing the quality of the final product are determined. The processes required to deliver the product as promised and as per the standards are defined.

An effective assurance program requires planning and follow through. The objective of creating a software assurance plan is to document/specify the conduct of the activities that will comprise software assurance for a specific project. A useful guide for documenting assurance plans is provided in the assurance sections of the SMAP management plan documentation standard.

Keep the following points in mind:

1. Plan software assurance along with the management and engineering plan.
2. Phase assurance activities properly. For example, design standards must be produced before design starts.
3. Procure or develop CASE tools beforehand only.

A complete SQA plan must now be implemented. Well-trained and qualified staff must be obtained. All of the management activities needs assurance staff as a resource to help complete them.

8.2. QUALITY PLAN OBJECTIVES

The major objectives of the SQA process are to ensure that the software development and software assurance processes comply with SQA plans and standards. To aim is to recommend, process improvements. SQA activities include *product evaluation* and *process monitoring* which ensure the product and the process used in the development are correctly carried out and standards are followed. The quality assurance process uses the system requirements and information about the purpose and criticality of the software to evaluate the outputs of the software development and software assurance processes.

NOTE *Audit is a fundamental SQA technique.*

It looks at a process or product in depth comparing them with established standards and procedures. SQA includes the conduct of FTRs using technical methods, software testings, standards enforcement, change control, metric and record keeping, and reporting.

8.3. PLANNING PROCESS OVERVIEW

To win top-level management's favor, a software quality program (plan) is essential. It involves the following steps:

1. **Preparation of the SQA plan:** It specifies its goals and tasks to be performed and the standards and procedures against which the development work is to be measured.
2. **SQA monitoring and controlling:** It deals with the configuration management monitoring, V & V monitoring, formal test monitoring, and FTRs.
3. The effective error detection will be ensured by the *testing phase.*

4. The appropriate standards useful to the products and process will be defined and illustrated in the phase setting of standard and procedures.
5. Finally, the development of relevant metrics and models will be discussed in the development and collection of relevant metrics and model phase.

8.4. BUSINESS PLAN AND QUALITY PLAN

Quality plans define the specifics of how a project intends to implement the organization's quality management system. The purpose is to meet the quality goals and objectives of the organization and of that project. The *software quality plans* can be defined in a stand alone software quality plan document or incorporated into the project plans. Many other quality plans exist like:

- V & V plan
- CM plan
- Risk management plan, etc.

These can also be incorporated as subplans in either the project plan or software quality part or they may be documented as stand alone planning documents. This means that the format of the plans is not significant. Note that quality planning takes place and is well documented. There may also be higher-level program or *product-level quality plans* that the *project-level quality plans* are based on.

Quality planning can be done at the:

a. Organizational level.
b. Unit level.
c. Project level.

At the organizational level, quality planning means that the company should define some policy definition and quality plans on the basis of its vision, mission, and policies as set by senior management. The purpose is to understand future customers, their needs, and what they expect from the company. Furthermore, these should be measurable as well.

At the unit level, focus is on the operational quality plans that must be synchronized with the organizational policies and strategies. Also understand

that the project plans and quality plan at the unit level must be consistent with the strategic quality plans at the company level.

At the project level, the projects should plan for quality at the project level. These are generally strategic-level quality plans with details of responsibilities and actions the project plan must define in all aspects of quality plan at the project level. Note that the quality objectives of the project may be inherited from the organizational-level objectives or may be defined separately for the project.

NOTE *An organization should use good inputs as needed by quality planning so that the output of the processes match with the organization's business plans. It includes people, machines, materials, and methods as inputs.*

Quality planning involves the following four (4) steps:

S1: Definition of processes, procedures, standards, and guidelines for developing and testing software. The review of these standards is also to be done as it improves software process maturity.

S2: Performing quality operations like verification and validation as per process definition during entire SDLC.

S3: Regular audits are to be done.

S4: Collecting metrics and defining action plans for the weaker software processes is also to be done.

NOTE *Quality is not by accident but it is an outcome of planned activities performed during SDLC.*

8.5. TOTAL QUALITY MANAGEMENT (TQM)

Total quality management (TQM) means "Quality in Totality". We can say that — "TQM is a systems approach to management that aims to continuously increase value to customers by designing and continuously improving the organizational process and systems. TQM involves all employees and extends backward and forwards to include the supply chain and the customers. TQM is concerned with managing the entire system and not only subsystems or functional departments."

The need is to integrate TQM with employee involvement. TQM is the model for deciding what is important to work on while employee involvement is a model for deciding how to go about working on these objects.

Principles of TQM

General

1. Get to know the next and final customer.
2. Get to know the direct competitors.
3. Dedicate to continual, rapid improvement in quality, response time, flexibility, and cost.
4. Achieve unified purpose via extensive sharing of information and involvement in planning and implementation of change.

Design and Organization

5. Cut the number of components or operations and number of suppliers to a few good ones.
6. Organize resources into chains of customers. Each chain must be self-contained and focused on a customer.

Operations

7. Cut flow time, distance, inventory, and space along the chain of customers.
8. Cut setup, changeover, get ready, and start-up time.
9. Operate at the customer's rate of use or a smoothed representation of it.

Human Resource Development

10. Continually invest in human resources through cross training, education, job switching, and multi-year re-assignments and improved health, safety, and security.
11. Develop operator of products, processes, and outcomes via broadened owner-like reward and recognition.

Quality and Process Improvement

12. Make it easier to produce or provide the product without mishap or process variation.
13. Record and own quality, process, and mishap data at the work place.
14. Ensure that front-line associates get first chance at process improvement before staff experts.

Accounting and Control

15. Cut transactions and reporting; control causes and measure performance at the source, not via periodic cost reports.

Capacity

16. Maintain/improve present resources and human work before thinking about new equipment and automation.
17. Automate incrementally when process variability cannot otherwise be reduced.
18. Seek to have workstations, machines, flow lines, and cells for each product per customer family.

Marketing and Sales

19. Market and sell your firm's increasing customer-oriented capabilities and competencies so that a good decision can be made. This is especially true for principle no. 4; achieve unified purpose via extensive sharing of information involvement in planning and implementation of change.

Principles 6 to 18, collectivity, are aimed at driving out costly overheads, speeding up the design and production process, improving flexibility of human and physical resonances, and eliminating uncertainties caused by rework and shaky suppliers.

8.6. TQM CONCEPTS

Total quality management (or TQM) is the application of quality principles to all facts and business processes of an organization. Dr. Edward Deming implemented a quality management system driven by "Total Quality Management" and "Continual Improvement" in Japan. It resulted in repetitive, cost-effective processes with an intention to satisfy customer requirements and achieve customers satisfaction.

The QA manager has very wide authority for setting and monitoring quality standards throughout the company's functioning. The QA manager then reports directly to the chief executive. The total QA approach to reliability and quality can be very useful when applied to correct quality when it is in doubt.

The incorporation of a TQM philosophy would no doubt add to the initial costs of operation and may even result in a reduced profit situation because it takes some time to build new quality culture. However, once the preliminary step is over and people in the organization get tuned to the new system, this initial investment would generate more payback in years to come than they get otherwise. Then, TQM will only be gainful.

TQM is a systems approach to management that aims to continuously increase value to customers by designing and continuously improving the organizational process and systems. TQM involves all employees and extends backward and forward to include the supply chain and the customers.

NOTE *Good QSE (quality system engineering) will result in a TQM system which will provide for a set of principle activities.*

KPA (Key Process Ares) of a TQM System

1. **Pre-production quality evaluation:** This implies that clear procedures are defined to analyze both the product and process design. This analysis must conclude whether or not the design will lead to fulfillment of customer's need.
2. **Quality planning-product and process:** This means that plans must be formalized before the start of manufacturing as to how to measure, attain, and control the desired product quality and consistency of the manufacturing process.
3. **Quality planning-purchase activities:** This provides the produces necessary to control the quality of purchased material. Such procedure must clearly delineate to vendors the quality requirements and how they can conform to them.

4. **Quality evaluation-product and process:** This means putting into practice what was planned. It includes:
   - Establishment of quality checks.
   - Provision of adequate measuring means.
   - Proper calibrations of devices.
   - Maintain proper quality records.
5. **Quality information system:** Quality information systems means how this information can be analyzed and properly utilized to control the quality activities.
6. **Post-production quality service:** It refers to "after-sales-service." It includes the establishment of procedures to satisfy customers.

Total quality in business strategy shows our managers how to operate within a business by focusing on the customer. TQ focuses first and foremost on "customer satisfaction". Please understand that the main focus is the customer and not the competitor. Also understand that by delivering superior value to the customers, the competitors are left in the dust. We draw a flowchart to show TQM implementation process (see Figure 8.2).

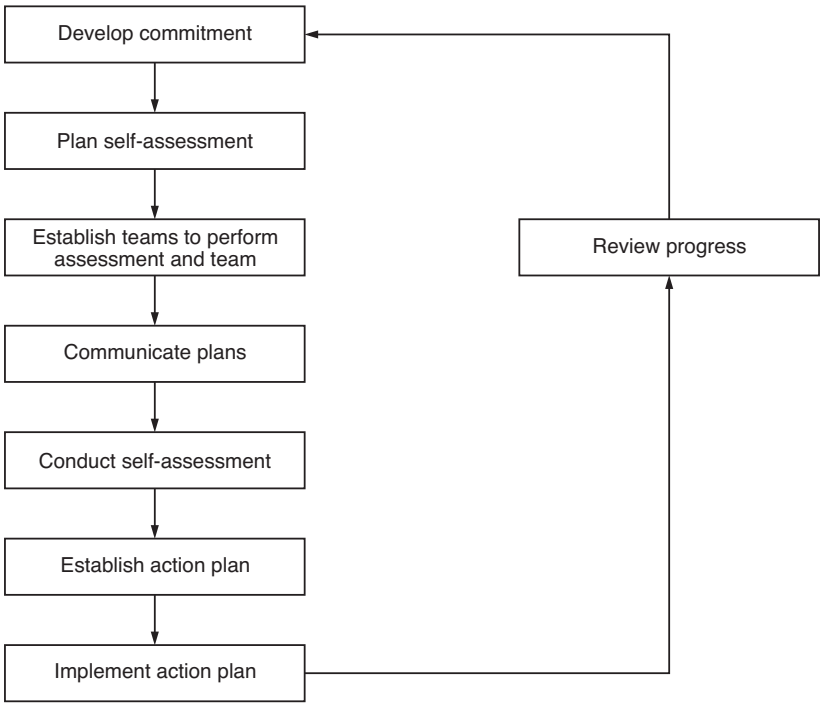


FIGURE 8.2 Flowchart of TQM Model.

8.7. ZERO DEFECT MOVEMENT

Zero defects (ZD) is an approach that attempts to ensure that all processes are performed without error by providing training and motivation to all of the people concerned. It can generate improvements but it is difficult to sustain. It is not the best approach.

From 1970 to until date, Philip Crosby is noted for his ideas relating to the cost of quality. The cost of quality is incurred as a result of producing poor-quality products and services. The essence of Crosby's teachings is contained in what he calls as "The Four Absolutes of Quality:"

1. Quality is conformance to requirements and not goodness.
2. The system is prevention, not appraisal.
3. The performance should be zero defect.
4. The measurement of quality is the price of nonconformance to requirements not quality indices.

Observe a *zero defect day:* To create an event that will let all employees realize through a personal experience that there has been a change.

Management does not want to delay the launch of a product unless it is a mission-critical project, where the goal is to have a *zero-defect product*.

SUMMARY

In this chapter, we have seen what quality assurance is and how to plan quality. What are the objectives of quality assurance? It explains the planning process. It relates the business plan with the quality plan. Finally, it discusses how total quality can be achieved and how zero-defect movement should occur.

MULTIPLE CHOICE QUESTIONS

1. QA ensures that:
   - **a.** Software is good.
   - **b.** Management is good.
   - **c.** Software conforms to technical requirements.
   - **d.** None of the above.

2. The fundamental SQA technique is:
   - **a.** Testing
   - **b.** Auditing
   - **c.** Meeting
   - **d.** None of the above.
3. Quality is:
   - **a.** By accident
   - **b.** Purchased
   - **c.** Planned activities during SDLC
   - **d.** None of the above.
4. Total quality management means:
   - **a.** Quality in totality
   - **b.** Quality of machines
   - **c.** Cost of quality
   - **d.** None of the above.
5. According to Crosby:
   - **a.** Performance should be "zero defect."
   - **b.** Performance should have 0/1 defects.
   - **c.** Performance should not be on the basis of defects.
   - **d.** None of the above.
6. Two commonly used metrics of quality are:
   - **a.** Defect count and rework cost
   - **b.** LOC count and project cost
   - **c.** FP count and project cost
   - **d.** None of the above.
7. A quality factor that reduces the project's development cost is:
   - **a.** Reliability
   - **b.** Reusability
   - **c.** Re-engineering
   - **d.** None of the above.
8. The goal is to have a:
   - **a.** Zero-defect product
   - **b.** Zero/one defect product
   - **c.** Defect product
   - **d.** None of the above.

9. Quality is NOT manufactured, it is:

a. Engineered **b.** Purchased

c. Hired **d.** None of the above.

10. The average time for a system to perform without failing is known as:

a. MTBF **b.** MTTF

c. MTS **d.** None of the above.

ANSWERS

1. c.	**2.** b.	**3.** c.	**4.** a.
5. a.	**6.** a.	**7.** b.	**8.** a.
9. a.	**10.** b.		

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

Q. 1. Differentiate between tools and techniques.

Ans.

Tools	Techniques
1. Tools are useful only if techniques are available to use.	**1.** Technique is not dependent on any tools.
2. Different techniques may use the same tool to achieve different results.	**2.** Same technique may use different tools to achieve same result.
3. Tool improvement needs technological change.	**3.** Technique change can be effected through produces change.
4. Contribution of tolls in improvement is limited.	**4.** Contribution of techniques in improvement is important.

Q. 2. Distinguish between guidelines and standards.

Ans.

Guidelines	Standards
1. They are suggested ways of doing things by some experts	**1.** Standards improve quality.
2. Guidelines may not be followed.	**2.** Standards are a must.
3. Guidelines may or may not be written.	**3.** Standards must be written to avoid any misunderstanding or loss of communication.

Q. 3. Differentiate between internal and external customer.

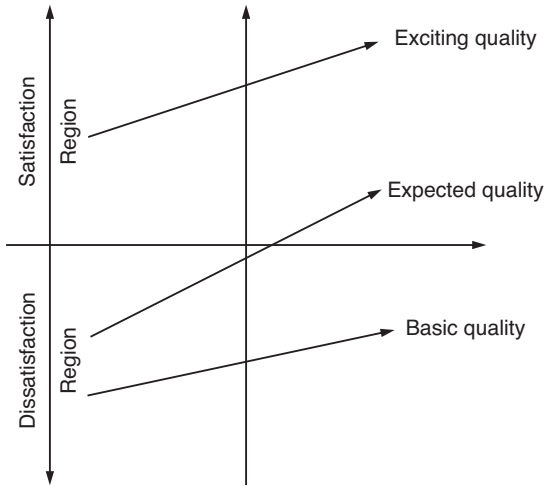
Ans. **Internal customer:** Internal customer satisfaction philosophy is guided by the principles of total quality management. There are different departments in a company. Thus, one department acts as a supplier or customer of another department. So every department must identify its customer and suppliers and try to fulfill their requirements. Thus, the external requirements of customer are satisfied.

External customer: The external customer is the final users who buy our software. SRS is prepared. Customers are paying us. Customer is the king!

Q. 4. Explain Kano's model of quality.

Ans. According to the Noritaki model, quality can be of 3 types:

1. **Basic quality:** There is a basic level of quality that a customer expects the product to have. These are quality requirements that are assumed by the customer and are typically not explicitly stated or requested. For example, customers expect the software vendors to provide a manual when they are purchasing software. They will not ask for these items but will expect them to be there. This level of quality does not satisfy the customer. This is shown in graph below.



2. **Expected quality:** These are these quality requirements that the customer explicitly requests. For example, they will state their preferences for the software versions when they buy. The customer will be dissatisfied if this level of quality is NOT met and increasingly satisfied as this quality level increases.
3. **Exciting quality:** As an example, we wish to develop a word processor software. It has all of the features of the competitors but if we also include features like sounds, music, etc. The features the customers need but he or she may not state them. These are exciting features.

Q. 5. What is benchmarking?

Ans. It is the process used by an organization to identify, understand, adapt, adopt outstanding practices, and processes from others.

Q. 6. What are the advantages and disadvantages of formal meetings?

Ans.

Advantages	Disadvantages
1. Allows multiple people to express opinions and issues.	**1.** Can degrade into time-wasters.
2. Creates team synergy.	**2.** Some people may feel uncomfortable.
3. Can discuss a wide range of topics and issues in detail.	**3.** All topics may not be relevant to all people.
4. Two-way communications.	**4.** Difficult when people are at remote (far) locations.

REVIEW QUESTIONS

1. The company maintains an information system for the purpose of meeting customers needs. How are these needs determined for external customers? For internal customers?
2. "Software quality is not by accident." Justify the statement.
3. How can we plan quality?
4. What is TQM? Explain TQM concepts.
5. What is the "Zero Defect" Movement?
6. Explain the different types of quality requirements.
7. How are testing and SQA are related?
8. Assume you are responsible for developing a web-based application and in that application you need to prepare the quality control (QC) checklist for conducting a database testing. Mention the specific items, responses, and comments that need to be incorporated in the checklist.

CHAPTER 9

QUALITY STANDARDS

Inside this Chapter:

9.0. Introduction
9.1. Quality Models/Standards/Guidelines
9.2. Types of Models
9.3. ISO Standards
9.4. CMM and CMMI
9.5. Six Sigma Concepts
9.6. Quality Challenge
9.7. National Quality Awards

9.0. INTRODUCTION

Literature says that unlike standardization in the communications field, in software development standardization is viewed with a mixed-type of reactions. Opponents (not in favor) say that standardization curtails individual drive to be innovative. **Proponents** (in favor) say that standards reduce the activity of reinventing the same or similar processes for development and QA. Note that the repeatability of processes is the key benefit of this standardization. Also note that repeatability reduces the cost of software development and produces a base-quality level of software products.

9.1. QUALITY MODELS/STANDARDS/GUIDELINES

Standards have always been associated with quality measures. They are the benchmarks against which the quality of the software application is measured.

One of the most vital attributes of standards is **consistency**. Standards must be flexible too, in order to adapt to changing technology. Standards should be usable and understandable. Standards can streamline transitions when developers leave projects. In a nutshell, standards are key components to successful software development.

As per Kramm and Graziano, using standards:

a. Provides consistency across projects.

b. Eliminates conjecturing (i.e., guess works).

c. Improves model's maintainability.

d. Reduces long-term costs.

e. Promotes reusability.

The ISO (International Standard Organization) has developed a series of standards, collectively named as **ISO 9000**. ISO was founded in 1946 in Geneva, Switzerland. It develops and promotes international standards in the field of **quality assurance** and **quality management**. ISO 9000 standards are generally applicable to all tangible products manufactured-from fans and light to software. ISO standards are reviewed and updated from time to time (once every 5-8 years). The latest ISO 9000 standards released in 2000 are named ISO 9000: 2000. It has 3 components:

ISO 9000: Fundamentals and Vocabulary
ISO 9001: Requirements
ISO 9004: Guidelines for Performance Improvements

NOTE

ISO 9002 — Deals with the quality system model for QA in production.
ISO 9003 — Deals with the quality system model for QA in testing products.

9.2. TYPES OF MODELS

To quantitatively and qualitatively evaluate the software quality, metrics users need to establish a quality metrics model. Most of the quality models are hierarchical models. Models may be:

a. Theoretical models: Based on hypothesis relations among variables.

b. Data-driven models: Based on statistical analysis.

c. Combined models: Intuition is used to find the basic type of model and data analysis is used to find model's constants.

There are 2 terms used here:

1. Quality factor
2. Quality criterion

A **quality factor** represents a behavioral characteristic of a system. Some high-level quality factors are correctness, reliability, efficiency, testability, portability, and reusability (as per McCall's quality factors). On the other hand, quality criterion is an attribute of a quality factor that is related to the software system. Please note that highly modular software allows designers to add cohesive components in one module so as to increase the system's maintainability. Similarly, **traceability** of a user requirement allows developers to accurately map the requirement to a subset of the modules, thus increasing the system's correctness. Some quality criteria relate to products and some to personnel. For example, **modularity** is a **product-level quality criterion** whereas training relates to development and SQA people.

According to McCall et al., there are 23 quality criteria (three quality categories) as follows:

The relationship between quality factors and quality criteria is also shown in Figure 9.1. It is clear from the figure that an arrow from a quality criterion to a quality factor means that the quality criterion has a positive impact on the quality factor. For example, traceability has a positive impact on correctness.

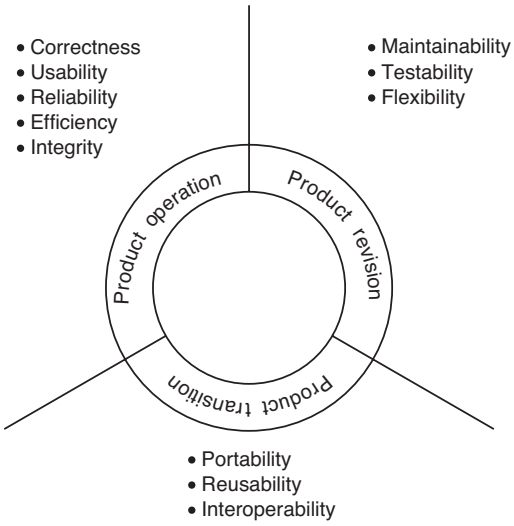


FIGURE 9.1 McCall's Quality Factors.

In general, quality factors are NOT completely independent. Please note the following:

1. These quality factors are inversely proportional, i.e., if we try to improve one quality factor, another one may be degraded. For example, if a software is made more testable then its efficiency will go down. Similarly, an effort to make code portable will reduce its efficiency.

2. Some quality factors have a positive impact on others. For example, an effort to enhance the correctness of a system will increase its reliability. Similarly, if we enhance a system's testability, we will improve its maintainability, too.

The relationship diagram is shown below:

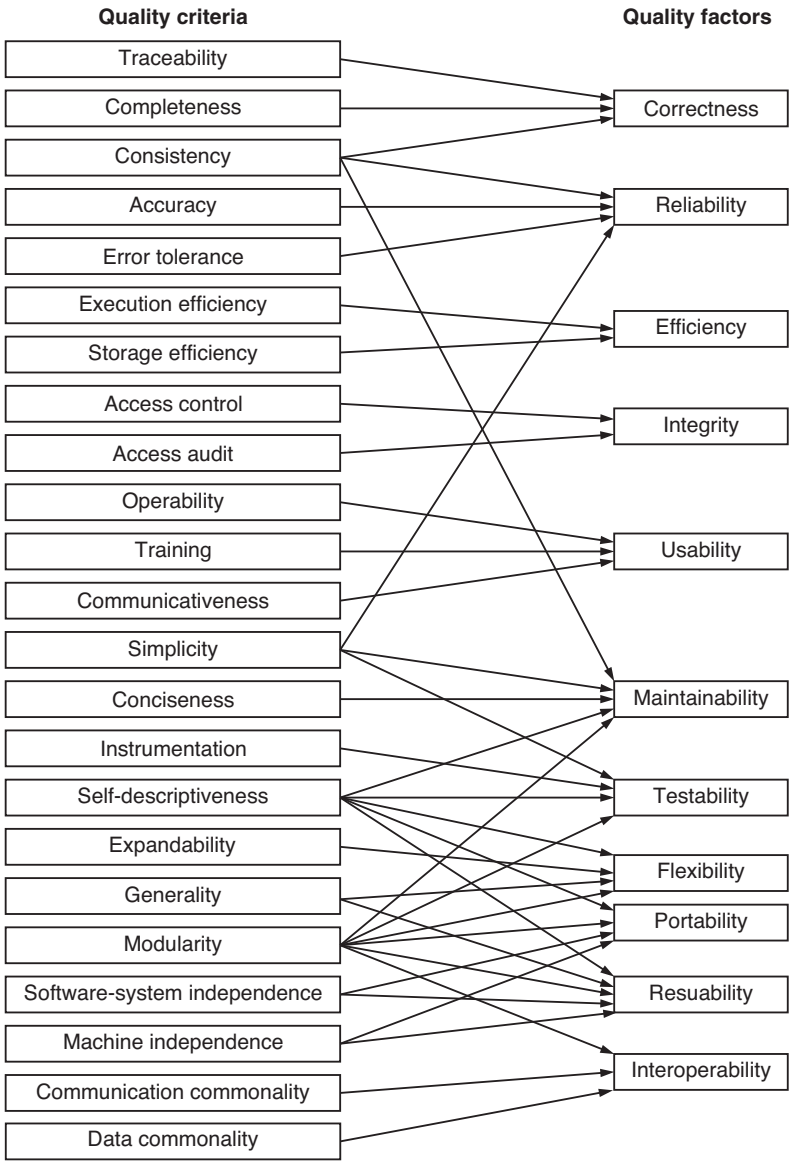


FIGURE 9.2 Relationships Between Quality Factors and Quality Criteria.

Another quality model, a very recent one is ISO 9126.

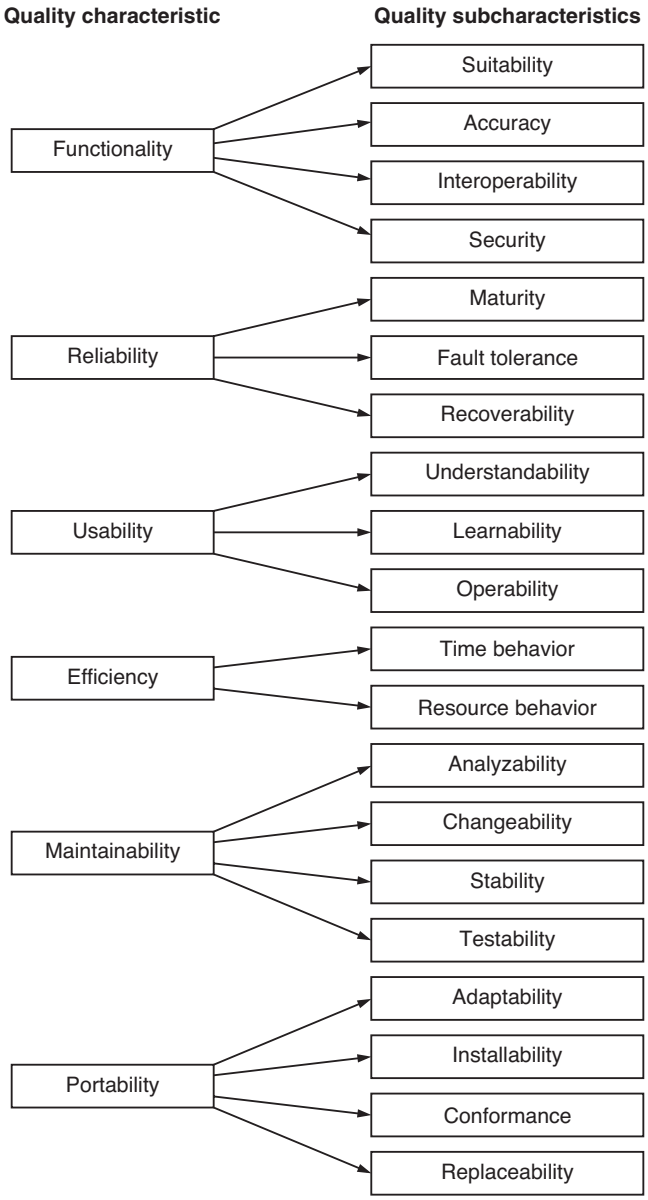


FIGURE 9.3 ISO 9126 Model.

ISO 9126 defines the 6 categories of quality characteristics as follows:

1. **Functionality:** The ability of a system to perform the required functions that satisfy user's need.
2. **Reliability:** A set of attributes that give the ability of a software to do failure-free operation under stated conditions for a stated period of time.
3. **Usability:** The ease with which a software can be used by its end users.
4. **Efficiency:** A set of attributes showing the relationship between the software's performance and the amount of resource used under stated conditions.
5. **Maintainability:** The ease with which a software can be changed like corrections, improvements of adaptations of software to environmental changes, etc.
6. **Portability:** The ability of a software system to run on different platforms both hardware and software platforms.

Actually, the ISO 9126 standard includes a quality model that further decomposes the quality characteristics into more correct subcharacteristics. The probability characteristic has been decomposed into 4 subcharacteristics—adaptability, insatiability, conformance, and replaceability.

There are broad and independent quality characteristics and totally 20 subcharacteristics. We define each of them.

Quality Characteristics (6):

1. **Functionality:** The extent to which the software performs its intended functions that satisfy customer's need.
2. **Reliability:** The ability of a software to perform (run) without fasting under stated conditions for a stated period of time.
3. **Usability:** The ease with which the users can use and understand the system.
4. **Efficiency:** A set of attributes that bear on the relationship between the software's performance and the amount of resource used under stated conditions.
5. **Maintainability:** The ease with which the software can be changed, improved, and corrected.
6. **Portability:** The ability of software to run on any platform (hardware or software environments).

Let us now define 20 subcharacteristics under these 6 quality characteristics, as given by the ISO 9126 software quality model:

1. **Suitability:** The capability of the software to provide an adequate set of functions for specified tasks and user objectives.
2. **Accuracy:** The capability of the software to provide the right or agreed upon results (as agreed in SRS).
3. **Interoperability:** The capability of the software to interact with one or more other systems (softwares), for example, Java with Oracle 8i (RDBMS).
4. **Security:** The ability of software to prevent or deny unintended access.
5. **Maturity:** The ability of the software to avoid failure as a result of faults in the software.
6. **Fault tolerance:** The ability of the software to maintain a specified level of performance in case of faults.
7. **Recoverability:** The ability of the software to reestablish its level of performance and recover data directly in case of a failure.
8. **Understandability:** The ability of the software product to enable the user to understand whether the software is suitable or not.
9. **Learnability:** The ability of the software to enable the user to learn its applications.
10. **Operability:** The capability of the software to enable a user to operate it.
11. **Attractiveness:** The capability of the software to be liked by the user.
12. **Time behavior:** The ability of software to provide suitable response, processing times, and throughput rates when performing its function under stated condition.
13. **Resource utilization:** The ability of software to use appropriate resources in an appropriate time when the software performs its function under stated condition.
14. **Analyzability:** The capability of the software to be diagnosed for deficiences or causes of failures in software.
15. **Changeability:** The ability of a software to implement change.
16. **Stability:** The ability of software to optimize unexpected effects from modifications of the software.

17. **Testability:** The ability of the software to enable the changed software to be validated.
18. **Adaptability:** The capability of the software to get changed/modified for different specified environments.
19. **Installability:** The ability of the software to be installed in a specified environment.
20. **Coexistence:** The ability of the software to co-exist with other independent software in a common environment sharing common resources.
21. **Replaceability:** The ability of the software to be used in place of other specified software. Organizations take time to reach best-quality level from the present level. It is a slow process.

Comparisons of McCall's and ISO 9126 Models

I. Similarities

Both models focus on one word only, i.e., quality. In McCall's model, what we call a quality factor is called a quality characteristic in ISO 9126 model. Both models have some quality factors in common like reliability, usability, efficiency, maintainability, and portability.

II. Dissimilarities

a. The ISO 9126 model focuses on the characteristics visible to customers whereas McCall's model focuses on internal qualities. For example, reusability is an internal characteristic of a product. Software developers try hard to produce remissible components whereas its impact is not perceived by customers.
b. In McCall's model, one quality criteria can impact several quality factors whereas in the ISO 9126 model, one subcharacteristic impacts exactly one quality characteristic.
c. A high-level quality factor like testability in McCall's method is a low-level subcharacteristic of manageability in the ISO 9126 model.
d. McCall suggests 11 high-quality factors whereas the ISO 9126 standard defines only 6 quality characteristics.

NOTE *Quality factors like reusability and interoperability are significant for software developers.*

The ISO 9126 model just considers the product.

d. There is no clear cut demarcation between top-level quality factor/characteristic and concrete quality. For example, interoperability is a dependent-quality characteristic. It is a part of a functionality attribute. It is not clear as to why?

9.3. ISO STANDARDS

Standards are the benchmark against which software quality is measured. Consistency is one of the most important attributes of standards. They must be flexible in order to adapt to the changing technology. Standards should be usable and understandable. Standards can ease the transition when developers leave projects.

According to Kramm, standards:

a. Provide consistency across all projects.

b. Eliminate guesswork.

c. Improve model's maintainability.

d. Reduce long-term costs.

e. Promote reusability.

ISO 9000 Standards

ISO is a worldwide federation of National Standards bodies. The ISO 9000 series is one of the quality system standards that is widely accepted all over the world.

ISO 9000 improves the cost structure of any company by 5–20%. Understand that ISO 9000 standards define minimum requirements for business quality assurance systems. These are consensus standards by ISO in Europe and ANSI in the United States.

The ISO 9000 series is emerging as the quality system standards for the industry with ISO 900-3 providing the guideline for applying ISO 9001 to software development, supply, and maintenance. First of all, ISO 9000 quality standards were published in 1987. Then they were revised in 1994 and in 2000. They are reviewed constantly to retain their efficiency.

ISO 9000 is a series of documents dealing with quality systems. The following documents exists here:

1. **ISO 9000 (Q90):** It is a guideline for the selection and use of quality system standards.
2. **ISO 9001 (Q91):** It defines minimum quality system requirements for design, production, installation, and service. It applies to manufacturing and service businesses.
3. **ISO 9002 (Q92):** It is a subset of 9001. It applies to production and installation activities only.
4. **ISO 9003 (Q93):** It is applicable to the final inspection and testing.
5. **ISO 9004 (Q94):** It is a guideline for quality system elements. It is like a helpbook that describes, explains, and recommends.

ISO 9000 is designed to provide a flexible structure that ensures the identification and enforcement of QA principles. For R & D organizations, ISO 9000 QA principles are as follows:

1. Precise goals/objectives related to specific requirements.
2. Clearly identified responsibilities.
3. Preventive and corrective actions.
4. Process control-flow of requirements.
5. Constant evaluation and feedbacks.
6. Plan-do-check-act (PDCA) cycle.

The ISO 9000 QA approach to management is a documented system approach. It adds the following features:

1. A method for system monitoring.
2. A method for correcting mistakes.
3. A method for changing the system if it has become old (obsolete).

These aspects are very much useful. ISO 900 prepares for it. Three types of failures are recognized:

a. Human error
b. Failure of inputs
c. Obsolescence of existing methods

The ISO 9000 quality system distinguishes between these causes. In the first case, the error or omission is corrected. In other cases, the method

is amended. Thus, we can say that ISO 9000 is both self-correcting and a learning system. It changes to reflect changing needs. Also, note that it is known as QA rather than a traditional quality control-type system.

The final management note is the quality review. This is possible because of the self-adjusting nature of the R & D QA system. The **management review** is the engine for that process. Review should find out:

a. What information is needed to be sure that the quality policy is being implemented?

b. What information is needed to decide whether the policy needs an amendment?

c. How frequently these data need to be collected?

Note that it is vital to decide what is critical and genuinely indicative of the health of the organization. The more data management asks for the less it will be able to make sense of the data. Also note that the cost of data collection needs to be considered. The more management asks for data that are not automatically collected as part of the day-to-day work, the more the cost of the quality system will rise. The efficient and economic way to resolve this is to scan the procedures once they have been written in order to identify data that exists in the system.

9.4. CMM AND CMMI

CMM or the capability maturity model is developed by the Software Engineering Institute (SEI) at Carnegie Mellon University. Actually, in order to improve a defined process, organizations need to evaluate its capabilities and limitations. So, CMM allows an organization to evaluate its software development processes. CMM is not a process model but supports incremental process improvement.

Immature organizations may not have a defined process. Even if it is there, it may not follow it. Developers and managers must realize that "prevention is better than cure", i.e., they must not react to problems when they occur but instead should take preventive measures to eliminate them or to reduce the frequency of their occurrences. So, immature organizations work in an adhoc manner. That is, both product and process problems are resolved in an adhoc manner. Also, in these organizations the cost, time, and quality estimates are highly inaccurate as there are no measurement programs to collect process data. That is why there are cost overruns and time slippages. No quality metrics exist in these immature organizations.

Mature organizations are those that perform its activities in a well-planned manner. These organizations will measure both process and product characteristics. The aim is to keep track of the progress and quality of products. As proper metrics exist in these organizations, so estimates of cost, time, manpower, and quality are much more accurate. Proper employee trainings are done. Constant effort is made to improve quality. Define processes are regularly updated in order to introduce newer techniques, tools, and experience from past projects. Note that as organizations grow old (mature) then standards and organizational policies play key roles in software development. Also note that an organization's processes become mature in a step-wise fashion, i.e., in an evolutionary manner, processes are improved.

NOTE *As an organization moves from one level to the next, its process maturity improves to produce better quality software at a lower cost.*

CMM Model

Capability Maturity Model (CMM)

The capability maturity model (CMM) is a methodology used to develop and refine an organization's software development process.

The core principles of CMM include:

- well-defined steps for performing each task.
- proper training for each person to perform his or her job.
- existence of management support for all efforts performed.
- measurement of all steps performed.

Overview of CMM

The CMM is not a software life-cycle model. Instead, it is a strategy for improving the software process, irrespective of the actual life-cycle model used. The CMM was developed by the Software Engineering Institute (SEI) in 1986. CMM is used to judge the maturity of the software processes of an organization. Term maturity is a measure of the goodness of the process itself.

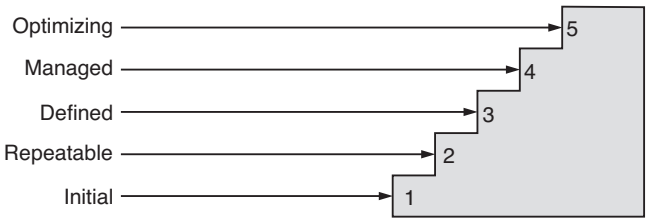


FIGURE 9.4

Level 1: Initial—The software process is characterized as adhoc, i.e., most activities are responses to crisis. It depends totally on the current staff; as the staff changes, so does the process. Few processes are defined and success depends on the individual effort. It is an unfortunate fact that majority of software organizations all over world are level-1 organizations.

Level 2: Repeatable—Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

Level 3: Defined—The software process for both management and engineering activities is documented, standardized, and integrated into an organization-wide software process. All projects use a documented and approved version of the organization's process for developing and supporting software. This level includes all characteristics defined for level 2.

Level 4: Managed—Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled using detailed measures. This level includes all of the characteristics defined for level 3.

Level 5: Optimizing—Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technology. This level includes all characteristics defined for level 4.

Key process area (KPA) determines a group of related activities which are performed collectively to achieve goals for establishing process at that maturity level.

Each maturity level comprises KPAs to identify the area that need improvements in the organization. The maturity levels and their KPAs are given below:

Summary of the above levels is given in the table below.

Maturity level	Characterization	Key process areas (KPAs)
1. Initial	Adhoc process	—
2. Repeatable	Basic project management process	Project planning, SCM
3. Defined	Process definition	Training program
4. Managed	Process measurement	Quality management
5. Optimizing	Process control	Defect prevention

Experience shows that advancing a complete maturity level usually takes from 18 months to 3 years, but moving from level 2 sometimes takes 3 or even 5 years.

Overview of Quality Standard (ISO 9001)

Quality assurance systems are created to help organizations ensure their products and services and to satisfy customer expectations. It is a different attempt to improve software quality. This standard is a set of documents dealing with quality systems that can be used for quality assurance purposes. ISO 9000 describes the elements of the quality assurance system in general terms. These elements include the organizational structure, procedures, processes, and resources needed to implement quality planning, quality control, quality assurance, and quality improvement.

ISO 9000 is a series of five related standards and within this series, ISO 9001 standard for quality system is the standard applicable to software development.

ISO 9001 is the quality assurance standard that applies to software engineering. This standard contains 20 requirements that must be present for an effective quality assurance system. The requirements delineated by ISO 9001 address topics such as management responsibility, quality system, contract review, design control, document and data control, product identification, process control, inspection and testing, corrective and preventive action, control of quality records, training, servicing, and statistical techniques.

Comparison of ISO 9001 and CMM

Differences

1. CMM emphasizes continuous process improvement whereas ISO 9001 addresses the minimum criteria for an acceptable quality system.
2. CMM focuses strictly on software while ISO 9001 focuses on hardware, software, processed materials, and services.

Similarities

1. The fundamental premise of ISO 9001 is that every important process should be documented. ISO 9001 requires documentation that contains the instructions or guidance on what should be done or how it should be done.

   Whereas CMM shares this emphasis on processes that are documented and practiced as documented.

2. The CMM also emphasizes the need to recover information for later use in the process and for improvement of the process. This is equivalent to the quality records of ISO 9001 that document whether or not the required quality is achieved and whether or not the quality system operates effectively.

CMM Applications

The KPAs of CMM, as discussed, must be satisfied by an organization to be at a certain level of maturity. For example, for an organization to be at level 3, the organization must meet all six KPAs associated with level 2. SEI proposes two methods to evaluate the current capabilities of organizations—internal assessments and external assessments.

CMMI—Capability Maturity Model Integration

CMM in the software area is known as CMM-SW, i.e., software CMM that was first released in 1991 as CMM-SW Ver 1.0. Then, CMM-SW Ver 1.1 was released in 1993. The concept of CMM is not only software specific.

Due to many CMMs, an organization used to have many problems because these different models have different structures, no commonality of goals, integration of different CMMs also become problematic. So, there was a need to have a unified view of process improvement throughout an organization. Thus, came into existence CMMI. CMMI inputs are received from Figure 9.5.

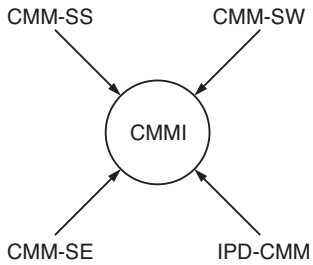


FIGURE 9.5 CMMI.

where CMM-SW is CMM for software,
IPD-CMM is integrated product development CMM,
CMM-SE is CMM for systems engineering,
and CMM-SS is CMM for supplier sourcing.

SEI-CMMI Model

From Figure 9.5, it is clear that each process area of the staged representation of the CMMI for development includes one or more specific goals. Also, the CMMI for development has generic goals that are applicable to all process areas. A goal is a required component that must be achieved to consider the process area satisfied. Each goal includes one or more practices. A **practice** is an expected component where each practice or an equivalent alternative must be achieved to consider the processes are satisfied. This is shown below:

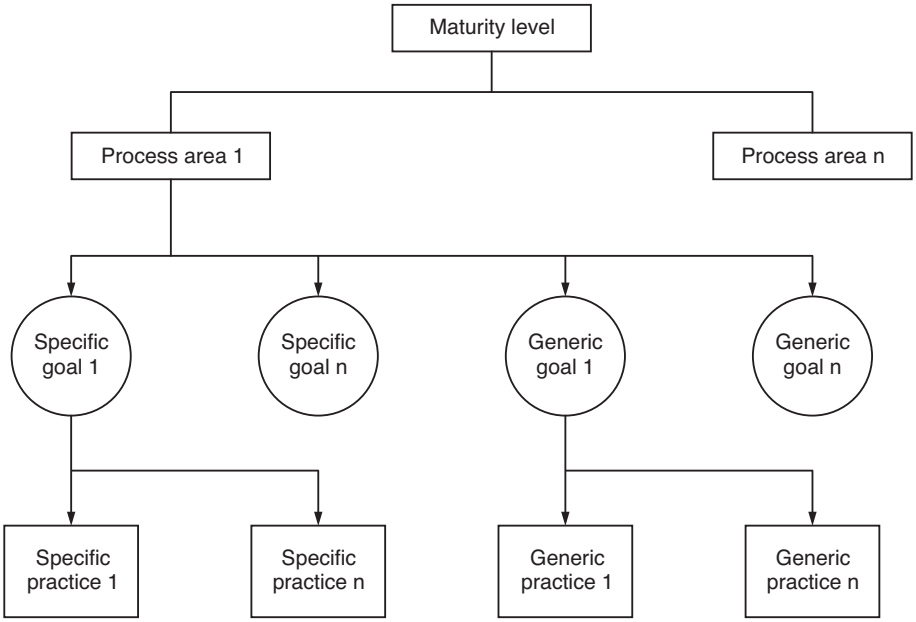


FIGURE 9.6 SEI-CMMI Representation.

Level	CMMI—Process Areas
1.	Initial (Adhoc)
2.	Repeatable: Project planning, SCM, project monitoring, requirements management, process and product quality assurance
3.	Defined: Requirements development, trainings, technical solution, risk management, V and V
4.	Quantitatively managed: Organizational process performance, SPM
5.	Optimizing: Organizational innovation and causal analysis

Need of CMMI

1. Present day web applications are very complex. So, they use some subsystems (third-party components) also. For example, a communication module may be purchased from some third-party vendor.
2. Many other components like RDBMS, messaging, security, and real-time processing are part of bigger softwares.

   Understand that the coexistence and interoperability of these different components developed by different vendors is very important for project's success. Thus, there is a need to find out the maturity level of an integrated product development process.
3. Complex software systems usually run on a specialized hardware and a specialized OS. These need to be evaluated.

9.5. SIX SIGMA CONCEPTS

Six Sigma is a business management strategy initially implemented by Motorola that today enjoys widespread application in many sectors of industry.

Six Sigma seeks to improve the quality of process outputs by identifying and removing the causes of defects (errors) and variability in manufacturing and business processes. It uses a set of quality management methods including statistical methods, and creates a special infrastructure of people within the organization ("Black Belts", "Green Belts", etc.) who are experts in these methods. Each Six Sigma project carried out within an organization follows a defined sequence of steps and has quantified financial targets (cost reduction or profit increase).

The term "Six Sigma" is derived from a field of statistics known as process capability studies. Originally, it referred to the ability of manufacturing processes to produce a very high proportion of output within specification. Processes that operate with "six sigma quality" over the short term are assumed to produce long-term defect levels below 3.4 defects per million opportunities (DPMO). Six Sigma's implicit goal is to improve all processes to that level of quality or better.

Why Six Sigma Is So Important?

Six Sigma refers to a 99.997% perfection rate, i.e., 3.4 parts per million. If a company is producing only 3.4 defective products per million units of the

product then it is considered to have achieved Six Sigma level. Top notch companies around the world generally are considered to be operating on around 99% perfection which suggests that there can be further improvement and reduction in defects hence helping the company increase profitability. This improvement can save millions of dollars for the companies which are wasted in various unnecessary processing and defects that lead to customer dissatisfaction.

Six Sigma uses process data and analytical techniques in order to find out various process variables. Once the process variables are obtained, they help in developing the exact understanding of various processes. This understanding/data is then used to improve the processes and help in reducing defects/losses in other areas of the organization.

Six Sigma stands for **Six Standard Deviations** (Sigma is the Greek letter used to represent standard deviation in statistics) from the mean. Six Sigma methodologies provide the techniques and tools to improve the capability and reduce the defects in any process. Six Sigma is a registered service mark and trademark of Motorola, Inc. Motorola has reported over $17 billion in savings from Six Sigma as of 2006.

Six Sigma methodology can also be used to create a brand new business process from the ground up using **DFSS** (*Design For Six Sigma*) principles. Six Sigma strives for perfection. It allows for only 3.4 defects per million opportunities for each product or service transaction. Six Sigma relies heavily on statistical techniques to reduce defects and measure quality.

Historical Overview

Six Sigma was originally developed as a set of practices designed to improve manufacturing processes and eliminate defects, but its application was subsequently extended to other types of business processes as well. In Six Sigma, a defect is defined as anything that could lead to customer dissatisfaction.

The particulars of the methodology were first formulated by Bill Smith at Motorola in 1986. Six Sigma was heavily inspired by six preceding decades of quality improvement methodologies such as quality control, TQM, and Zero Defects, based on the work of pioneers such as Shewhart, Deming, Juran, Ishikawa, Taguchi, and others.

Like its predecessors, Six Sigma asserts that:

1. Continuous efforts to achieve stable and predictable process results (i.e., reduce process variation) are of vital importance to business success.

2. Manufacturing and business processes have characteristics that can be measured, analyzed, improved, and controlled.
3. Achieving sustained quality improvement requires commitment from the entire organization, particularly from top-level management.

Features that set Six Sigma apart from previous quality improvement initiatives include:

1. A clear focus on achieving measurable and quantifiable financial returns from any Six Sigma project.
2. An increased emphasis on strong and passionate management leadership and support.
3. A special infrastructure of "Champions," "Master Black Belts," "Black Belts," etc. to lead and implement the Six Sigma approach.
4. A clear commitment to making decisions on the basis of verifiable data rather than assumptions and guesswork.

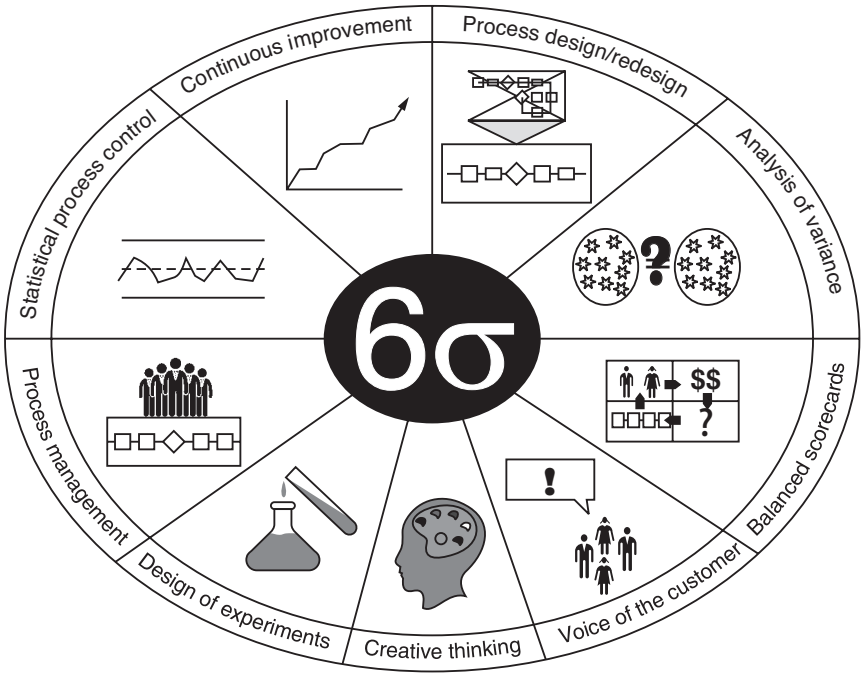


FIGURE 9.7 Six Sigma.

Software Requirements

Six Sigma software tools augment the implementation of Six Sigma methodology by complimenting and sometimes substituting human efforts. Six Sigma software tools fill in the vacuum of additional needs by companies that are implementing the Six Sigma methodology.

Software developers, in accordance with the needs and demands of different businesses, have developed various Six Sigma software tool modules. Some examples of Six Sigma software tools are as follows:

1. DMAIC Six Sigma - a process management tool
2. Design for Six Sigma or DFSS - a design tool
3. Quality improvement package - a quality control tool
4. Production management package - a process simulation tool
5. Project optimization and simulation - an analytical tool
6. Testing and measurement - a testing and control tool

The all-encompassing comprehensive Six Sigma software tool packages pack a lot of powerful features into them which help to speed up the decision-making process and data mining, while dramatically simplifying predictive modeling activities.

List of Six Sigma Software Packages

There are generally three classes of software used to support Six Sigma:

1. Analysis tools which are used to perform statistical or process analysis
2. Program management tools used to manage and track a corporation's entire Six Sigma program
3. DMAIC and Lean online project collaboration tools for local and global teams

ANALYSIS TOOLS

1. iGrafx Process for Six Sigma
2. EngineRoom by MoreSteam
3. IBM WebSphere Business Modeler
4. JMP
5. Microsoft Visio
6. Minitab

7. QPR ProcessGuide by QPR Software
8. Quality Companion by Minitab
9. SigmaXL
10. SigmaFlow(BPA)
11. Software AG webMethods BPM Suite
12. Statgraphics
13. STATISTICA
14. Telelogic System Architect
15. Actuate
16. The Unscrambler
17. Oracle Crystal Ball
18. Select Architect Business Process Modeling

DMAIC/LEAN ONLINE COLLABORATION TOOLS

1. SigmaFlow Integrated, Project Management Tools (PPM), (BPA), and (ECM)
2. Grouputer SigmaSense

HARDWARE REQUIREMENTS

Most of the Six Sigma software tools are available for both Mac and IBM compatible PCs. The minimum system requirements are:

- At least Pentium 386; but for most products - 1.0 GHz Pentium processor
- 256MB RAM
- 1.0GB of free disk space
- Graphics card (at least VGA or better is recommended)
- Windows, several versions; depending on which product you buy

With Six Sigma software tools at your disposal, you can process a lot of data, more than you ever could by hand. Artificial intelligence is used for faster, more dependable project selection and analysis. Six Sigma software tools also assist you in predicting future behaviors and tendencies. Six Sigma software tools have finally come of age and are here to stay.

SIX SIGMA METHODOLOGIES

Six Sigma methodologies are very easy to implement and integrate into any organization if all of the processes are properly followed. The Six Sigma methodology operates on the following main steps:

1. **Define:** In this step, the team responsible for Six Sigma methodology implementation in the organization collectively defines various goals and sub-goals. The first step emphasizes customer satisfaction, identification of root causes of any identified defect, and improvement and establishment of infrastructure in order to help achieve the defined goals and sub-goals.
2. **Measure:** This step involves activities such as preparing various metrics based on the data available in hand, more data collection activities, basic data packets creation or sampling, etc.
3. **Analyze:** This step involves analysis of defects using "Cause & Effect" and other diagrams, study of possible modes of failure, i.e., root cause analysis, etc. in order to prepare various charts to improve process and control, and monitor the process improvement activities. This step can help in establishing the facts about processes currently being utilized in terms of their effectiveness, contribution to Six Sigma methodology success, defect rates, and at what extent these processes help in achieving the organization-level success etc. and can help in achieving various goals.
4. **Improve:** This step involves utilization of data collection, metrics, and analysis done during the Define, Measure, and Analysis steps. Improvised techniques are brought into action and applied to the process improvement model. The team working on Six Sigma methodology application to the organization also may experiment with various techniques to help in process improvement.
5. **Control**: In this step, all of the techniques being utilized for process improvement are monitored and adherence to procedural methodology is ensured. Various charts based on the techniques being utilized against time frame are used to help keep a check on the successful execution of Six Sigma methodology.

It's very important to understand the Six Sigma methodology and its application for any industry/ organization. Six Sigma methodology not only emphasizes reduction of defects, i.e., only 3.4 per million, it also gives significant importance to customer satisfaction and customer voice.

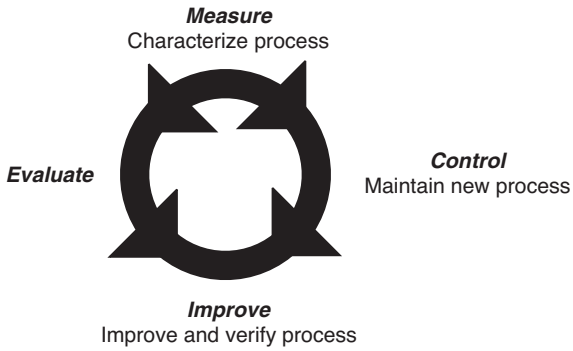


FIGURE 9.8

Six Sigma projects follow two project methodologies inspired by Deming's Plan-Do-Check-Act Cycle. These methodologies comprise five phases each and are known by the acronyms DMAIC and DMADV.

- **DMAIC** is used for projects aimed at improving an existing business process.
- **DMADV** is used for projects aimed at creating new products or process designs.

DMAIC

The five phases in the DMAIC project methodology are:

1. Define high-level project goals and the current process.
2. Measure key aspects of the current process and collect relevant data.
3. Analyze the data to verify cause-and-effect relationships. Determine what the relationships are and attempt to ensure that all factors have been considered.
4. Improve or optimize the process based on data analysis using techniques like design of experiments.
5. Control to ensure that any deviations from the target are corrected before they result in defects. Set-up pilot runs to establish process capability, move on to production, set up control mechanisms, and continuously monitor the process.

DMADV

The five phases in the DMADV project methodology are:

1. Define design goals that are consistent with customer demands and the enterprise strategy.

2. Measure and identify CTQs (Critical To Quality characteristics), product capabilities, production process capability, and risks.
3. Analyze to develop and design alternatives, create a high-level design, and evaluate design capability to select the best design.
4. Design details, optimize the design, and plan for design verification. This phase may require simulations.
5. Verify the design, set up pilot runs, implement the production process, and hand it over to the process owners.

DMADV is also known as DFSS, an abbreviation of "Design For Six Sigma".

The Similarities of DMAIC and DMADV

Let's first look at the DMAIC and DMADV methodologies and talk about how they're alike. DMAIC and DMADV are both:

- Six Sigma methodologies used to drive defects to less than 3.4 per million opportunities.
- Data intensive solution approaches. Intuition has no place in Six Sigma—only cold, hard facts.
- Implemented by Green Belts, Black Belts, and Master Black Belts.
- Ways to help meet the business/financial bottom-line numbers.

The Differences of DMAIC and DMADV

DMAIC and DMADV sound very similar, don't they? The acronyms even share the first three letters. But that's about where the similarities end.

When To Use DMAIC

The DMAIC methodology, instead of the DMADV methodology, should be used when a product or process is in existence at your company but is not meeting customer specification or is not performing adequately.

DMAIC	**Define**	• Define the project goals and customer (internal and external) deliverables
	Measure	• Measure the process to determine current performance
	Analyze	• Analyze and determine the root cause(s) of the defects
	Improve	• Improve the process by eliminating defects
	Control	• Control future process performance

When To Use DMADV

The DMADV methodology, instead of the DMAIC methodology, should be used when:

- A product or process is not in existence at your company and one needs to be developed
- The existing product or process exists and has been optimized (using either DMAIC or not) and still doesn't meet the level of customer specification or Six Sigma level

DMADV	**Define**	• Define the project goals and customer (internal and external) deliverables
	Measure	• Measure and determine customer needs and specifications
	Analyze	• Analyze the process options to meet the customer needs
	Design	• Design (detailed) the process to meet the customer needs
	Verify	• Verify the design performance and ability to meet customer needs

QUALITY MANAGEMENT TOOLS

Within the individual phases of a DMAIC or DMADV project, Six Sigma utilizes many established quality management tools that are also used outside of Six Sigma. The following list shows an overview of the main methods used:

1. 5 Whys
2. Analysis of variance
3. ANOVA Gauge R&R
4. Axiomatic design
5. Business Process Mapping
6. Catapult exercise on variability
7. Cause & effects diagram (also known as fishbone or Ishikawa diagram)
8. Chi-square test of independence and fits
9. Control chart
10. Correlation

11. Cost-benefit analysis
12. CTQtree
13. Quantitative marketing research through use of Enterprise Feedback Management (EFM) systems
14. Design of experiments
15. Failure mode and effects analysis
16. General linear model

IMPLEMENTATION ROLES

One of the key innovations of Six Sigma is the professionalizing of quality management functions. Prior to Six Sigma, quality management in practice was largely relegated to the production floor and to statisticians in a separate quality department. Six Sigma borrows martial arts ranking terminology to define a hierarchy (and career path) that cuts across all business functions and a promotion path straight into the executive suite.

Six Sigma identifies several key roles for its successful implementation.

1. Executive Leadership includes the CEO and other members of top management. They are responsible for setting up a vision for Six Sigma implementation. They also empower the other role holders with the freedom and resources to explore new ideas for breakthrough improvements.
2. Champions are responsible for Six Sigma implementation across the organization in an integrated manner. The Executive Leadership draws them from upper management. Champions also act as mentors to Black Belts.
3. Master Black Belts, identified by Champions, act as in-house coaches on Six Sigma. They devote 100% of their time to Six Sigma. They assist Champions and guide Black Belts and Green Belts. Apart from statistical tasks, their time is spent on ensuring consistent application of Six Sigma across various functions and departments.
4. Black Belts operate under Master Black Belts to apply Six Sigma methodology to specific projects. They devote 100% of their time to Six Sigma. They primarily focus on Six Sigma project execution, whereas Champions and Master Black Belts focus on identifying projects/ functions for Six Sigma.

5. Green Belts are the employees who take up Six Sigma implementation along with their other job responsibilities. They operate under the guidance of Black Belts.

ORIGIN AND MEANING OF THE TERM "SIX SIGMA PROCESS"

A graph of the normal distribution underlies the statistical assumptions of the Six Sigma model. The Greek letter σ (sigma) marks the distance on the horizontal axis between the mean, μ, and the curve's inflection point. The greater this distance is, the greater is the spread of values encountered. For the curve shown in Figure 9.9, $\mu = 0$ and $\sigma = 1$. The upper- and lower-specification limits (USL, LSL) are at a distance of 6σ from the mean. Due to the properties of the normal distribution, values lying that far away from the mean are extremely unlikely. Even if the mean were to move right or left by 1.5o at some point in the future (1.5 sigma shift), there is still a good safety cushion. This is why Six Sigma aims to have processes where the mean is at least 6σ away from the nearest specification limit.

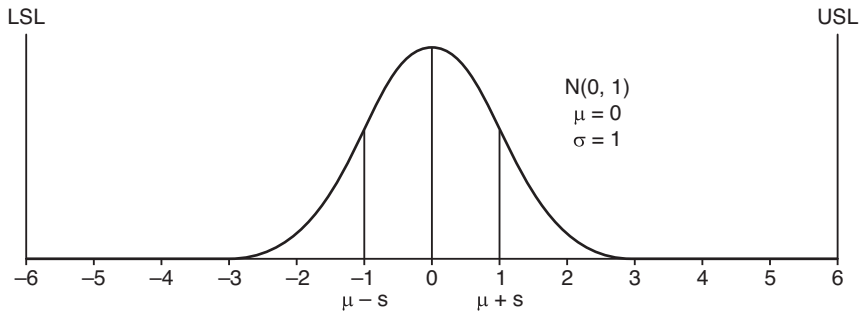


FIGURE 9.9

The term "Six Sigma process" comes from the notion that if one has six standard deviations between the process mean and the nearest specification limit, as shown in Figure 9.9, there will be practically no items that fail to meet specifications. This is based on the calculation method employed in process capability studies.

In a capability study, the number of standard deviations between the process mean and the nearest specification limit is given in sigma units. As process standard deviation goes up, or the mean of the process moves away from the center of the tolerance, fewer standard deviations will fit between the mean and the nearest specification limit, decreasing the sigma number and increasing the likelihood of items outside specification.

Role of the 1.5 Sigma Shift

Experience has shown that in the long term, processes usually do not perform as well as they do in the short. As a result, the number of sigma that will fit between the process mean and the nearest specification limit is likely to drop over time, compared to an initial short-term study. To account for this real-life increase in process variation over time, an empirically based 1.5 sigma shift is introduced into the calculation.

According to this idea, a process that fits Six Sigma between the process mean and the nearest specification limit in a short-term study will in the long term only fit 4.5 sigmas—either because the process mean will move over time, or because the long-term standard deviation of the process will be greater than that observed in the short term, or both.

Hence, the widely accepted definition of a six sigma process is one that produces 3.4 defective parts per million opportunities (DPMO). This is based on the fact that a process that is normally distributed will have 3.4 parts per million beyond a point that is 4.5 standard deviations above or below the mean (one-sided capability study). So the 3.4 DPMO of a "Six Sigma" process in fact corresponds to 4.5 sigma, namely 6 sigma minus the 1.5 sigma shift introduced to account for longterm variation. This is designed to prevent an underestimation of the defect levels likely to be encountered in real-life operation.

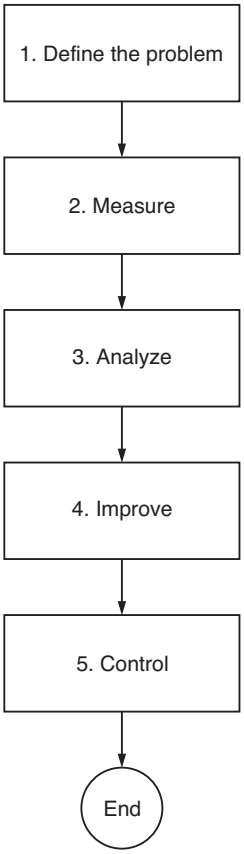
SIGMA LEVELS

The table below gives long-term DPMO values corresponding to various short-term sigma levels.

Note that these figures assume that the process mean will shift by 1.5 sigma towards the side with the critical specification limit. In other words, they assume that after the initial study determining the short-term sigma level, the long-term C_{pk} value will turn out to be 0.5 less than the short-term C_{pk} value. So, for example, the DPMO figure given for 1 sigma assumes that the long-term process mean will be 0.5 sigma beyond the specification limit ($C_{pk} = -0.17$), rather than 1 sigma within it, as it was in the short-term study ($C_{pk} = 0.33$). Note that the defect percentages only indicate defects exceeding the specification limit that the process mean is nearest to. Defects beyond the far specification limit are not included in the percentages.

Sigma level	DPMO	Percent defective	Percentage yield	Short-term Cp_k	Long-term C_{pk}
1	691,462	69%	31%	0.33	1-0.17
2	308,538	31%	69%	0.67	0.17
3	166,807	6.7%	93.3%	1.00	0.5
4	6,210	0.62%	99.38%	1.33	0.83
5	233	0.023%	99.977%	1.67	1.17
6	**3.4**	**0.00034%**	**99.99966%**	**2.00**	**1.5**
7	0.019	0.0000019%	99.9999981%	2.33	1.83

FLOW CHART



(A) Identify what the customer wants.
(B) Organize a team to improve the process.
(C) Create a flowchart of the process – SIPOC chart (Supplier-Input-Process-Output-Customer).

(A) Select measurable CTQC's (Critical To Quality Characteristics).
(B) Define performance standards.
(C) Validate the measurement system.
(D) Establish baseline performance in terms of sigma capability – defects per million opportunities.

(A) Identify significant characteristics and establish process capability.
(B) Define performance targets for significant characteristics.
(C) Identify the root cause of process variation.

(A) Identify and evaluate potential solutions.
(B) Implement short-term counter measures.
(C) Implement long-term corrective actions.
(D) Identify systemic indirect effects and unintended consequences of improvement actions.
(E) Establish operating tolerances for new processes.

(A) Verify corrective actions and validate new measurement systems.
(B) Determine process capability.
(C) Establish and implement a control plan.

Move on to next highest priority process.

FIGURE 9.10

BENEFITS OF SIX SIGMA

Obviously, in almost all cases, the reasons for Six Sigma failure have been external factors such as wrong or misguided selection of the tools, lack of application, and lack of support from upper management. It is important to keep in mind that the successful implementation of Six Sigma requires a top-down approach and perseverance throughout. Also important to the process is proper and thorough Six Sigma training.

1. Problem solving involves rational thinking. Somehow, companies always found themselves compromising quality with problem solving, which is the main reason why companies decide to implement Six Sigma and support Six Sigma training.
2. The top-down approach of Six Sigma requires dedication and application at all levels of the organization and on a continuous basis. The statistical methodology of Six Sigma sheds light on existing flaws and their causes after thorough analysis. Emphasis is placed on experimentation following analysis and redefining the processes and their goals. This is unlike other quality assurance methodologies. The benefits of supporting Six Sigma training for company professionals are apparent.
3. Financial benefits: Cash flow increases due to creation of additional revenue. Through this process although cost decreases and increased profitability can be seen. It is important that all professionals involved in Six Sigma implementation have proper Six Sigma training. Although Six Sigma training is relatively expensive, the financial benefits of supporting it greatly outweigh the upfront costs.
4. Operational benefits of Six Sigma training: Employee satisfaction due to improvement in work flow, reduction in process times and steps, better usage of work space, etc. result from implementation of Six Sigma. One major operational reason for choosing Six Sigma is its success in waste reduction and redundancy. Waste reduction is measured in terms of improving time, product movement, and decreasing material consumption.
5. Conceptually, the benefits of implementation of Six Sigma emerge from breaking the mindset that product processes are invariable. Benefits also emerge as a result of interconnected activities. The result of this methodical approach to quality management is evidenced by reduced fluctuations in processes. Stability of this kind triggers a series of positive chain reactions within organizations.

Success stories of Six Sigma training are evident in all fields of business. Because Six Sigma methodology encompasses the entire process of doing business, it is likely to show a flaw here or there, such as companies that embraced Six Sigma have found out. Howsoever small in number the failures may be, they are all due to differing reasons. However, any negative results can probably be traced back to either improper implementation or incomplete Six Sigma training.

CRITICISM OF SIX SIGMA

Six Sigma has made a huge impact on industry and is widely employed as a business strategy for achieving and sustaining operational and service excellence. However, there have also been various criticisms of Six Sigma.

Lack of Originality

Noted quality expert Joseph M. Juran has described Six Sigma as "a basic version of quality improvement," stating that "[t]here is nothing new there. It includes what we used to call facilitators. They've adopted more flamboyant terms, like belts with different colors. I think that concept has merit to set apart, to create specialists who can be very helpful. Again, that's not a new idea. The American Society for Quality long ago established certificates, such as for reliability engineers."

Role of Consultants

The use of "Black Belts" as itinerant change agents is controversial as it has created a cottage industry of training and certification. Critics argue there is overselling of Six Sigma by too great a number of consulting firms, many of which claim expertise in Six Sigma when they only have a rudimentary understanding of the tools and techniques involved.

The expansion of the various "Belts" to include "Green Belts," "Master Black Belts," and "Gold Belts" is commonly seen as a parallel to the various "belt factories" that exist in martial arts.

Potential Negative Effects

A Fortune article stated that "of 58 large companies that have announced Six Sigma programs, 91 percent have trailed the S&P 500 since." The statement is attributed to "an analysis by Charles Holland of consulting firm Qualpro (which espouses a competing quality-improvement process)." The gist of the article is that Six Sigma is effective at what it is intended to do, but that it is "narrowly designed to fix an existing process" and does not help in

"coming up with new products or disruptive technologies." Many of these claims have been argued as being in error or ill-informed.

Based on Arbitrary Standards

While 3.4 defects per million opportunities might work well for certain products/processes, it might not be ideal or cost-effective for others. A pacemaker process might need higher standards, for example, whereas a direct mail advertising campaign might need lower ones. The basis and justification for choosing 6 as the number of standard deviations is not clearly explained. In addition, the Six Sigma model assumes that the process data always conform to the normal distribution. The calculation of defect rates for situations where the normal distribution model does not apply is not properly addressed in the current Six Sigma literature.

Criticism of the 1.5 Sigma Shift

Because of its arbitrary nature, the 1.5 sigma shift has been dismissed as "goofy" by the statistician Donald J. Wheeler. Its universal applicability is seen as doubtful.

The 1.5 sigma shift has also been contentious because it results in stated "sigma levels" that reflect short-term rather than long-term performance: a process that has long-term defect levels corresponding to 4.5 sigma performance is, by Six Sigma convention, described as a "6 sigma process." The accepted Six Sigma scoring system thus cannot be equated to actual normal distribution probabilities for the stated number of standard deviations, and this has been a key bone of contention about how Six Sigma measures are defined. The fact that it is rarely explained that a "6 sigma" process will have long-term defect rates corresponding to 4.5 sigma performance rather than actual 6 sigma performance has led several commentators to express the opinion that Six Sigma is a confidence trick.

APPLICATION OF SIX SIGMA

India, as a fast growing economy, cannot afford to compromise on the quality of the products and services delivered. The demand for enhanced quality and reduced cost of the goods and services has coaxed many companies to introduce the Six Sigma system irrespective of the industry to which they belonged.

In India, Wipro was one of the first companies to introduce Six Sigma into its applications in 1997. Wipro's every business from software development to hardware, FMCG, BPO; all adopted Six Sigma into its processes. It is said that after the initiation of Six Sigma into its projects, the company has been able to complete 91% of its projects on time, when compared to the

industrial average of 55%. The metrics are used throughout the life cycle of the product, in production as well as in the software development. The need and importance of reducing defects have through the adoption of the Six Sigma has helped companies like Wipro to sustain SEI CMM Level 5.

In the pharmaceutical industry, adoption of the Six Sigma technique helped the industry reduce wastage and rework involved in the production. It was said that 5–10% of medicines produced during a period were to be discarded or modified due to the defects. The adoption of Six Sigma helped the pharmaceutical companies to reduce the errors in the production. The success story on the adoption of Six Sigma prompted Pfizer to achieve the Pfizer Global Manufacturing mission of aiming at zero defects through Right First Time. Right First Time or RFT is a technique adapted by Pfizer to its core processes to assure quality to its products and customer services the first time.

The airline industry had to adopt the Six Sigma metrics for its survival. The increased cost of fuel, the competition driven by low budget airlines, etc has made the need for lower cost without a hit to quality the need of the hour. The number of errors in handling the calls from customers, and ticketing is to be minimized drastically. It was with this intention that the airline industry adopted Six Sigma into the organization. Indian companies like Kingfisher and Jet Airways, all have adopted Six Sigma techniques.

Hospitality services are another industry which benefited by the adoption of Six Sigma techniques. Providing personalized service to each and every customer by bending to their demands within a limited time without comprising the quality was aided by the Six Sigma metrics. The Six Sigma technique is adopted in every field from maintaining full occupancy to efficient housekeeping, ensuring a balanced inventory supply, and to minimize wastage in the inventory. Starwood Hotels and Resorts Inc was the first company to adopt Six Sigma in the hospitality sector.

Steel industries like TISCO use this technique to minimize the inadequacies in the design, imperfect products, etc. In 1998, Forbes Magazine applauded the Mumbai Tiffin Suppliers Association or the Mumbai Dabbawallahs for their way of functioning with just one error in every 8 million. Logistics, insurance, call centers, all embrace the Six Sigma techniques for improving the quality of service provided by them.

Irrespective of the type of industry, all companies have to adopt Six Sigma techniques as quality and timely delivery are crucial for their survival.

THE FUTURE OF SIX SIGMA

Since its introduction in the 1990s, Six Sigma has become the buzzword in both the manufacturing and service industries. The various methodologies used in Six Sigma are based on a disciplined and data driven approach that

help in eliminating defects and achieving near perfection by restricting the number of possible defects to less than 3.4 defects per million. The methodologies are effective in managing business processes of both the manufacturing and service industries. In manufacturing industries, the concepts and methodologies are used for reducing the number of defects whereas in service industries, they are used mainly for reducing transactional errors.

Six Sigma may appear similar to other quality management tools such as TQM or Kaizen Events, but in reality, it is quite different. Other quality management programs often reach a stage after which no further quality improvements can be made. Six Sigma, on the other hand, is different as it focuses on taking quality improvement processes to the next level. This means that Six Sigma has the potential to outlast other quality management programs in the future.

The scope of Six Sigma is also much broader than other quality management programs as it can be applied to every business process of an organization. The future is bright for Six Sigma programs with the growing awareness in small and medium enterprises about the potential benefits that can be derived from implementing such programs.

9.6. QUALITY CHALLENGE

As we know software quality is conformance to explicitly stated and agreed functional and nonfunctional requirements. Some challenges of quality are as follows:

1. Software cannot be seen, touched, or heard. So, it is virtual in nature. Our correctional measuring methods cannot measure them.
2. There is a big communication gap between users and developers/testers of software is another quality challenge.
3. Customer requirements as stated in SRS are many times not clear and inconsistent.
4. The need is to shift focus from quality control to quality management.
5. Any level of exhaustive testing is not capable of testing each and every branch, condition, and algorithm. Quality testing is now needed.
6. Project-level objectives must be in synchronization with organizational-level objectives.
7. Certain SDLC phases do not include quality as an important parameter. So, complete quality engineering is to be followed.

8. Total quality management (TQM) costs more.
9. Expert personnel are not available.

9.7. NATIONAL QUALITY AWARDS

- **Deming Awards** were the first national quality awards (in Japan). These are presented every year to individuals, groups, and companies which achieve notable quality levels or improvements.
- The **Baldrige Awards** were introduced (in the United States). US companies submit themselves to the assessment process to find out whether they achieve the highest scores in a wide range of companies today.
- In Europe, the European Foundation of Quality Management (EFQM) has produced a self-assessment guide which companies can use to conduct their own evaluations. A comparison of TQ with Deming principles and with Baldrige is given below:

TABLE 9.1 A Comparison for the House of Total Quality with the Baldrige Categories and Deming Principles

House of Total Quality[a]	Baldrige Categories[b]	Deming Principles[c]
The Roof		
Management System		
1. Systems, process 2. Leadership 3. Strategy 4. Mission, vision, values	SYSTEM 1.0 Leadership 3.0 Strategic quality planning * Long-range planning	1. Publish the aims and purpose of the organization 2. Learn the new philosophy 7. Teach and institute leadership
Social System		
1. Structure 2. Social norms 3. Teams 4. Organizational personality	4.0 Human resources development and management • Employee development • Partnership development • Cross-functional teams	14. Take action to accomplish the transformation • Hierarchy style of management must change • Transformation can only be accomplished by people not hardware

(*Continued*)

House of Total Quality[a]	Baldrige Categories[b]	Deming Principles[c]
Technical System		
1. Work processes 2. Job descriptions 3. Problem-solving tools 4. Decision making 5. Measurement tools	5.0 Processes management • Reduced cycle time • Design quality and QFD	9. Optimize efforts of teams, groups, staff toward aims arid purposes * Statistical measurement
Educational System		
1. Lifelong learning 2. Retraining	4.0 Human resource development and management * Employee participation and development	• Continual learning 6. Institute training
The Pillars		
Customer satisfaction	7.0 Customer focus and satisfaction	4. End the practice of awarding business on price tag alone • Aim quality at the needs of the customer, present and future
Continuous improvement	2.0 Information and analysis 5.0 Process management 6.0 Business results	3. Understand the purpose of inspection-improve process, reduce costs 11a-b. Eliminate quotas & MBO. Concentrate on improvement
Speak with facts	2.0 Information and analysis * Management by fact	3. Understand the purpose of inspection—improve processes, reduce costs • In God we trust; all others bring facts • Statistical measurement
Respect for people	4.0 Human resource development and management • Job design • Compensation and management • Employee well-being and satisfaction	14. Take action to accomplish the transformation 11a. Eliminate numerical quotas 11. Eliminate MBO 12. Remove barriers that rob people of pride in workmanship

House of Total Quality[a]	Baldrige Categories[b]	Deming Principles[c]
Ethical Work Culture		
Ethical work culture	1.0 Leadership 4.0 Human resource development and management • Manage for organizational integrity	8. Drive out fear 7. Tech. and institute leadership * Create and maintain system integrity

SUMMARY

Implementation and use of software quality standards is as important as is software development. Quality assurance is an umbrella activity, i.e., it must be dealt with during the entire SDLC. Better standards, tools, metrics, and quality models are still the need of researchers. This chapter focuses on all of these aspects.

MULTIPLE CHOICE QUESTIONS

1. According to P. Crosby:

a. Quality is fitness for use.
b. Quality is free.
c. Quality is degree of excellence.
d. None of the above.

2. ISO 9126 standard uses:

a. Hierarchical method
b. Relational method
c. Network method
d. None of the above.

3. The classification of quality is done in:

a. CMM
b. CMMI
c. Kano's Model
d. None of the above.

4. The process used by an organization to identify, understand, adapt, and adopt outstanding practices and processes from others to help that organization improve its processes is known as:

a. Watermarking
b. Benchmarking
c. Levelmarking
d. None of the above.

5. In CMMI, "I" stands for:
   a. Internet
   b. Integration
   c. Intranet
   d. None of the above.
6. ZD stands for:
   a. Zero deflection
   b. Zero defect
   c. Zero demand
   d. None of the above.
7. The first national quality awards were introduced by:
   a. Flemming
   b. Deming
   c. Keming
   d. None of the above.
8. In Europe, approach used is:
   a. TQM
   b. EFQM
   c. QM
   d. None of the above.
9. Quality is also related to:
   a. Freedom
   b. Under-estimation
   c. Estimation
   d. None of the above.
10. The cost incurred due to some kind of failure is known as:
    a. White money
    b. Red money
    c. Blue money
    d. Gray money
11. TQM has made:
    a. PDCA cycle popular
    b. SDLC cycle popular
    c. PDLC cycle popular
    d. None of the above.
12. The shift is from quality control to:
    a. Quality management
    b. Quality testing
    c. Quality assurance
    d. None of the above.
13. A roadmap for software project development is:
    a. SCM
    b. SQA
    c. SDLC
    d. None of the above.

14. To realize an organization's financial objectives of increasing profits, the technique used is:

a. Six Sigma **b.** Benchmarking

c. SQA **d.** None of the above.

15. A classification schemes to assess an organization's ability to develop quality software is:

a. CMM **b.** CMMI

c. MTTF **d.** None of the above.

ANSWERS

1. b.	**2.** a.	**3.** c.	**4.** b.
5. b.	**6.** b.	**7.** b.	**8.** b.
9. a.	**10.** b.	**11.** a.	**12.** a.
13. a.	**14.** a.	**15.** a.	

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

Q. 1. Distinguish between CMM and ISO 9001.

Ans.

CMM	ISO 9001
1. It is specific to software development.	**1.** It is intended as generic.
2. Used in USA, mostly.	**2.** Accepted in most countries.
3. Assesses on 5-levels.	**3.** Has one acceptable level.
4. No time limit.	**4.** Certification is valid for three years.
5. Provides detailed and specific definitions of what is required for given levels.	**5.** Specifies concepts, principles, and safeguards that should be in place.

Q. 2. What are the problems associated with ISO 9001?

Ans. A common problem of ISO 9001 is the amount of money, time, and paperwork required for registration.

Opponents: Claim that it is only for documentation
Proponents (in favor): Claim that if a company has documented its quality systems, then most of the paperwork has already been completed [Barnes].
ISO 9001 promotes specification, control, and produces rather than understanding and improvement [Seddon].
Wade argues that ISO 9001 is effective as a guideline but promoting it as a standard helps to mislead companies into thinking that certification means better quality.
Blind reliance on ISO 9001 does not guarantee a successful quality system.
Competition among the numerous certifying bodies is another problem.

Q. 3. What is quality of conformance?
Ans. It is the degree to which design speciations are followed during SDLC.

Q. 4. Define quality control.
Ans. It is the series of inspections, reviews, and tests throughout SDLC to ensure that each work product meets the requirements set from it.

Q. 5. What is quality cost?
Ans. It includes all costs incurred in maintaining quality of the software and performing quality-related activities.

Q. 6. What are the factors involved in prevention cost?
Ans. **1.** Quality planning, **2.** FTRs, **3.** Test equipment, **4.** Quality training

Q. 7. Define maturity.
Ans. It is a measure of the goodness of the organization's process.

Q. 8. What is zero defect day?
Ans. To create an event that will let all employees realize through a personal experience that there has been a change.

Q. 9. What is CPI under TQM umbrella?
Ans. CPI stands for constant process improvement. It is defined as a systematic approach taken by all employees to achieve the highest levels of quality and competitiveness through continuous improvements of operations, products, and services.

Q. 10. What are quality circles?

Ans. Volunteer group of workers, usually members of the same department, who meet regularly to discuss the problems and make presentations to management with their ideas to solve the problems. Quality circles started in Japan in 1962 by ka or Ishikawa as another method of improving quality.

REVIEW QUESTIONS

1. What is quality modeling?
2. "If a software system is understandable it would definitely be maintable also." Justify this statement.
3. Distinguish between quality control and quality assurance.
4. What testing processes are likely to enhance software quality?
5. Give Dr. Demings and Philip Crosby's definitions of quality.
6. Give a quality definition as per international standards.
7. Compare Boehm's and ISO 9/26 quality models.
8. Explain CMM. How is CMMI different?
9. What are the factors considered for a quality software product?
10. What is Six Sigma quality? What are its organization structures?

CHAPTER 10

REDUCING THE NUMBER OF TEST CASES

Inside this Chapter:

10.0. Prioritization Guidelines
10.1. Priority Category Scheme
10.2. Risk Analysis
10.3. Regression Testing—Overview
10.4. Prioritization of Test Cases for Regression Testing
10.5. Regression Testing Technique—A Case Study
10.6. Slice Based Testing

10.0. PRIORITIZATION GUIDELINES

The goal of prioritization of test cases is to reduce the set of test cases based on some rational, non-arbitrary, criteria, while aiming to select the most appropriate tests. If we prioritize the test cases, we run the risk that some of the application features will not be tested at all.

The prioritization schemes basically address these key concepts:

a. What features must be tested?

b. What are the consequences if some features are not tested?

The main guide for selecting test cases is to assess the risk first. At the end of the test case prioritization exercise, the tester and the project authority must feel confident with the tests that have been selected for execution. If someone is distressed about the omitted tests, then re-evaluate the list and apply another prioritization scheme to analyze the application from another point of view. Ideally, the project authority and possibly other project members, must buy-in—and sign off—to the prioritized set of tests.

There are four schemes that are used for prioritizing the existing set test cases. These reduction schemes are as follows:

1. Priority category scheme
2. Risk analysis
3. Interviewing to find out problematic areas
4. Combination schemes

All of these reduction methods are independent. No one method is better than the other. One method may be used in conjunction with another one. It raises confidence when different prioritization schemes yield similar conclusions.

We will discuss these techniques now.

10.1. PRIORITY CATEGORY SCHEME

The simplest scheme for categorizing tests is to assign a priority code directly to each test description. One possible scheme is a three-level priority categorization scheme. It is given below:

Priority 1: This test must be executed.
Priority 2: If time permits, execute this test.
Priority 3: If this test is not executed, the team won't be upset.

So, assigning a priority code is as simple as writing a number adjacent to each test description. Once priority codes have been assigned, the tester estimates the amount of time required to execute the tests selected in each category. If the time estimate falls within the allotted schedule, then the partitioning exercise is completed and you have identified the tests to use. Otherwise, further partitioning is required.

The second scheme can be a new five-level scale to further classify the tests. This scheme is given below:

Priority 1a: This test must pass or the delivery will be out of date.
Priority 2a: Before final delivery, this test must be executed.
Priority 3a: If time permits, execute this test.
Priority 4a: This test can wait after the delivery date.
Priority 5a: We will probably never execute this test.

We try to divide the tests. For example, say tests from priority 2 are now divided between priorities 3a, 4a, and 5a, we can downgrade or upgrade any test in the similar fashion.

10.2. RISK ANALYSIS

All software projects benefit from risk analysis. Even non-critical software, using risk analysis at the beginning of a project highlights the potential problem areas. This helps developers and managers to mitigate the risks. The tester uses the results of risk analysis to select the most crucial tests.

How Risk Analysis Is Done?

Risk analysis is a well-defined process that prioritizes modules for testing. A risk contains three components:

1. The risk, r_i, associated with a project ($i \leftarrow 1$ to n).
2. The probability of occurrence of a risk, (l_i).
3. The impact of the risk, (x_i).

Risk analysis consists of first listing the potential problems and then assigning a probability and severity value for each identified problem. By ranking the results, the tester can identify the potential problems most in need of immediate attention and select test cases to address those needs. During risk analysis we draw a table as shown below:

Problem ID	Potential problem (r_i)	Probability of occurrence (l_i)	Impact of risk (x_i)	Risk exposure = $l_i * x_i$
A	Loss of power	1	10	10
B	Corrupt file header	2	1	2
C	Unauthorized access	6	8	48
D	Databases not synchronized	3	5	15
E	Unclear user documentation	9	1	9
F	Lost sales	1	8	8
G	Slow throughput	5	3	15
:	:	:	:	:
:	:	:	:	:
:	:	:	:	:

FIGURE 10.1 Risk Analysis Table (RAT).

Here, in Figure 10.1,

Problem ID. It is a unique identifier associated with a risk.
Potential problem. It is a brief description of the problem.
Probability of occurrence, (l_i). It is a probability value on a scale of 1 (low) to 10 (high).
Severity of impact, (x_i). It is a severity value on a scale of 1 (low) to 10 (high).
Risk exposure. It is defined as the product of l_i and x_i.

In this table, the values of l_i and x_i range from 1 to 10. Multiplying the probability and severity values yields the risk exposure. *"The higher the risk exposure product, the more important it is to test for that condition."* Applying this rule to Figure 10.1, we will get the following rankings of the potential risk problems based on the product of risk-exposure. The order of preference is:

C—D—G—A—E—F—B

Although problems D and G have the same risk exposure, they differ by their probability and severity values.

For some organizations, this method may produce enough information. Others may wish to do risk-matrix analysis also. We will now discuss risk matrix analysis.

What Is a Risk Matrix?

A risk matrix allows the tester to evaluate and rank potential problems by giving more weight to the probability or severity value as necessary. The software tester uses the risk matrix to assign thresholds that classify the potential problems into priority categories.

There are four main methods of forming risk matrices:

Method I: Typically, the risk matrix contains four quadrants, as shown in Figure 10.2.

Each quadrant in Figure 10.2 represents a priority class defined as follows:

Priority 1: High severity and high probability
Priority 2: High severity and low probability
Priority 3: Low severity and high probability
Priority 4: Low severity and low probability

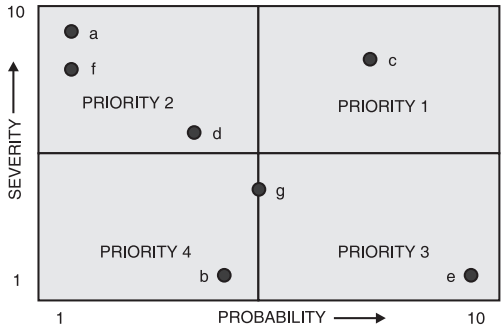


FIGURE 10.2 Method I.

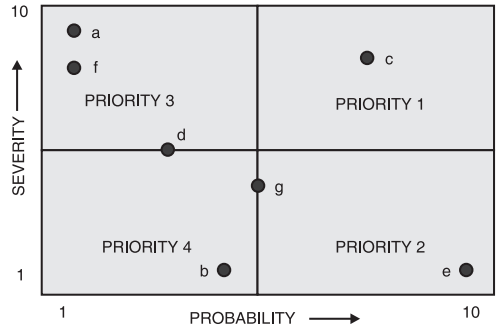


FIGURE 10.3 Method II.

We can see from the graph of Figure 10.2 that a risk with high severity is deemed more important than a problem with high probability. Thus, all risks mapped in the upper-left quadrant fall into priority 2.

For example, the risk-e which has a high probability of occurrence but a low severity of impact is put under priority 3.

Method II: For an entirely different application, we may swap the definitions of priorities 2 and 3, as shown in Figure 10.3.

An organization favoring Figure 10.3 seeks to minimize the total number of defects by focusing on problems with a high probability of occurrence.

Dividing a risk matrix into quadrants is most common, testers can determine the thresholds using different types of boundaries based on application specific needs.

Method III: Diagonal band prioritization scheme.

If severity and probability tend to be equal weight, i.e., if $l_i = x_i$, then diagonal band prioritization scheme may be more appropriate. This is shown in Figure 10.4.

This threshold pattern is a compromise for those who have difficulty in selecting between priority-2 and priority-3 in the quadrant scheme.

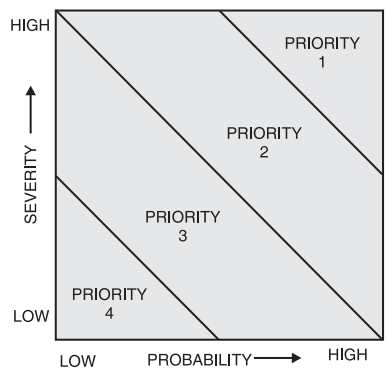


FIGURE 10.4 Method III.

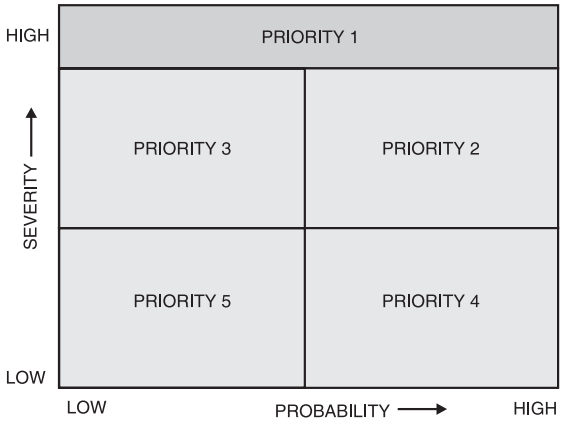


FIGURE 10.5 Method IV.

Method IV: The problems with high severity must be given the top priority, irrespective of the value of probability. This problem is solved with method-IV and is shown in Figure 10.5. The remainder of the risk matrix is partitioned into several lower priorities, either as quadrants (Method-I and -II) or as diagonal bands (Method-III).

10.3. REGRESSION TESTING—OVERVIEW

Regression testing is done to ensure that enhancements or defect fixes made to the software works properly and does not affect the existing functionality. It is usually done during maintenance phase.

As a software system ages, the cost of maintaining the software dominates the overall cost of developing the software. Regression testing is a testing process that is used to determine if a modified program still meets its specifications or if new errors have been introduced. Improvements in the regression testing process would help reduce the cost of software.

10.3.1. DIFFERENCES BETWEEN REGRESSION AND NORMAL TESTING

Let us now compare normal and regression testing in the below Table.

Normal testing	Regression testing
1. It is usually done during fourth phase of SDLC.	**1.** It is done during the maintenance phase.
2. It is basically software's verification and validation.	**2.** It is also called program revalidation.
3. New test suites are used to test our code.	**3.** Both old and new test cases can be used for testing.
4. It is done on the original software.	**4.** It is done on modified software.
5. It is cheaper.	**5.** It is a costlier test plan.

10.3.2. TYPES OF REGRESSION TESTING

Four types of regression testing techniques are discussed one-by-one. They are

1. Corrective regression testing
2. Progressive regression testing
3. Retest-all regression testing
4. Selective regression testing

10.3.2.1. CORRECTIVE REGRESSION TESTING

Corrective regression testing applies when specifications are unmodified and test cases can be reused.

10.3.2.2. PROGRESSIVE REGRESSION TESTING

Progressive regression testing applies when specifications are modified and new test cases must be designed.

10.3.2.3. THE RETEST-ALL STRATEGY

The retest-all strategy reuses all tests, but this strategy may waste time and resources due to the execution of unnecessary tests. When the change to a system is minor, this strategy would be wasteful.

10.3.2.4. THE SELECTIVE STRATEGY

The selective strategy uses a subset of the existing test cases to reduce the retesting cost. In this strategy, a test unit must be rerun if and only if any of the program entities, e.g., functions, variables etc., it covers have been changed. The challenge is to identify the dependencies between a test case and the program entities it covers. Rothermel and Harrold specified a typical selective regression testing process. In general, the process includes the following steps:

Step 1. Identify affected software components after program, P, has been modified to P′.

Step 2. Select a subset of test cases, T′, from an existing test suite, T, that covers the software components that are affected by the modification.

Step 3. Test modified program P′ with T′ to establish the correctness of P′ with respect to T′.

Step 4. Examine test results to identify failures.
Step 5. Identify and correct the fault(s) that caused a failure.
Step 6. Update the test suite and test history for P′.

From these steps, we can observe the following characteristics of selective regression testing:

1. Identifying the program components that must be retested and finding those existing tests that must be rerun are essential.
2. When selected test cases satisfy retest criterion, new test cases are not needed.
3. Once regression testing is done, it is necessary to update and store the test information for reuse at a later time.

Regression testing is one kind of testing that is applied at all three levels of testing. White suggested applying specification based (system) testing before structure based (unit) testing to get more test case reuse because system testing cases could be reused in unit testing but the converse is not true. McCarthy suggested applying regression unit testing first to find faults early.

10.3.2.5. Regression testing at unit level

Unit testing is the process of testing each software module to ensure that its performance meets its specifications. Yau and Kishimoto developed a method based on the input partition strategy. McCarthy provided a way to automate unit regression testing.

Gupta, Harrold, and Soffa proposed an approach to data flow based regression testing.

10.3.2.6. Regression testing at integration level

Integration testing is the testing applied when individual modules are combined to form larger working units until the entire program is created. Integration testing detects failures that were not discovered during unit testing. Integration testing is important because approximately 40% of software errors can be traced to module integration problems discovered during integration testing.

Leung and White introduced the firewall concept to regression testing at the integration level. A firewall is used to separate the set of modules affected

by program changes from the rest of the code. The modules enclosed in the firewall could be those that interact with the modified modules or those that are direct ancestors or direct descendants of the modified modules.

The firewall concept is simple and easy to use, especially when the change to a program is small. By retesting only the modules and interfaces inside the firewall, the cost of regression integration testing can be reduced.

10.3.2.7. REGRESSION TESTING AT SYSTEM LEVEL

System testing is testing of the entire software system against the system specifications. It must verify that all system elements have been properly integrated and perform allocated functions. It can be performed without the knowledge of the software implementation at all.

Test Tube is a system developed at AT&T Bell laboratories to perform system level regression testing. Test Tube is an example of a selective retesting technique. Test Tube partitions a software system into basic code entities, then monitors the execution of a test case, analyzes its relationship with the system under test, and determines which subset of the code entities the test covers. There are plans to extend Test Tube to non-deterministic systems such as real-time telecommunications software.

10.3.2.8. REGRESSION TESTING OF GLOBAL VARIABLES

A global variable is an output parameter in a module where the variable is defined and an input parameter for a module that uses the variable. The global variables can be retested at the unit level by running the test that exercise the changed code and the instruction referencing the global variable.

A global variable can be retested at the integration level. If any of its defining modules have been changed then all of its using modules must be retested. The regression testing of global variables is very time consuming and, therefore, costly.

10.3.2.9. COMPARISON OF VARIOUS REGRESSION TESTING TECHNIQUES

There are four criteria that form a framework for evaluation of different selective regression testing techniques. They are discussed below:

1. *Inclusiveness*: It measures the extent to which a technique chooses tests that will expose faults caused by program changes.
2. *Precision*: It measures the ability of a technique to avoid choosing tests that will not reveal the faults caused by the changes.

3. *Efficiency*: It measures the computational cost of a technique.
4. *Generality*: It measures the ability of a technique to handle different language constructs and testing applications.

10.3.2.10. Regression testing in object-oriented software

Object-oriented concepts such as inheritance and polymorphism present unique problems in maintenance and, thus, regression testing of object-oriented programs. Several regression testing techniques have been extended to retesting of object-oriented software.

Rothermal and Harrold extended the concept of selective regression testing to object-oriented environments. Their algorithm constructs program or class dependence graphs and uses them to determine which test cases to select. To test a graph, driver programs are developed to invoke the methods in a class in different sequences. The class dependence graph links all these driver programs together by selecting one driver as the root of the graph and adding edges from it to the public methods in the class. Now the methods can be invoked in different sequences.

Abdullah and White extended the firewall concept to retesting object-oriented software. A firewall is an imaginary boundary that encloses the entities that must be retested in a modified program. Unlike the firewall in a procedure-based program, the firewall in an object-oriented program is constructed in terms of classes and objects. When a change is made to a class or an object, the other classes or objects that interact with this changed class or object must be enclosed in the firewall. Because the relations between classes and objects are different, there should be different ways to construct the firewalls for classes and objects.

10.4. PRIORITIZATION OF TEST CASES FOR REGRESSION TESTING

Prioritization of tests requires a suitable cost criterion. Please understand that tests with lower costs are placed at the top while those with higher costs at the bottom.

What Cost Criterion to Use?

We could use multiple criteria to prioritize tests. Also, note that the tests being prioritized are the ones selected using some test selection technique. Thus, each test is expected to traverse at some modified portion of P′.

Prioritization of regression tests offers a tester an opportunity to decide how many and which tests to run under time constraints. When all tests cannot be run, one needs to find some criteria to decide when to stop testing. This decision could depend on a variety of factors such as:

a. Time constraints

b. Test criticality

c. Customer requirements

10.5. REGRESSION TESTING TECHNIQUE—A CASE STUDY

Regression testing is used to confirm that fixed bugs have been fixed and that new bugs have not been introduced. How many cycles of regression testing are required will depend upon the project size. Cycles of regression testing may be performed once per milestone or once per build. Regression tests can be automated.

The Regression–Test Process

The regression-test process is shown in Figure 10.6.

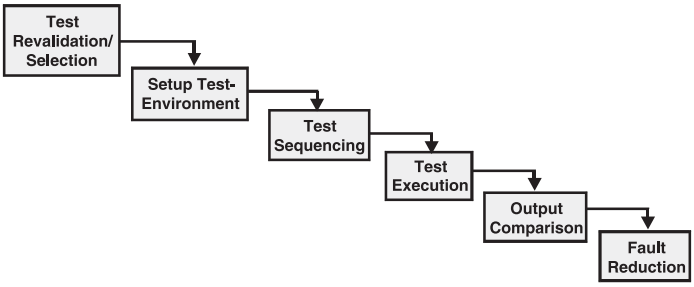


FIGURE 10.6

This process assumes that P′ (modified is program) available for regression testing. There is a long series of tasks that lead to P′ from P.

Test minimization ignores redundant tests. For example, if both t_1 and t_2, test function, f in P, then one might decide to reject t_2 in favor of t_1. The purpose of minimization is to reduce the number of tests to execute for regression testing.

Test prioritization means prioritizing tests based on some criteria. A set of prioritized tests becomes useful when only a subset of tests can

be executed due to resource constraints. Test selection can be achieved by selecting a few tests from a prioritized list. The possible sequence to execute these tasks is:

Test setup means the process by which AUT (application under test) is placed in its intended or simulated environment and is ready to receive data and output the required information. Test setup becomes more challenging when we test embedded software like in ATMs, printers, mobiles, etc.

The sequence in which tests are input to an application is an important issue. Test sequencing is very important for applications that have an internal state and runs continuously. For example, an online-banking software.

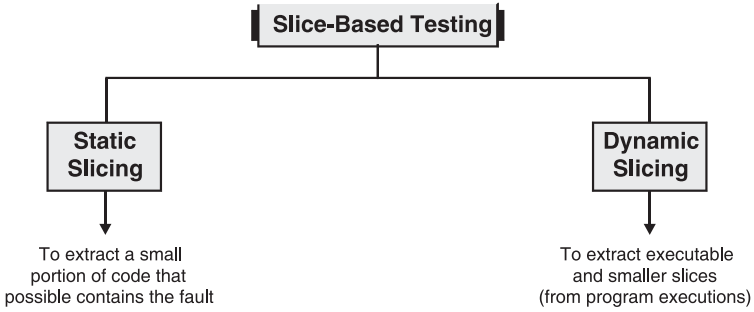
We then execute the test cases. Each test needs verification. This can also be done automatically with the help of CASE tools. These tools compare the expected and observed outputs. Some of the tools are:

a. **Test Tube (by AT&T Bell Labs.) in 1994:** This tool can do selective retesting of functions. It supports C.

b. **Echelon (by Microsoft) in 2002:** No selective retesting but does test prioritization. It uses basic blocks to test. It supports C and binary languages.

c. **ATACLx Suds (by Telcordia Technologies) in 1992:** It does selective retesting. It allows test prioritization and minimization. It does control/data flow average. It also supports C.

10.6. SLICE-BASED TESTING



Static slicing may lead to an unduly large program slice. So, Korel and Laski proposed a method for obtaining dynamic slices from program executions. They used a method to extract executable and smaller slices and to allow

more precise handling of arrays and other structures. So, we discuss dynamic slicing.

Let "P" be the program under test and "t" be a test case against which P has been executed. Let "l" be a location in P where variable v is used. Now, the dynamic slice of P with respect to "t" and "v" is the set of statements in P that lie in trace (t) and did effect the value of "v" at "l." So, the dynamic slice is empty if location "l" was not traversed during this execution. Please note that the notion of a dynamic slice grew out of that of a static slice based on program "P" and not on its execution.

Let us solve an example now.

EXAMPLE 10.1. Consider the following program:

```
1.  main ( ) {
2.  int p, q, r, z;
3.  z = 0;
4.  read (p, q, r);
5.  if (p < q)
6.  z = 1; //modified z
7.  if (r < 1)
8.  x = 2
9.  output (z);
10. end
11. }
```

Test case (t1): <p = 1, q = 3, r = 2>. What will be the dynamic slice of P with respect to variable "z" at line 9? What will be its static slice? What can you infer? If t_2: <p = 1, q = 0, r = 0> then what will be dynamic and static slices?

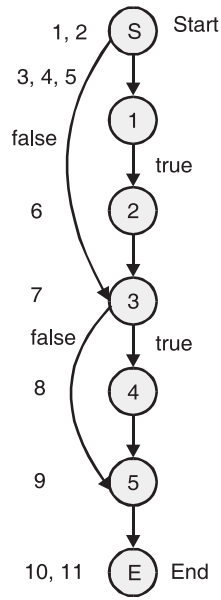


FIGURE 10.7

SOLUTION. Let us draw its flow graph first shown in Figure 10.7.

∴ Dynamic slice (P) with respect to variable z at line 9 is t_d = <4, 5, 7, 8>
Static slice, t_s = <3, 4, 5, 6, 7, 8>

NOTE *Dynamic slice for any variable is generally smaller than the corresponding static slice.*

Now, t_2: <p = 1, q = 0, r = 0>
∴ Its dynamic slice is statements (3, 4, 5) while the static slice does not change.

NOTE *Dynamic slice contains all statements in trace (t) that had an effect on program output.*

Inferences mode:

1. A dynamic slice can be constructed based on any program variable that is used at some location in P, the program that is being modified.
2. Some programs may have several locations and variables of interest at which to compute the dynamic slice, then we need to compute slices of all such variables at their corresponding locations and then take union of all slices to create a combined dynamic slice. This approach is useful for regression testing of relatively small components.
3. If a program is large then a tester needs to find out the critical locations that contain one or more variables of interest. Then, we can build dynamic slices on these variables.

SUMMARY

Regression testing is used to confirm that fixed bugs have, in fact, been fixed and that new bugs have not been introduced in the process and that features that were proven correctly functional are intact. Depending on the size of a project, cycles of regression testing may be performed once per milestone or once per build. Some bug regression testing may also be performed during each acceptance test cycle, focusing on only the most important bugs. Regression tests can be automated.

MULTIPLE CHOICE QUESTIONS

1. The main guide for selecting test cases is
   - **a.** To assess risks.
   - **b.** To assess quality.
   - **c.** Both (a) and (b)
   - **d.** None of the above.

2. Which of the following is not a risk reduction scheme?
   a. Priority category scheme
   b. Risk analysis
   c. Interviewing
   d. None of the above.
3. The potential problems are identified and their probabilities and impacts are given weights in which technique.
   a. Priority categorization scheme
   b. Risk analysis
   c. Interviews
   d. None of the above.
4. Risk exposure is given by which formula.
   a. It is the product of probability of occurrence of risk and its impact
   b. It is the sum of probability of its occurrence and its impact
   c. It is the standard deviation of the sum of its probability and its impact
   d. None of the above.
5. A risk matrix
   a. Allows the testers to evaluate and rank potential problems by giving weights to proabability or severity value as necessary.
   b. Allows testers to assign thresholds that classify the potential problems into priority categories.
   c. Both (a) and (b).
   d. None of the above.
6. In diagonal band prioritization scheme
   a. Severity equals probability.
   b. Severity is never equal to probability.
   c. Either (a) or (b).
   d. Both (a) and (b).

7. Some managers found out that
   a. Probability of risk to occur is very important.
   b. Problems with high severity must be given top priority.
   c. Severity of risk is not at all important.
   d. None of the above.
8. Which one of the following is NOT a regression testing strategy?
   a. Correction regression testing
   b. Retest-all strategy
   c. Combinational explosion
   d. None of the above.
9. Which of the following testing strategy is applicable at all three levels of testing?
   a. White-box testing
   b. Mutation testing
   c. Regression testing
   d. None of the above.
10. What is used to separate a set of modules affected by program changes from the rest of the code?
   a. Firewall
   b. NIC
   c. Emulator
   d. None of the above.

ANSWERS

1. a.	**2.** d.	**3.** b.	**4.** a.
5. c.	**6.** a.	**7.** b.	**8.** c.
9. c.	**10.** a.		

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

Q. 1. Explain the risk reduction method?

Ans. The formula for quantifying risk also explains how to control or minimize risk. The formula is as follows:

Loss Due to Risk = Frequency of Occurrence × Loss Per Occurrence

For example, say ten users a day terminated their business website. The average customer places $80 orders and thus,
Loss due to unfriendly website is = 10 * $80 = $800
Therefore, once the variables and the loss expectation formula have been defined, controls can then be identified to minimize that risk.

Q. 2. Can you list the most common risk factors for various project types given below:

a. MIT
b. Commercial Software Sectors

Ans. **a. For MIS Projects**

Risk Factors	% of Projects at Risk
1. Creeping user requirements	80%
2. Excessive schedule pressure	65%
3. Low quality	60%
4. Cost overruns	55%
5. Inadequate configuration control	50%

b. For Commercial Software Sectors

Risk Factors	% of Projects at Risk
1. Inadequate user documentation	70%
2. Low-user satisfaction	55%
3. Excessive time to market	50%
4. Harmful competitive actions	45%
5. Litigation expense	30%

Q. 3. What is SEI-SRE service?

Ans. The Software Engineering Institute (SEI) defines a Software Risk Evaluation (SRE) service.
SRE is a diagnostic and decision-making tool that enables the identification, analysis, tracking, mitigation, and communications of risks in software-intensive programs. It is used to identify and categorize

specific program risks arising from product, process, management, resources, and constraints.
An SRE can be used for:

a. Preparing for a critical milestone.
b. Recovery from crisis.

An SRE is conducted by a trained SEI-program team. By implementing SRE, the management improves its ability to ensure success through the creation of a proactive risk management methodology.

Q. 4. Why is smoke testing done?

Ans. Smoke testing is an integration testing approach that constructs and tests software on a daily basis. It consists of:

a. Finding out the basic functionality that a product must satisfy.

b. Designing test cases to ensure that the basic functionality work and combining these test cases into a smoke test suite.

c. To ensure that everytime a product is built, this suite is run first.

d. If this suite fails then escalate to the developers to find out the changes or to roll back the changes to a state where the smoke test suite succeeds.

Q. 5. How can regression testing be done at the module level and at the product level?

Ans. **a.** At a module level, it may involve retesting module-execution-paths (MEPs) traversing the modification.

b. At a product level, this activity may involve retesting functions that execute the modified area.

The effectiveness of these strategies is highly dependent on the utilization of test matrices, which enable identification of coverage provided by particular test cases.

REVIEW QUESTIONS

1. What do you mean by regression testing? Discuss the different types of regression testing.
2. Define and discuss the following:
   a. Regression testing and how regression test selection is done.
   b. Selective retest and coverage techniques.
   c. Minimization and safe techniques.
3. Write a short paragraph on regression testing.
4. What is regression testing? Discuss the regression test selection problem.
5. What is the importance of regression testing?
6. Explain how risks are prioritized.
7. How can we reduce the number of test cases?
8. See the following priority scheme for test cases:

   Priority 0: Test cases that check basic functionality are run for accepting the build for further testing, and are run when the product undergoes a major change. These test cases deliver a very high project value to development teams and customers.
   Priority 1: Uses the basic and normal setup and these test cases deliver high project value to both development teams and customers.
   Priority 2: These test cases deliver moderate project value. They are executed as part of the testing cycle and selected for regression testing on a need basis.

   Using this scheme of prioritization, prioritize the following test cases as P_0 or P_1 or P_2:

   a. A test case for DBMS software that tests all options of a select query.
   b. A test case for a file system that checks for deallocation of free space.

c. A test case that checks the functionality of a router/bridge.

d. A test case that tests the 0S boot strap process with normal parameters.

e. A test case that checks a login form of a website.

9. Explain in brief the reduction schemes on prioritizing the test cases to reduce required testing effort? As a tester, how do you evaluate and rank potential problems? Suggest some guidelines of your own approach to reduce the number of test cases.

10. Explain how risk matrix can be used to prioritize the test cases. Explain giving an example. Why do we need to prioritize the test cases?

11. **a.** What are prioritization categories and guidelines for test case reduction?

b. What is the need for regression testing? How it is done?

12. Suppose your company is about to roll out an e-commerce application. It is not possible to test the application on all types of browsers on all platforms and operating systems. What steps would you take in the testing environment to reduce the testing process and all possible risks?

13. List and explain prioritization guidelines.

14. What is regression testing and the explain different types of regression testing? Compare various regression testing techniques.

15. **a.** What is the role of risk matrix for the reduction of test cases?

b. How is risk analysis used in testing? Explain the role of the risk analysis table in testing.

CHAPTER 11

TEST MANAGEMENT AND AUTOMATION

Inside this Chapter:

11.0. Automated Testing
11.1. Consideration During Automated Testing
11.2. Static and Dynamic Analysis Tools
11.3. Problems with Manual Testing
11.4. Scope of Automated Testing
11.5. Disadvantages of Automated Testing
11.6. Testing Tools
11.7. Test Automation: "No Silver Bullet"
11.8. Testing and Debugging
11.9. Criteria for Selection of Test Tools
11.10. Design and Architecture for Automation
11.11. Characteristics of Modern Tools
11.12. Case Study on Automated Tools, Namely, Rational Robot, WinRunner, Silk Test, and Load Runner

11.0. AUTOMATED TESTING

Developing software to test the software is called test automation/automated testing. In simple terms, automated testing is automating the manual testing process. It is used to replace or supplement manual testing with a suite of

testing tools. Automated testing tools assist software testers to evaluate the quality of the software by automating the mechanical aspects of the software testing task. Automated testing tools vary in their underlying approach, quality, and ease of use.

Manual testing is used to document tests, produce test guides based on data queries, provide temporary structures to help run tests, and measure the result of the test. Manual testing is considered to be costly and time-consuming. Therefore, automated testing is used to cut down time and cost.

11.1. CONSIDERATION DURING AUTOMATED TESTING

While performing testing with automated tools, the following points should be noted:

1. Clear and reasonable expectations should be established in order to know what can and what cannot be accomplished with automated testing in the organization.
2. There should be a clear understanding of the requirements that should be met in order to achieve successful automated testing. This requires that the technical personnel should use the tools effectively.
3. The organization should have detailed, reusable test cases that contain exact expected results and a stand-alone test environment with a restorable database.
4. Testing tool should be cost-effective. The tool must ensure that test cases developed for manual testing are also useful for automated testing.
5. Select a tool that allows the implementation of automated testing in a way that conforms to the specified long-term testing strategy.

Automated tools like **Mothora** are used to create and execute test cases, measure test case adequacy, determine input-output correctness, locate and remove faults or bugs, and control and document the test.

Similarly, **Bug Trapper** is used to perform white-box testing. This tool traces the path of execution and captures the bug along with the path of execution and the different input values that resulted in the error.

Table 11.1 lists some commonly used automated test tools.

TABLE 11.1 Software Testing Tools and Their Vendors/Manufacturers.

S. No.	Manufacturer	Testing tools
1.	Segue	• Silk Test • Silk Performer • Silk Central
2.	IBM/Rational	• Requirements Pro • Robot • Clear Case
3.	Mercury Interactive	• WinRunner • Load Runner • Test Director
4.	Compuware	• Reconcile • QA Load • QA Run

In any phase of the software development life cycle (SDLC), automated tools are there. Tools like MS-Project are project management tools. Design tools like object modeling technique (OMT) and testing tools like flow graph generator are also available in the market. Regression testing tools are finally used during testing life cycle. These tools are often known as **Computer Aided Software Testing (CAST) Tools**.

Table shows certain other tools used during testing and their field of applications.

S. No.	Tool name	Vendor	Applications
1.	SQA Manager	Rational (IBM)	Test management
2.	Review Pro	Software Development Technologies (SDT)	Review management
3.	Visual Quality	McCabe and his associates	Cyclomatic complexities of source code
4.	Java Scope	Sun Microsystems	Coverage analysis
5.	Bounds Checker	Compuware	Memory testing
6.	CA-Datamacs/II	Computer Associates	Database generators

11.2. STATIC AND DYNAMIC ANALYSIS TOOLS

Because testing is of two types

1. Static testing
2. Dynamic testing

And also that the tools used during these testing are accordingly named as

a. Static testing tools.
b. Dynamic testing tools.

Static testing tools seek to support the static testing process whereas dynamic testing tools support dynamic testing process. Note that static testing is different from dynamic testing. We tabulate the differences between static and dynamic testing before discussing about its tools. They are shown in Table 11.2.

TABLE 11.2 Static Testing V/s Dynamic Testing.

S. No.	Static testing	Dynamic testing
1.	Static testing does not require the actual execution of software.	Dynamic testing involves testing the software by actually executing it.
2.	It is more cost effective.	It is less cost effective.
3.	It may achieve 100% statement coverage in relatively short time.	It achieves less than 50% statement coverage because it finds bugs only in part of codes that are actually executed.
4.	It usually takes shorter time.	It may involve running several test cases, each of which may take longer then compilation.
5.	It may uncover a variety of bugs.	It uncovers a limited type of bugs that are explorable through execution.
6.	It can be done before compilation.	It can take place only after executables are ready.

Software testing tools are frequently used to ensure consistency, thoroughness, and efficiency in testing software products and to fulfill the requirements of planned testing activities. These tools may facilitate unit (module) testing and subsequent integration testing (e.g., drivers and stubs)

as well as commercial software testing tools. Testing tools can be classified into one of the two categories listed below:

(A) **Static test tools:** These tools do not involve actual input and output. Rather, they take a symbolic approach to testing, i.e., they do not test the actual execution of the software. These tools include the following:
   a. **Flow analyzers:** They ensure consistency in data flow from input to output.
   b. **Path tests:** They find unused code and code with contradictions.
   c. **Coverage analyzers:** It ensures that all logic paths are tested.
   d. **Interface analyzers:** It examines the effects of passing variables and data between modules.

(B) **Dynamic test tools**: These tools test the software system with "live" data. Dynamic test tools include the following:
   a. **Test driver:** It inputs data into a module-under-test (MUT).
   b. **Test beds:** It simultaneously displays source code along with the program under execution.
   c. **Emulators:** The response facilities are used to emulate parts of the system not yet developed.
   d. **Mutation analyzers:** The errors are deliberately "fed" into the code in order to test the fault tolerance of the system.

11.3. PROBLEMS WITH MANUAL TESTING

The main problems with manual testing are listed below:

1. **Not reliable:** Manual testing is not reliable as there is no yardstick available to find out whether the actual and expected results have been compared. We just rely on the tester's words.
2. **High risk:** A manual testing process is subject to high risks of oversights and mistakes. People get tired, they may be temporarily inattentive, they may have too many tasks on hand, they may be insufficiently trained, and so on. Therefore, unintentionally mistakes happen in entering data, in setting parameters, in execution, and in comparisons.
3. **Incomplete coverage:** Testing is quite complex when we have a mix of multiple platforms, OS servers, clients, channels, business processes, etc. Testing is non exhaustive. Full manual regression testing is impractical.

4. **Time consuming:** Limited test resources makes manual testing simply too time consuming. As per a study done, 90% of all IT projects are delivered late due to manual testing.
5. **Fact and fiction:** The fiction is that manual testing is done while the fact is only some manual testing is done depending upon the feasibility.

Please note that manual testing is used to document tests, produce test guides based on data queries, provide temporary structures to help run tests, and measure the results of the test. Manual testing is considered to be costly and time-consuming; so we use automated testing to cut down time and cost.

11.4. SCOPE OF AUTOMATED TESTING

Automated testing is the process of automating the manual testing process. It is used to replace or supplement manual testing with a suite of testing tools. Automated testing tools assist software testers to evaluate the quality of the software by automating the mechanical aspects of the software testing task. The benefits of automation include increased software quality, improved time to market, repeatable test procedures, and reduced testing costs. We will now list some more benefits of test automation. They are given below:

1. Automated execution of test cases is faster than manual execution. This saves time. This time can also be utilized to develop additional test cases, thereby improving the coverage of testing.
2. Test automation can free test engineers from mundane tasks and make them focus on more creative tasks.
3. Automated tests can be more reliable. This is because manually running the tests may result in boredom and fatigue, more chances of human error. Automated testing overcomes all of these shortcomings.
4. Automation helps in immediate testing as it need not wait for the availability of test engineers. In fact,

   Automation = Less Person Dependence

5. Test cases for certain types of testing such as reliability testing, stress testing, load, and performance testing cannot be executed without

automation. For example, if we want to study the behavior of a system with millions of users logged in, there is no way one can perform these tests without using automated tools.

6. Manual testing requires the presence of test engineers but automated tests can be run a round the clock, in a 24 × 7 environment. So, automated testing provides a round the clock coverage.
7. Tests, once automated, take comparatively far less resources to execute. A manual test suite requiring 10 persons to execute it over 31 days, i.e., 31 × 10 = 310 man-days, may take just 10 man-days for execution, if automated. Thus, a ratio of 1 : 31 is achieved.
8. Automation produces a repository of different tests which helps us to train test engineers to increase their knowledge.
9. Automation does not end with developing programs for the test cases. Automation includes many other activities like selecting the right product build, generating the right test data, analyzing results, and so on.

Automation should have scripts that produce test data to maximize coverage of permutations and combinations of input and expected output for result comparison. They are called test data generators.

It is important for automation to relinquish the control back to test engineers in situations where a further set of actions to be taken are not known.

As the objective of testing is to catch defects early, the automated tests can be given to developers so that they can execute them as part of unit testing.

11.5. DISADVANTAGES OF AUTOMATED TESTING

Despite of many benefits, the pace of test-automation is slow. Some of its disadvantages are given below:

1. An average automated test suite development is normally 3–5 times the cost of a complete manual test cycle.
2. Automation is too cumbersome. Who would automate? Who would train? Who would maintain? This complicates the matter.
3. In many organizations, test automation is not even a discussion.
4. There are some organizations where there is practically no awareness or only some awareness on test automation.

5. Automation is not an item of higher priority for managements. It does not make much difference to many organizations.
6. Automation would require additional trained staff. There is no staff for the purpose.

Automation actually allows testing professionals to concentrate on their real profession of creating tests and test-cases rather than doing the mechanical job of test execution.

11.6. TESTING TOOLS

The skills required depend on what generation of automation the company is in.

1. Capture/playback and test harness tools (first generation).
2. Data driven tools (second generation).
3. Action driven (third generation).

We discuss these three generations on which skills depends, one by one.

(I) Capture/Playback and Test Harness Tools

One of the most boring and time-consuming activities during testing life cycle is to rerun manual tests a number of times. Here, capture/playback tools are of great help to the testers. These tools do this by recording and replaying the test input scripts. As a result, tests can be replayed without attendent for long hours especially during regression testing. Also these recorded test scripts can be edited as needed, i.e., whenever changes are made to the software. These tools can even capture human operations, e.g., mouse activity, keystrokes, etc.

A capture/playback tool can be either intrusive or non intrusive. Intrusive capture/playback tools are also called native tools as they along with software-under-test (SUT) reside on the same machine.

Non intrusive capture/playback tools on the other hand reside on the separate machine and is connected to the machine containing software to be tested using special hardware. One advantage of these tools is to capture errors which users frequently make and which developers cannot reproduce.

Test harness tools are the category of capture/playback tools used for capturing and replaying a sequence of tests. These tools enable running a large volume of tests unattended and help in generating reports by using comparators. These tools are very important at CMM level 2 and above.

(II) Data-driven Tools

This method helps in developing test scripts that generates the set of input conditions and corresponding expected output. The approach takes as much time and effort as the product. However, changes to the application does not require the automated test cases to be changed as long as the input conditions and expected output are still valid. This generation of automation focuses on input and output conditions using the black-box testing approach.

(III) Action-driven Tools

This technique enables a layman to create automated tests. There are no input and expected output conditions required for running the tests. All actions that appear on the application are automatically tested, based on a generic set of controls defined for automation.

From the above approaches/generations of automation, it is clear that different levels of skills are needed based on the generation of automation selected.

11.7. TEST AUTOMATION: "NO SILVER BULLET"

Test automation is a partial solution and not a complete solution. One does not go in for automation because it is easy. It is a painful and resource-consuming exercise but once it is done, it has numerous benefits. For example, developing a software to automate inventory management may be a time-consuming, painful, and costly resource-intensive exercise but once done, inventory-management becomes relatively a breeze.

Of course, test-automation is "**no silver bullet**". It is only one of the many factors that determine software quality.

11.8. TESTING AND DEBUGGING

Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. After testing, we begin an investigation to locate the error, i.e., to find out which module or interface is causing it. Then that section of the code is to be studied to determine the cause of the problem. This process is called debugging.

Hence, debugging is an activity of locating and correcting errors. Debugging is not testing but it always occurs as a consequence of testing. The debugging process begins with the execution of a test case. The debugging process will always have one of two outcomes.

1. The cause will be found, corrected, and removed, or
2. The cause will not be found.

During debugging, we encounter errors that range from mildly annoying to catastrophic.

Some guidelines that are followed while performing debugging are:

1. Debugging is the process of solving a problem. Hence, individuals involved in debugging should understand all of the causes of an error before starting with debugging.
2. No experimentation should be done while performing debugging. The experimental changes often increase the problem by adding new errors in it instead of removing errors.
3. When there is an error in one segment of a program, there is a high possibility of the presence of another error in that program. So, the rest of the program should be properly examined.
4. It is necessary to confirm that the new code added in a program to fix errors is correct. And to ensure this, regression testing should be performed.

The Debugging Process

During debugging, errors are encountered that range from less damaging (like input of an incorrect function) to catastrophic (like system failure, which leads to economic or physical damage). The various levels of errors and their damaging effects are shown in Figure 11.1.

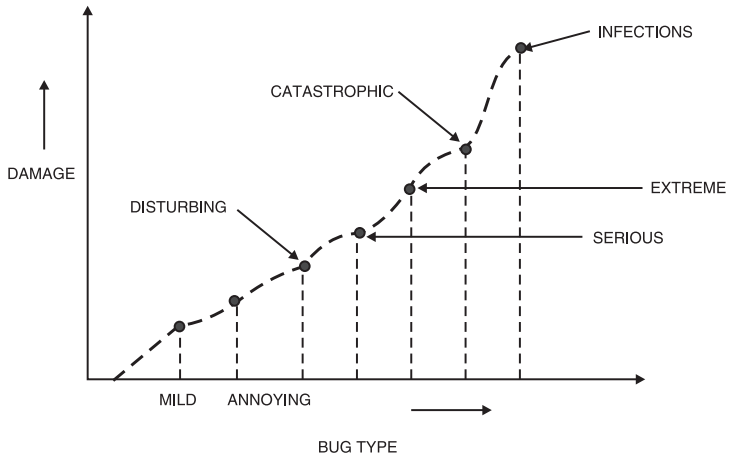


FIGURE 11.1 Level of Error and its Effect.

Note that in this graph, as the number of errors increase, the amount of effort to find their causes also increases.

Once errors are identified in a software system to debug the problem a number of steps are followed:

Step 1. Identify the errors.
Step 2. Design the error report.
Step 3. Analyze the errors.
Step 4. Debugging tools are used.
Step 5. Fix the errors.
Step 6. Retest the software.

After the corrections are made, the software is retested using regression tests so that no new errors are introduced during the debugging process.

Note that debugging is an integral component of the software testing process. Debugging occurs as a consequence of successful testing and revealing the bugs from the software-under-test (SUT). When a test case uncovers an error, debugging is the process that results in the removal of the bugs. Also note that debugging is not testing, but it always occurs as a consequence of testing. The debugging process begins with the execution of a test case. This is shown in Figure 11.2.

The debugging process attempts to match the symptom with the cause thereby leading to error correction. The purpose of debugging is to locate and fix the offending code responsible for a symptom violating a known specification.

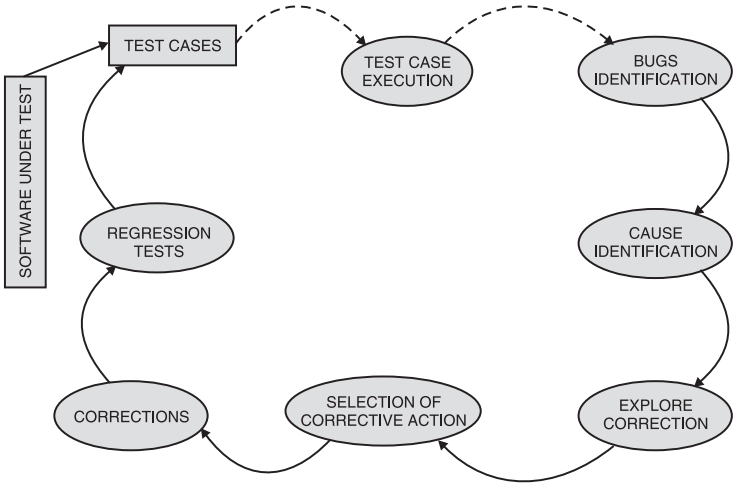


FIGURE 11.2 Overall Debugging Life Cycle.

Testing uses unit-, integration-, and system-level approaches for doing fault detection. On the other hand, debugging checks the correctness and the performance of software to do fault detection. Table shows the differences between these two techniques.

S. No.	Software testing	Debugging
1.	It is the process of executing the program with the intent of finding faults.	It is an activity of locating and correcting errors.
2.	It is a phase of the software development life cycle (SDLC).	It occurs as a consequence of testing.
3.	Once code is written, testing commences.	The debugging process begins with the execution of a test case.
4.	It is further comprised of validation and verification of software.	It attempts to match symptom with cause, thereby leading to error correction.
5.	It uses unit-, integration-, and system-level testing.	It checks the correctness and performance of the software.

Debugging Approaches

Several approaches have been discussed in literature for debugging software-under-test (SUT). Some of them are discussed below.

1. **Brute force method:** This method is most common and least efficient for isolating the cause of a software error. We apply this method when all else faild. In this method, a printout of all registers and relevant memory locations is obtained and studied. All dumps should be well documented and retained for possible use on subsequent problems.
2. **Back tracking method:** It is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward until the site of the cause is found. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

3. **Cause elimination:** The third approach to debugging, cause elimination, is manifested by induction or deduction and introduces the concept of **binary partitioning**. This approach is also called **induction** and **deduction**. Data related to the error occurrence are organized to isolate potential causes. A "cause hypothesis" is devised and the data are used to prove or disprove the hypothesis. Alternatively, a list of all possible causes is developed and tests are conducted to eliminated each. If initial tests indicate that a particular cause hypothesis shows promise, the data are refined in an attempt to isolate the bug.

Tools for debugging: Each of the above debugging approaches can be supplemented with debugging tools. For debugging we can apply a wide variety of debugging tools such as debugging compilers, dynamic debugging aids, automatic test-case generators, memory dumps, and cross reference maps. The following are the main debugging tools:

1. **Turbo Debugger for Windows:** The first debugger that comes to mind when you think of a tool especially suited to debug your Delphi code is Borland's own Turbo Debugger for Windows.
2. **Heap Trace:** Heap Trace is a shareware heap debugger for Delphi 1·X and 2·X applications that enables debugging of heap memory use. It helps you to find memory leaks, dangling pointers, and memory overruns in your programs. It also provides optional logging of all memory allocations, de-allocations, and errors. Heap trace is optimized for speed, so there is only a small impact on performance even on larger applications. Heap trace is configurable and you can also change the format and destination of the logging output, choose what to trace, etc. You can trace each allocation and de-allocation and find out where each block of memory is being created and freed. Heap trace can be used to simulate out of memory condition to test your program in stress conditions.
3. **MemMonD:** MemMonD is another shareware memory monitor for Delphi 1·X applications. It is a stand alone utility for monitoring memory and stack use. You don't need to change your source code but only need to compile with the debug information included, MemMonD detects memory leaks and deallocations with wrong size.
4. **Re-Act:** Re-Act is not really a debugger for Delphi, but more a Delphi component tester. However, this tool is a real pleasure to use. So it could not resist including it in this list of debugging tools.

Currently, reach is for Delphi 2 only, but a 16 bits version is in the works. Re-Act version 2.0 is a really nice component, view and change properties at run time with the built-in component inspector, monitor events in real time, set breakpoints and log events visually and dynamically. You can find elusive bugs, evaluate third-party components, and learn how poorly documented components really work. If you build or purchase components you need this tool. It's totally integrated with Delphi 2.0 and the CDK 2.0.

11.9. CRITERIA FOR THE SELECTION OF TEST TOOLS

The main criteria for the selection of test tools are given below:

1. Meeting requirements
2. Technology expectations
3. Training/skills
4. Management aspects

We now discuss these criteria one by one.

1. Meeting Requirements

a. There are many tools available in the market today but rarely do they meet all of the requirements of a given product or a given organization. Evaluating different tools for different requirements involves a lot of effort, money, and time. A huge delay is involved in selecting and implanting test tools.

b. Test tools may not provide backward or forward compatibility with the product-under-test (PUT).

c. Test tools may not go through the same amount of evaluation for new requirements. Some tools had a Y2K-problem also.

d. A number of test tools cannot distinguish between a product failure and a test failure. This increases analysis time and manual testing. The test tools may not provide the required amount of trouble-shooting/debug/error messages to help in analysis. For example, in case of GUI testing, the test tools may determine the results based on messages and screen coordinates at run-time. So, if the screen elements of the product are changed, it requires the test suite to be

changed. The test tool must have some intelligence to proactively find out the changes that happened in the product and accordingly analyze the results.

2. Technology Expectations

a. In general, test tools may not allow test developers to extend/modify the functionality of the framework. So, it involves going back to the tool vendor with additional cost and effort. Very few tools available in the market provide source code for extending functionality or fixing some problems. Extensibility and customization are important expectations of a test tool.

b. A good number of test tools require their libraries to be linked with product binaries. When these libraries are linked with the source code of the product, it is called the "instrumented code". This causes portions of testing be repeated after those libraries are removed, as the results of certain types of testing will be different and better when those libraries are removed. For example, the instrumented code has a major impact on the performance testing because the test tools introduce an additional code and there could be a delay in executing the additional code.

c. Finally, test tools are not 100% cross-platform. They are supported only on some OS platforms and the scripts generated from these tools may not be compatible on other platforms. Moreover, many of the test tools are capable of testing only the product, not the impact of the product/test tool to the system or network. When there is an impact analysis of the product on the network or system, the first suspect is the test tool and it is uninstalled when such analysis starts.

3. Training Skills

Test tools require plenty of training, but very few vendors provide the training to the required level. Organization-level training is needed to deploy the test tools, as the users of the test suite are not only the test team but also the development team and other areas like SCM (software configuration management). Test tools expect the users to learn new languages/scripts and may not use standard languages/scripts. This increases the skill requirements for automation and increases the need for a learning curve inside the organization.

4. Management Aspects

A test tool increases the system requirement and requires the hardware and software to be upgraded. This increases the cost of the already-expensive test tool. When selecting the test tool, it is important to note the system requirements and the cost involved in upgrading the software and hardware needs to be included with the cost of the tool. Migrating from one test tool to another may be difficult and requires a lot of effort. Not only is this difficult as the test suite that is written cannot be used with other test tools but also because of the cost involved. As the tools are expensive and unless the management feels that the returns on investment (ROI) are justified, changing tools are generally not permitted.

Deploying a test tool requires as much effort as deploying a product in a company. However, due to project pressures, test tools effort at deploying gets diluted, not spent. Thus, later it becomes one of the reasons for delay or for automation not meeting expectations. The support available on the tool is another important point to be considered while selecting and deploying the test tool.

11.10. DESIGN AND ARCHITECTURE FOR AUTOMATION

There are seven steps to select and deploy a test tool in an organization. These steps are:

Step 1. Identify your test suite requirements among the generic requirements discussed. Add other requirements, if any.

Step 2. Make sure experiences discussed in previous sections are taken care of.

Step 3. Collect the experiences of other organizations which used similar test tools.

Step 4. Keep a checklist of questions to be asked to the vendors on cost/effort/support.

Step 5. Identify a list of tools that meet the above requirements and give priority for the tool which is available with the source code.

Step 6. Evaluate and shortlist one set of tools and train all of the test developers on the tool.

Step 7. Deploy the tool across test teams after training all of the potential users of the tool.

11.11. CHARACTERISTICS OF MODERN TESTING TOOLS

The modern testing tools available today have some salient features that are discussed below:

i. It should use one or more testing strategy for performing testing on host and target platforms.
ii. It should support GUI-based test preparation.
iii. It should provide complete code coverage and create test documentation in various formats like .doc, .html, .rtf, etc.
iv. These tools should be able to adopt the underlying hardware.
v. It should be easy to use.
vi. It should provide a clear and correct report on test case, test case status (PASS/FAIL), etc.

11.12. CASE STUDY ON AUTOMATED TOOLS, NAMELY, RATIONAL ROBOT, WINRUNNER, SILK TEST, AND LOAD RUNNER

Rational Robot: Its main features are:

1. "Rational" is for the name of the company, "Rational software cooperation (IBM)". It is called as "Robot" because like a robot it operates the application and does data-entry on screens.
2. Rational Robot is highly useful and effective in automating screen-based testing.
3. Operations on Windows/Web GUI, entry of data on screens, and other screen operations can be automatically captured by Robot into a script.
4. The recorded scripts can be played back.
5. Once the operations and data-entry are recorded, they can be automatically played back anytime, dispensing with human intervention and manual data-entry.
6. The scripts can be edited. Scripts are in SQA Basic. SQA Basic is a powerful language providing many options. For example, instead of recording/capturing data-entry, data can be read-in from a repository/database and entered into target screens. One way would

be to capture screen operations and entry of just one record into a script through Robot and then to suitably edit the script to provide for data-entry by reading-in values from a repository/database.

7. Rational Robot and SQA Basic provide an excellent, user-friendly development environment.
8. GUI applications such as VS.NET, VB6.0, Oracle Forms, HTML, and JAVA can be tested through Robot.
9. Robot uses object-oriented recording technology (OORT). It identifies objects by their internal object names and lets you generate scripts automatically by simply running and using the application-under-test (AUT).
10. There are many similar tools like Mercury Interactive's WinRunner, Seague's Silk Test, etc.

WinRunner: Its main features are:

1. WinRunner is a recording and playback tool of Mercury Interactive.
2. WinRunner captures, verifies, and replays user GUI interactions automatically.
3. Its DataDriver facilitates the process of preparing test data and scripts. It examines and compares expected and actual results using multiple verfications for text, GUI, bitmaps, URLs, and databases.
4. WinRunner enables the same test to be used to validate applications in Internet Explorer, Netscape, etc.
5. It integrates with other testing solutions from Mercury like Load Runner for load testing and Test Director for global test management.
6. WinRunner supports Windows platform, Web browsers, and various ERP/CRM applications.
7. Its recovery manager and exception handling mechanism automatically troubleshoots unexpected events, errors, and application crashes to ensure smooth test completion.
8. WinRunner has an interactive reporting tool also. It generates reports so that the results can be interpreted.
9. Testers need not modify multiple tests when the application's screens are modified over time. Instead, they can apply changes to the GUI map, central repository of test-related information and WinRunner will automatically propagate changes to relevant scripts.

10. WinRunner supports VB6, ActiveX, Oracle, C/C++, Java, JavaScript, JDK, Delphi, Centura, and Windows platform and Terminal Emulators like VT 100.

Segue's Silk Test: Its main features are:

1. Silk Test is a recording and playback tool of **Segue Software Inc**.
2. It provides user simulation capabilities driving the application exactly how an end-user would.
3. Its recovery system allows tests to be run unattended. If unexpected errors such as application crashes occur, the errors are logged and the application is restored to its original base state so that subsequent tests can be run.
4. Its 4 Test scripting language is an object-based fourth-generation language (4GL).
5. Silk Test supports Internet Explorer, Netscape, Windows Platform, VB 6.0, Power Builder, etc.

LoadRunner: Its main features are:

1. LoadRunner is a load/performance testing tool of Mercury Interactive.
2. Using minimal hardware resources, it emulates hundreds to thousands of concurrent users with real-life user loads, monitors performance of servers and network, and helps identify performance bottle necks and scalability issues before they impact end-users.
3. It measures the response times of key business processes and transactions, captures and displays performance data from every tier-server and component.
4. Its analysis module allows drill down to specific sources of bottlenecks and generate reports.
5. It allows organizations to minimize testing cycles, optimize performance, and reduce risks.
6. Its WAN emulation capability allows testing for network errors, bandwidth, and latency.
7. It supports Citrix's ICA for testing applications deployed with Citrix Metaframe.
8. LoadRunner supports a wide variety of platforms.

SUMMARY

Testing is an expensive and laborious phase of the software process. As a result, testing tools were among the first software tools to be developed. These tools now offer a range of facilities and their use and significantly reduces the cost of the testing process. Different testing tools may be integrated into the testing workbench.

These tools are:

1. **Test manager:** It manages the running of program tests. It keeps track of test data, expected results, and program facilities tested.
2. **Test data generator:** It generates test data for the program to be tested. This may be accomplished by selecting data from a database.
3. **Oracle:** It generates predictions of expected test results.
4. **File comparator:** It compares the results of program tests with previous test results and reports differences between them.
5. **Report generator:** It provides report definition and generation facilities for test results.
6. **Dynamic analyzer:** It adds code to a program to count the number of times each statement has been executed. After the tests have been run, an execution profile is generated showing how often each program statement has been executed.

Testing workbenches invariably have to be adapted to suit the test plan of each system. A significant amount of effort and time is usually needed to create a comprehensive testing workbench. Most testing work benches are open systems because testing needs are organization specific.

Automation, however, makes life easier for testers but is not a "silver bullet". It makes life easier for testers for better reproduction of test results, coverage, and reduction in effort. With automation we can produce better and more effective metrics that can help in understanding the state of health of a product in a quantifiable way, thus taking us to the next change.

MULTIPLE CHOICE QUESTIONS

1. Automated testing is
   - **a.** To automate the manual testing process
   - **b.** To assure quality of software
   - **c.** To increase costs
   - **d.** None of the above.
2. Automated tools can be used during the
   - **a.** SRS phase
   - **b.** Design phase
   - **c.** Coding phase
   - **d.** Testing and maintenance phases
   - **e.** All of the above.
3. Testing a system with live data is done using
   - **a.** Static test tools
   - **b.** Dynamic test tools
   - **c.** Both (a) and (b)
   - **d.** None of the above.
4. Tools used to record and replay the test input scripts are known as
   - **a.** Test Harness tools
   - **b.** Data driven tools
   - **c.** Action driven tools
   - **d.** None of the above.
5. Thread testing is used for testing
   - **a.** Real-time systems
   - **b.** Object-oriented systems
   - **c.** Event-driven systems
   - **d.** All of the above.
6. Debugging is
   - **a.** An activity of locating and correcting errors
   - **b.** A process of testing
   - **c.** A process of regression testing
   - **d.** None of the above.

7. Which of the following is a debugging approach?
   - **a.** Brute-Force method
   - **b.** Inheritance
   - **c.** Data flow diagrams
   - **d.** None of the above.
8. Which of these is a debugging tool?
   - **a.** Windows
   - **b.** Heap-trace
   - **c.** SCM
   - **d.** None of the above.
9. CASE stands for
   - **a.** Computer aided software engineering
   - **b.** Case aided system engineering
   - **c.** Computer aided system engineering
   - **d.** None of the above.
10. Which of these can be used for testing as a tool?
   - **a.** Rational Robot
   - **b.** Waterfall model
   - **c.** MS-WORD
   - **d.** None of the above.

ANSWERS

1. a.	**2.** e.	**3.** b.	**4.** a.
5. d.	**6.** a.	**7.** a.	**8.** b.
9. a.	**10.** a.		

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

Q. 1. List one example of a test objective and two examples of a test requirement.

Ans. Test objective example: Creating a new order with the system.
Test requirement example:
- **a.** Verifying that the insertion done appears in the status bar of the application's window.
- **b.** Verifying that an order number is displayed in the order number box of the AUT (application-under-test).

Q. 2. What is the importance of learning AUT before creating automated test scripts?

Ans. You can record a basic test with the correct user actions in a short amount of time.

Q. 3. How can you estimate the total number of testers required for your project?

Ans. We have a simple formula:

$$\text{Total no. of testers} = \frac{\text{Total testing hours}}{\text{Total no. of weeks} \times \text{Hours / week}}$$

$$\text{If total testing hours} = 100$$

$$\text{Total no. of weeks} = 4$$

$$\text{and hours/week} = 25$$

$$\text{Then, No. of testers required} = \frac{100}{4 \times 25} = \frac{25}{25} = 1 \text{ tester.}$$

Q. 4. What should you do when you evaluate test automation?

Ans. When evaluating test automation, we should:

1. Look for the tests that take the most time.
2. Look for tests that could otherwise not be run like server tests.
3. Consider acceptance tests.
4. Look for stable application components.

Q. 5. Which is costlier—an automated test suite or a manual test suite?

Ans. An average automated test suite development is normally 3–5 times the cost of a complete manual test cycle.

REVIEW QUESTIONS

1. Answer the following:
   - **a.** What is debugging?
   - **b.** What are different approaches to debugging?
   - **c.** Why is exhaustive testing not possible?
2. Explain the following:
   - **a.** Modern testing tools.

3. a. Differentiate between static and dynamic testing tools with examples in detail.
   b. Will exhaustive testing guarantee that the program is 100% correct?
4. Compare testing with debugging.
5. Differentiate between static and dynamic testing.
6. Write a short paragraph on:
   a. Testing tools.
7. Compare testing with debugging.
8. Explain back-tracking method for debugging.
9. Differentiate between static and dynamic testing tools.
10. What are the benefits of automated testing tools over conventional testing tools?
11. Discuss various debugging approaches with some examples.
12. a. What is debugging? Describe various debugging approaches.
    b. Differentiate between static testing tools and dynamic testing tools.
13. Briefly discuss dynamic testing tools.
14. Write short notes on any two:
    a. Static testing tools.
    b. Dynamic testing tools.
    c. Characteristics of modern tools.
15. a. Discuss in detail automated testing and tools. What are the advantages and disadvantages?
    b. Explain in brief modern tools in the context of software development and their advantages and disadvantages?
16. List and explain the characteristics of modern testing tools.
17. Explain modern testing tools.

18. Write short paragraphs on – static and dynamic testing tools.

19. Differentiate between static and dynamic testing tools.

20. **a.** Explain the characteristics of Modern Testing tools.

b. What are the different categories of software testing tools? Explain them in details.

CHAPTER 12

A Case Study on Testing of E-Learning Management Systems

Abstract

Software testing is the process of executing a program or system with the intent of finding errors. It involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. To deliver successful software products, quality has to be ensured in each and every phase of a development process. Whatever the organizational structure may be, the most important point is that the output of each phase should be of very high quality. The SQA team is responsible to ensure that all the development team should follow the quality-oriented process. Any modifications to the system should be thoroughly tested to ensure that no new problems are introduced and that the operational performance is not degraded due to the changes. The goal of testing is to determine and ensure that the system functions properly beyond the expected maximum workload. Additionally, testing evaluates the performance characteristics like response times, transaction rates, and other time sensitive issues.

Chapter One
Introduction

NIIT Technologies is a global IT and business process management services provider with a footprint that spans 14 countries across the world. It has been working with global corporations in the USA, Europe, Japan, Asia Pacific, and India for over two decades. NIIT Technologies provides independent validation and verification services for your high-performance applications. Their testing services help organizations leverage their experience in testing to significantly reduce or eliminate functional, performance, quality, and reliability issues. NIIT Technologies helps enterprises and organizations make their software development activities more successful and finish projects in time and on budget by providing systematic software quality assurance.

The government of India Tax Return Preparers scheme to train unemployed and partially employed persons to assist small and medium taxpayers in preparing their returns of income has now entered its second phase. During its launch year, on a pilot basis, close to 5,000 TRPs at 100 centers in around 80 cities across the country were trained. 3737 TRPs were certified by the Income Tax Department to act as Tax Return Preparers who assisted various people in filing their IT returns. The government has now decided to increase their area of operations by including training on TDS returns and service tax returns to these TRPs. The quality assurance and testing team of NIIT who constantly indulges in testing and maintaining the product quality have to test such online learning content management websites such as *www.trpscheme.com* in the following manner:

- Functional and regression testing
- System testing: Load/stress testing, compatibility testing
- Full life cycle testing

Chapter Two
Software Requirement Specifications

Inside this Chapter:

2.1. Introduction
2.2. Overall Descriptions
2.3. Specific Requirements
2.4. Change Management Process
2.5. Document Approval
2.6. Supporting Information

2.1. INTRODUCTION

This document aims at defining the overall software requirements for "testing of an online learning management system (*www.trpscheme.com*)." Efforts have been made to define the requirements exhaustively and accurately.

2.1.1. Purpose

This document describes the functions and capabilities that will be provided by the website, *www.trpscheme.com*. Its purpose is that the resource center will be responsible for the day-to-day administration of the scheme. The functions of the resource center will include to specify the curriculum and all other matters relating to the training of the Tax Return Preparers and maintain the particulars relating to the Tax Return Preparers. Also, any other function that is assigned to it by the Board for the purposes of implementation of the scheme.

2.1.2. Scope

The testing of the resource center section for service tax is done manually mainly using functional and regression testing. Other forms of testing may also be used such as integration testing, load testing, installation testing, etc.

2.1.3. Definitions, Acronyms, and Abbreviations

Definitions:

1. **Test plan:** We write test plans for two very different purposes. Sometimes the test plan is a product; sometimes it's a tool. In software testing, a test plan gives detailed testing information regarding an upcoming testing effort.
2. **Test case:** A set of conditions or variables under which a tester will determine if a requirement upon an application is partially or fully satisfied. It may take many test cases to determine that a requirement is fully satisfied.
3. **Manual testing:** The most popular method of software application testing. It can be improved by using requirements traceability, test cases, test plan, SQA and debugging, and testing techniques and checklist.
4. **Test report:** A management planning document that shows: Test Item Transmittal Report, Test Log, Test Incident Report, and Test Summary Report.
5. **Staging server:** A staging server is a web server used to test the various components of, or changes to, a website before propagating them to a production server.

Acronym and Abbreviation:

1. TRP–Tax Return Preparer
2. STRP–service tax return preparers scheme
3. QA–quality assurance
4. HTML–hyper text mark-up language
5. STC–service tax code
6. EAR File–enterprise archive file
7. VSS–visual source safe
8. STD–service tax department

2.1.4. References Books

- Chopra, R. 2017. *Software testing and quality assurance: A practical approach*. S.K. Kataria & Sons, New Delhi.

- Aggarwal. K. K. 2005. *Software engineering*. New Age International, New Delhi.

Sites

- http://en.wikipedia.org/Software_testing

2.1.5. OVERVIEW

The rest of the SRS document describes the various system requirements, interfaces, features, and functionalities in detail.

2.2. OVERALL DESCRIPTIONS

2.2.1. PRODUCT PERSPECTIVE

The application will be self-contained.

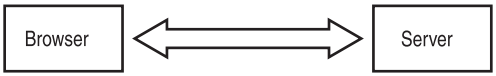


FIGURE 2.1

2.2.1.1. SYSTEM INTERFACES

None.

2.2.1.2. USER INTERFACES

The application will have a user-friendly and menu-based interface. The login page will entertain both user and admin. The following forms and pages will be included in the menu:

- Login screen
- Homepage
- Return filed report
- Return filed form
- STRP wise report
- Service wise report
- Zone commissionerate wise report
- STRP summary report

2.2.1.3. Hardware interfaces

1. **Processor:** Intel Pentium (4) Processor
2. **Ram:** 512 MB and above
3. **Storage Space:** 5 GB and above
4. A LAN card for the Internet

2.2.1.4. Software interfaces

1. **Language:** Java, XML
2. **Software:** Bugzilla, Putty, Toad
3. **Database:** Oracle
4. **Platform:** Windows 2000 (Server) / Linux

2.2.1.5. Communication interfaces

The application should support the following communication protocols:

1. **Http**
2. **Proxy server:** In computer networks, a proxy server is a server (a computer system or an application program) that acts as a go-between for requests from clients seeking resources from other servers.

2.2.1.6. Memory constraints

At least 512 MB RAM and 2 GB hard disk will be required.

2.2.1.7. Site adaptation requirements

The terminals at the client site will have to support the hardware and software interfaces specified in the above sections.

2.2.2. Product Functions

According to the customer use and needs the website function shall include:

i. To specify, with prior approval of the Board,

   a. The number of persons to be enrolled during a financial year for training to act as Tax Return Preparers;

b. The number of centers for training and their location where training is to be imparted during a financial year;
c. The number of persons to be trained at each center for training during a financial year;

ii. To specify the curriculum and all other matters relating to the training of Tax Return Preparers;

iii. Maintain the particulars relating to the Tax Return Preparers;

iv. Any other function which is assigned to it by the Board for the purposes of implementation of the scheme.

2.2.3. User Characteristics

- **Education level:** The user be able to understand one of the languages of the browser (English, Hindi, Telugu). The user must also have a basic knowledge of tax return and payments rules and regulations.
- **Technical expertise:** The user should be comfortable using general-purpose applications on a computer.

2.2.4. Constraints

- Monitor sizes and ratios and color or black-and-white monitors render it virtually impossible to design pages that look good on all device types.
- Font sizes and colors need to be changeable to fit the requirements of sight-impaired viewers.

2.2.5. Assumptions and Dependencies

- Some pages display wrong with some browsers.
- Some web master along the way programmed in some browser-specific codes.

2.2.6. Apportioning of Requirements

None.

2.3. SPECIFIC REQUIREMENTS

This section contains the software requirements to a level of detail sufficient to enable designers to design the system and the testers to test the system.

2.3.1. USER INTERFACES AND VALIDATIONS

The following interfaces will be provided:

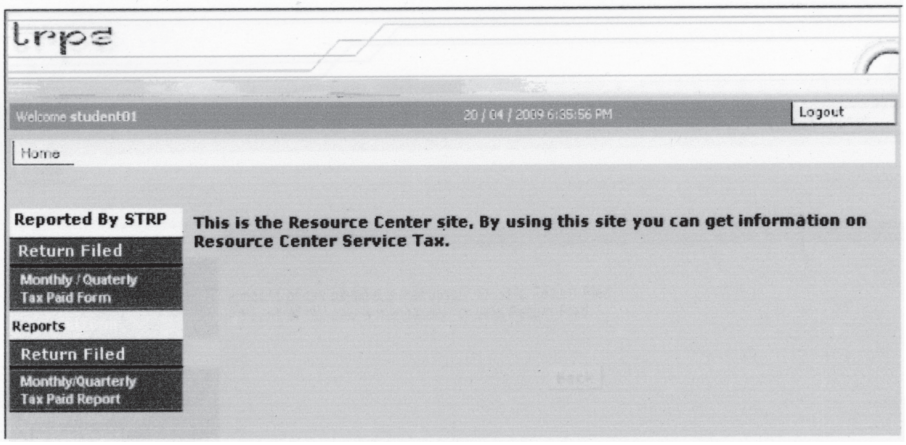
- **Login screen**

Visit *www.trpscheme.com*. The following page appears:

When the STRPs click on the login button of resource center service tax on *TRPscheme.com*, the following page will be displayed.

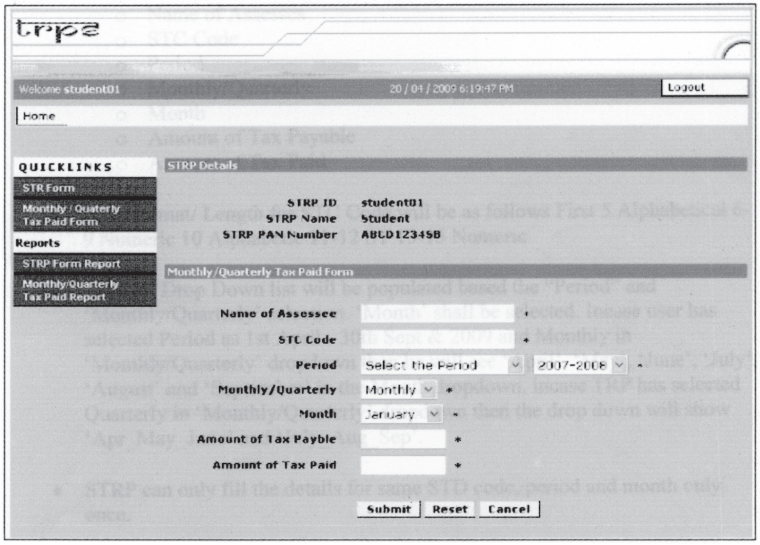
- **Homepage**

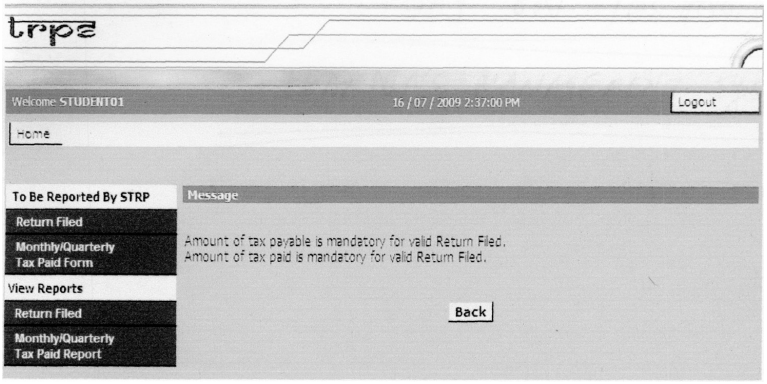
When the STRP logs in by user id and password, the homepage is displayed. The homepage has "Reported by STRP" menu on the left under which the user will see two links, "Return Filed" and "Monthly/Quarterly Tax Paid Form." The user will also see two report links, "Return Filed" and "Monthly/Quarterly Tax Paid Report" under the "Reports" menu. A message "This is the Resource Center site. By using this site you can get information on Resource Center Service Tax" also appears on the homepage.



- **Monthly/Quarterly tax paid form**

This form is used by the STRPs to fill out the Monthly/Quarterly Tax Paid by the Assesses in order to capture the information so that the total tax paid can be assessed. On this page STRP Details will be displayed and the fields that need to be filled in for the completion of the Monthly/Quarterly Tax Paid form. Filling out the "Monthly/Quarterly Tax Paid Form" is mandatory before filling out the Return Filed form. This will not be applied when STRP is filling return for the STC code first time.





Form validation: This form will require validation of the data such as all of the mandatory fields cannot be left blank and "STC Code" must be filled in otherwise the form will not be submitted. Fields such as "Amount of Tax Payable," "Amount of Tax Paid," and "Interest Paid" will only be numeric.

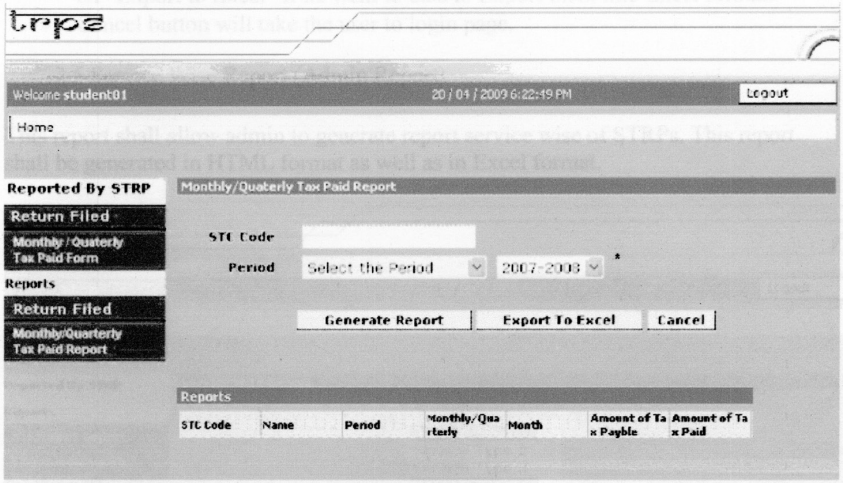
- To complete a form, the user must fill out the following fields. All of the fields are mandatory in this form.
  - Name of Assesses
  - STC Code
  - Period
  - Monthly/Quarterly
  - Month
  - Amount of Tax Payable
  - Amount of Tax Paid
- Field format/length for STC Code will be as follows: [First 5 alphabetical] [6-9 numeric] [10 alphabetical] [11-12 ST] [13-15 numeric]
- "Month" drop-down list will be populated based on the "Period" and "Monthly/Quarterly" selection. "Month" will be selected. If the user has selected "Period" as April 1 though Sept 30 and 2009 and "Monthly" in "Monthly/Quarterly" drop down then he or she will see April, May, June, July, August, and September in the "Month" drop down. If the TRP has selected "Quarterly" in "Monthly/Quarterly" drop down then the drop down will show Apr_May_June and July_Aug_Sep.
- The STRP can only fill in the details for the same STD code, period, and month only once.
- Report to view Monthly\Quarterly form data.

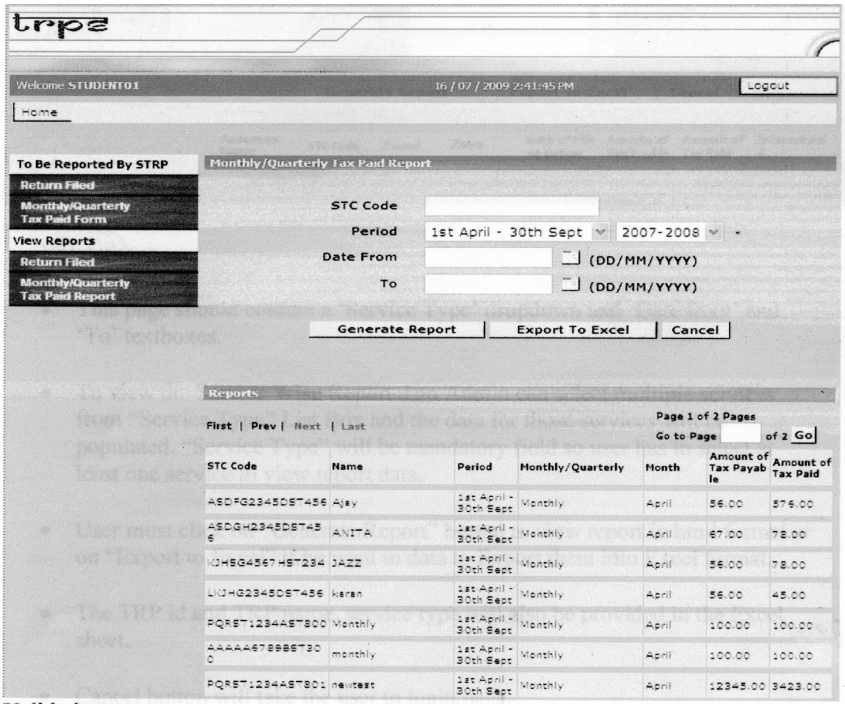
This report will allow STRPs to view Monthly\Quarterly Tax Paid form data and will be able generate a report of the data. STRPs will generate reports in HTML format and also be to able to export them into Excel format.

- To view the report data the STRP is required to provide the "Period" in the given fields that are the mandatory fields.
- The STRP can also use the other field STC code to generate.
- The user must click on the "Generate Report" button to view the report in HTML format or on "Export to Excel" if he or she wants to export the data into Excel format.

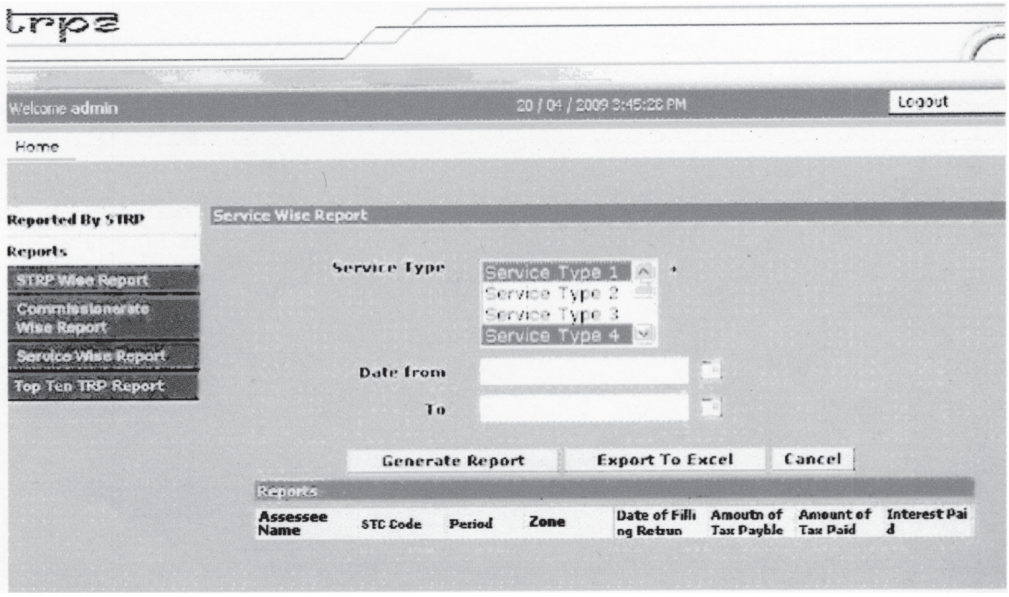
- The "Cancel" button will take the user to the login page.

2.5 Service Wise Report (Admin Report)





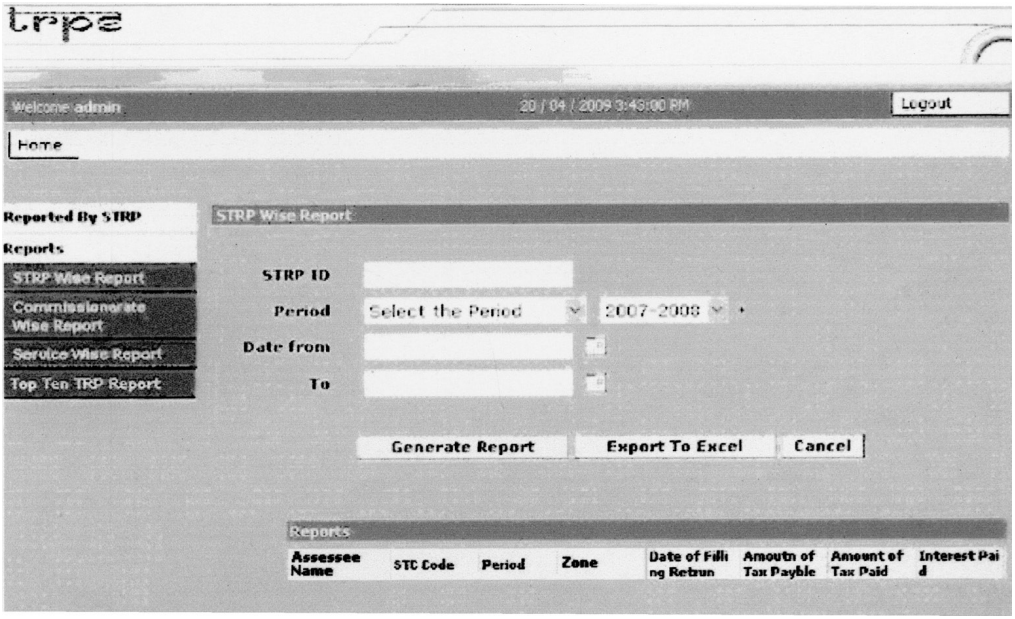
This report will allow the admin to generate a Report Service Wise of STRPs. This report will be generated in HTML format as well as in Excel format.



Validations:

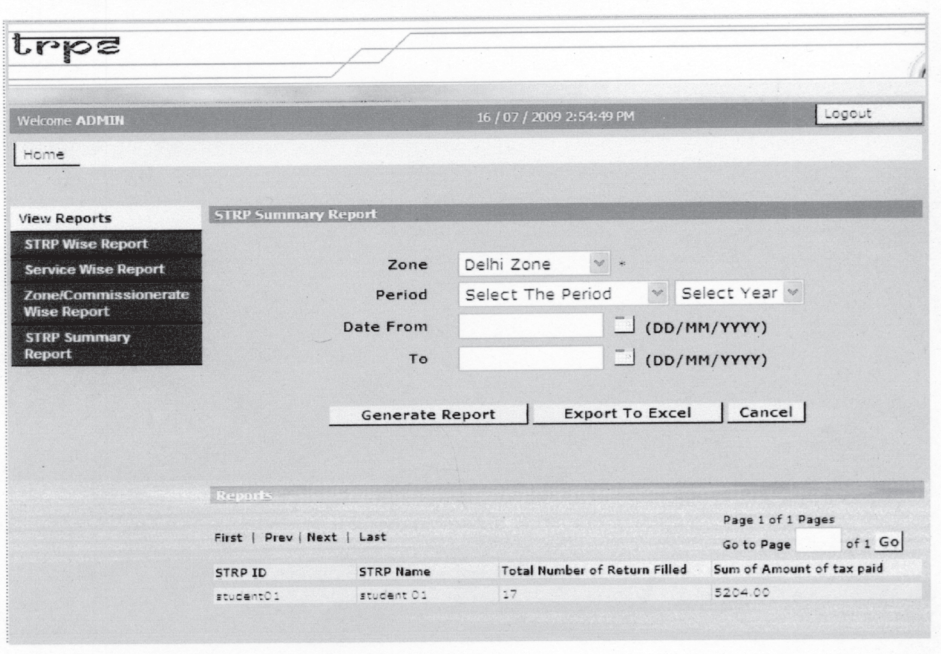
- This page should contain a "Service Type" drop down and "Date from" and "To" textboxes.
- To view the Service Wise Report data the admin can select multiple services from the "Service Type" list box and the data for those services will be populated. "Service Type" will be a mandatory field so the user has to select at least one service to view the report data.
- The user must click on the "Generate Report" button to view the report in HTML format or on "Export to Excel" if he or she wants to export the data them into Excel format.
- The TRP id, TRP name, and service type will also be provided in the Excel sheet.
- The "Cancel" button will take the user to the login page.
- The user needs to fill in both the "Date from" and "To" fields. "Date from" and "To" will extract the data based on "Date of Filling Return."
- STRPs Wise Report (Admin Report)

This report will allow the admin to search the data of the STRPs and will be able to generate a report of the data. The admin will generate reports in HTML format and also in Excel format.



- To view the STRPs Wise Report data users have to give a "Period" because its a mandatory field while the rest of the fields are non mandatory.
- The user can also provide the date range if the user wants data from a particular date range. If no date range is provided then all the data from all of the STRPs will be populated for the given period.
- The user needs to fill in both "Date from" and "To" fields. "Date from" and "To" will extract the data based on "Date of Filling Return."
- The user must click on the "Generate Report" button to view the report in HTML format or on "Export to Excel" if he or she wants to export the data into Excel format.
- The "TRP id" and "TRP name" will also be provided in the Excel sheet.
- The "Cancel" button will take the user to the login page.
- STRP Summary Report (Admin Report).

This report will allow the admin to generate a report for the top ten STRPs based on the highest amount of tax paid for each return filled by the TRP. This report will be generated in HTML format as well as in Excel format.



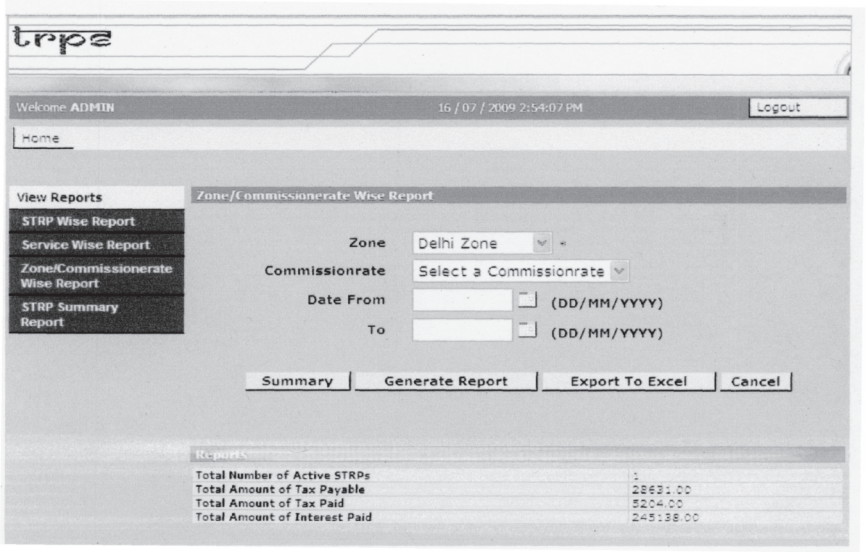
Validations:

- To view this report the user will have to select a "Zone" as well as a "Period." These are mandatory filters.
- There will be an option of "ALL" in the "Zone" drop down if the report needs to be generated for all the zones.
- The user must click on the "Generate Report" button to view the report in HTML format or on "Export to Excel" if he or she wants to export the data into Excel format.
- The "Cancel" button will take the user to the login page.
- The user needs to fill both "Date from" and "To" fields. "Date from" and "To" will extract the data based on "Date of Filling Return."

The user can either select the "Period" or "Date from" and "To" to generate the report. Both of the fields cannot be selected.

- Zone/Commissionerate Wise Report (Admin Report)

This report will allow the admin to generate the report Zone/Commissionerate Wise of STRPs. This report will be generated in HTML format as well as in Excel format.



Validations:

- To view the Commissionerate Wise Report data the admin can provided "Zone," "Commissionerate," and "Division" to view the data but if no input is provided then the data will include the entire "Zone," the "Commissionerate," and the "Division." The user will have to select "Zone" because it will be a mandatory field. There will be an option of "ALL" in the "Zone" drop down if the report needs to be generated for all of the "Zone."
- "Commissionerate" will be mapped to the "Zone" and "Division" will be mapped to "Commissionerate," i.e., if a user selects a "Zone" then all the "Commissionerate" under that "Zone" will come in to the "Commissionerate" drop down and if a user selects a "Commissionerate" then only those "Division" will be populated in the "Division" drop down that are under that "Commissionerate." If any LTU is selected in the "Zone" drop down the no other field will be populated.
- The user must click on the "Generate Report" button to view the report in HTML format or on "Export to Excel" if he or she wants to export the data into Excel format.
- The "TRP id," "TRP name," "Commissionerate," and "Division" will also be provided in the Excel sheet.
- The "Cancel" button will take the user to the login page.

2.3.2. Functions

It defines the fundamental actions that must take place in the software in accepting and processing the inputs and generating the outputs. The system will perform the following:

VALIDITY CHECKS

- The address should be correct.
- An Internet connection should be present.

RESPONSES TO ABNORMAL SITUATIONS

- An error message will be generated if the date format is wrong.
- An error message will be generated if the STC code is entered incorrectly.
- An error message will be generated if two users are assigned the same STC code.

2.3.3. Modules

Test Plan

We write test plans for two very different purposes. Sometimes the test plan is a product; sometimes it's a tool. It's too easy but also too expensive to confuse these goals. In software testing, a test plan gives detailed testing information regarding an upcoming testing effort including:

- Scope of testing
- Schedule
- Test deliverables
- Release criteria risks and contingencies
- How the testing will be done?
- Who will do it?
- What will be tested?
- How long it will take?
- What the test coverage will be, i.e., what quality level is required?

Test Cases

A test case is a set of conditions or variables under which a tester will determine if a requirement upon an application is partially or fully satisfied. It may take many test cases to determine that a requirement is fully satisfied. In order to fully test that all of the requirements of an application are met, there must be at least one test case for each requirement unless a requirement has

sub requirements. In that situation, each sub requirement must have at least one test case. There are different types of test cases.

- Common test case
- Functional test case
- Invalid test case
- Integration test case
- Configuration test case
- Compatibility test case

2.3.4. Performance Requirements

Static numerical requirements are:

- HTTP should be supported
- HTML should be supported
- Any number of users can be supported

Dynamic numerical requirements include the number of transactions and tasks and the amount of data to be processed within certain time periods for both normal and peak workload conditions depend upon the connection speed of the user.

2.3.5. Logical Database Requirements

The database must be updated and maintained on a regular basis by a database administrator.

2.3.6. Design Constraints

None.

2.3.7. Software System Attributes

Quality attributes that can serve as requirements:

- **Reliability:** It supports the latest functions as per the user requirements.
- **Availability:** It can be downloaded from the site.
- **Security:** It supports the privacy mode.
- **Portability:** It supports all operating systems.
- **Efficiency:** Appropriate amount of computing resources and code.

2.4. CHANGE MANAGEMENT PROCESS

Changes in project scope and requirements will be done if there is:

- Software update
- Change in technology (presence of any future OS)
- Change in user requirements

2.5. DOCUMENT APPROVAL

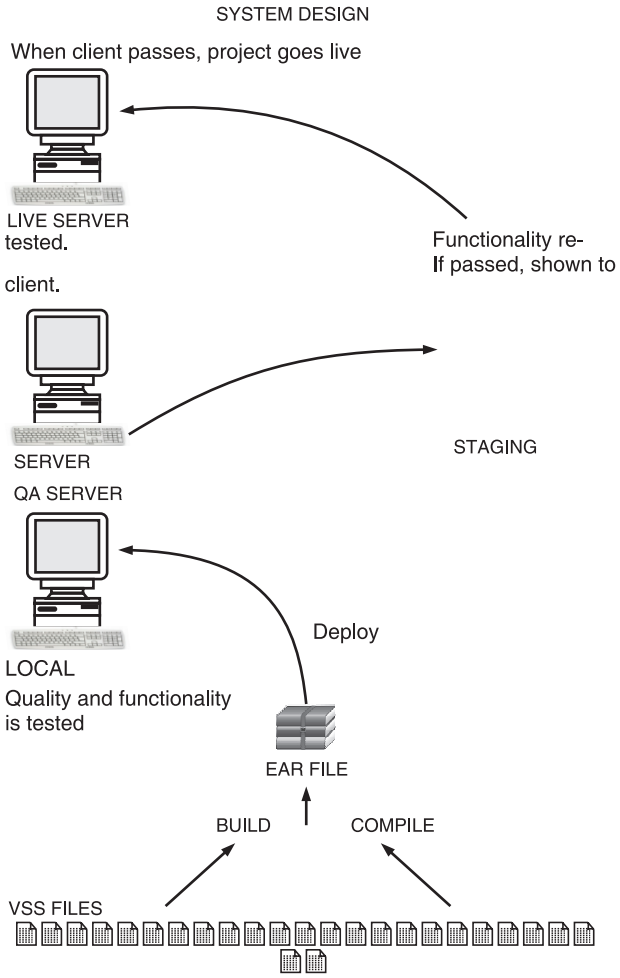
None.

2.6. SUPPORTING INFORMATION

The table of contents is given.

Chapter Three

System Design



Chapter Four
Reports And Testing

4.1. TEST REPORT

A management planning document that shows the following:

- **Test Item Transmittal Report:** Reporting on when tested software components have progressed from one stage of testing to the next.
- **Test Log:** Recording which test cases were run, who ran them, in what order, and whether each test passed or failed.
- **Test Incident Report:** Detailing for any test that failed, the actual versus expected result, and other information intended to throw light on why a test has failed.
- **Test Summary Report:** A management report providing any important information uncovered by the tests accomplished, and including assessments of the quality of the testing effort, the quality of the software system under test, and statistics derived from incident reports. The report also records what testing was done and how long it took in order to improve any future test planning. This final document is used to indicate whether the software system under test is fit for its purpose according to whether or not it has met acceptance criteria defined by project stakeholders.

4.2. TESTING

The importance of software testing and its implications with respect to software quality cannot be overemphasized. Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design, and code generation.

4.2.1. Types of Testing

White-Box Testing: This type of testing goes inside the program and check all the loops, paths, and branches to verify the program's intention.

Black-Box Testing: This type of testing is done to identify whether the output are the expected ones or not. This type of testing verifies that the software generates the expected output with a given set of inputs.

Static Analysis: In this type of testing, code is examined rather than exercised to verify its conformance to a set of criteria.

4.2.2. LEVELS OF TESTING

Unit Testing: This is the first phase of testing. The unit test of the system was performed using a unit test plan. The parameters that are required to be tested during a unit testing are as follows:

Validation check: Validations are all being performed correctly. For this, two kinds of data are entered for each entry going into the database—valid data and invalid data.

Integrating Testing: It is a systematic technique for constructing the program structure while at the same time conducting tests to uncover tests associated with interfacing. The objective is to take unit tested components and build a program structure that has been dictated by design.

In this testing we followed the bottom-up approach. This approach implies construction and testing with atomic modules.

Stress Testing: The final test to be performed during unit testing in the stress test. Here the program is put through extreme stress like all of the keys of the keyboard being pressed or junk data being put through. The system being tested should be able to handle that stress.

Functional Testing: Functional testing verifies that your system is ready for release. The functional tests define your working system in a useful manner. A maintained suite of functional tests:

- Captures user requirements in a useful way.
- Gives the team (users and developers) confidence that the system meets those requirements.

Load Testing: Load testing generally refers to the practice of modeling the expected usage of a software program by simulating multiple users accessing the program's services concurrently. As such, this testing is most relevant for multi-user systems often one built using a client/server model, such as web servers.

Regression Testing: Regression testing is initiated after the programmer has attempted to fix a recognized problem or has added the source code to a program that may have inadvertently introduced errors. It is a quality control measure to ensure that the newly modified code still complies with its specified requirements and that unmodified code has not been affected by the maintenance activity.

Installation Testing: Installation testing will check the installation and configuration procedure as well as any missing dependencies. Installation tests test the installation and configuration procedures. These tests are a set of scripts that automatically download all necessary packages and install them. To test that the installation was successful there is an edge-java-security-test RPM which contains a simple script to test the SSL handshake with the secure service.

CHAPTER FIVE
TEST CASES

5.1. RETURN FILED REPORT

Test case ID	Objective	Test steps	Test data	Expected results	Actual results	Test status (Pass/ Fail)	Bug ID
STRP_R FR_301	**To verify the availability of "Return Filed Report" to student role.**	**1.** Login as student. Homepage of the user appears.	Loginid: student01 password: pass123	A quicklink "Return Filed" appears on the left hand side of the screen under "View Reports"	Same as expected.	**PASS**	
STRP_R FR_302	**To verify the accessibility of "Return Filed" button.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed" under "View Reports" heading.	Loginid: student01 password: pass123	Return "Filed Report" page appears.	Same as expected.	**PASS**	**118560**
STRP_R FR_306	**To verify the report outputs in an Excel spreadsheet and HTML format.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Fill in the "Period" field. **4.** Click on the "Export to Excel" button. **5.** Next select the same period and click on the "Generate Report" button. **6.** Observe and verify the values under the respective column headings in HTML format with the Excel spreadsheet format.	Loginid: student01 password: pass123	The values under the respective columns in HTML and Excel spreadsheet should match. The column headings are as follows: Name STC Code Period Date of Filling Return Amount of Tax Payable Amount of Tax Paid Interest Paid	Same as expected.		

(*Continued*)

Test case ID	Objective	Test steps	Test data	Expected results	Actual results	Test status (Pass/ Fail)	Bug ID
STRP_R FR_307	**To verify the functionality of the "Generate Report" button when the "STC Code" filed is blank and the "Period" field is selected.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return on Filed Report" page appears. **3.** Fill all the mandatory fields except the "STC Code." **4.** Click on the "Generate Report" button.	Loginid: student01 password: pass123	**1.** Report should be generated for selected period showing correct values under the respective column headings. **2.** Message "No Record Found" should appear if no record for selected period exists.	Same as expected.	**PASS**	
STRP_R FR_308	**To verify the functionality of the "Export to Excel" button when the "STC Code" field is blank and the "Period" field is selected.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Fill all the mandatory fields except the "STC Code." **4.** Click on the "Generate Report" button.	Loginid: student01 password: pass123	1. "File Download" dialog box appears with options "Open," "Save," and "Cancel." 2. Report should be generated in Excel for selected period showing the correct values under the respective column headings. 3. Message "No Record Found" should appear if no record for selected period exists.	Same as expected.	**PASS**	
STRP_R FR_309	**To verify the functionality of the "Calendar" button on "Return Filed Report."**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Select a period in the "Period" field. **4.** Click on the "Pick a date" button.	Loginid: student01 password: pass123	A Date Time Picker Window should pop up with the current date selected in the calendar.	Same as expected.	**PASS**	

(*Continued*)

Test case ID	Objective	Test steps	Test data	Expected results	Actual results	Test status (Pass/ Fail)	Bug ID
STRP_10 FR_310	**To verify the format of the "STC Code" textbox.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Fill "STC Code" in the following template. STC code length: 15 characters 1-5: alphabetical 6-9: numerical 10th: alphabetical 11-12: ST 13-15: numerical **4.** Fill all the other Mandatory details. **5.** Click on the "Generate Report" button.		The report should be generated.	Same as expected.	**PASS**	
STRP_R FR_311	To verify the functionality of the "Generate Report" button when length of the "STC Code" is less than 15 characters.	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed Report," "Return Filed Report" page appears. **3.** Fill in the "STC Code" in the following template. STC code length: 14 characters 1-5: alphabetical (In Caps) 6-9: numerical 10th: alphabetical (In Caps) 11-12: ST 13-14: numerical **4.** Fill all the others Mandatory details. **5.** Click on the "Generate Report" button.	Loginid: student01 password: pass123 STC code: ASZDF23 45GST87 Period: April 1st - Sept 30th; 2007-2008	**1.** An error message should appear stating "STC Code is invalid." with a "Back" button. **2.** By clicking on the "Back" button, "Return Filed Report" page appears.	Same as expected.	**PASS**	

(*Continued*)

Test case ID	Objective	Test steps	Test data	Expected results	Actual results	Test status (Pass/ Fail)	Bug ID
STRP_R FR_312	**To verify the functionality of "Export To Excel" button when the length of the "STC Code" is less than 15 characters.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed Report," "Return Filed Report" page appears. **3.** Fill in the "STC Code" in the following template. STC code length: 14 characters. 1-5: alphabetical (In Caps) 6-9: numeral 10th: alphabetical (In Caps) 11-12: ST 13-14: numeral **4.** Fill all the others Mandatory details. **5.** Click on the "Export To Excel" button.	Loginid: student01 password: pass123 STC Code: ASZDF23 45GST87 Period: April 1st - Sept 30th; 2007-2008	**1.** An error message should appear stating "STC Code is invalid." with a "Back" button. **2.** By clicking on the "Back" button "Return Filed Report" page appears.	Same as expected.	**PASS**	
STRP_R FR_313	**To verify the functionality of the "Generate Report" button when the letters of the "STC Code" are written in small letters.**	1. Login as student. Homepage of the user appears. 2. Click on the quicklink "Return Filed Report," "Return Filed Report" page appears. 3. Fill in the "STC Code" in the following template. STC code length: 15 characters. 1-5: alphabetical (In Small) 6-9: numeral 10th: Alphabet (In Small) 11-12: ST 13-15: numeral 4. Select a period the "Period" field. 5. Click on the "Export To Excel" button.	Loginid: student01 password: pass123 STC Code: asdfg234 5gST87 Period: April 1st - Sept 30th; 2007-2008	An error message should appear stating "STC Code is invalid." With a "Back" button.	Same as expected.		

(*Continued*)

Test case ID	Objective	Test steps	Test data	Expected results	Actual results	Test status (Pass/ Fail)	Bug ID
STRP_R FR_317	**To verify the functionality of the "Generate Report" button when all of the characters of the "STC Code" are alphabetical.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed Report," "Return Filed Report" page appears. **3.** Fill in the "STC Code" in the following template. STC code length: 15 characters. 1-15: alphabetical (In Caps) **4.** Fill all the other Mandatory details. **5.** Click on the "Generate Report" button.	Loginid: student01 password: pass123 STC Code: ASZDFJU ILHGLO YU Period: April 1st - Sept 30th; 2007-2008	**1.** An error message should appear stating "STC Code is invalid." With a "Back" button. **2.** By clicking on the "Back" button "Return Filed Report" page appears.	Same as expected.	**PASS**	
STRP_R FR_321	**To verify the functionality of the "Generate Report" button when the date format is "dd/mm/ yyyy" in any or both of the "Date from" and "To" textboxes.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Fill the "Date from" and/or "To" in "dd/mm/yyyy" format. **4.** Select a period in the "Period" field. **5.** Click on the "Generate Report" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008 Date from: 10/01/2007	The report should be generated.	Same as expected.	**PASS**	
STRP_R FR_322	**To verify the functionality of the "Export To Excel" button when**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008	The report should be generated.	Same as expected.		

(*Continued*)

Test case ID	Objective	Test steps	Test data	Expected results	Actual results	Test status (Pass/ Fail)	Bug ID
	date format is "dd/ mm/yyyy" in any or both of the "Date from" and "To" textboxes.	**3.** Fill the "Date from the" in "dd/ mm/yyyy" format. **4.** Select a period in "Period" field. **5**. Click on the "Export To Excel" button.	Date from: 01/10/2007			**PASS**	
STRP_R FR_323	**To verify the functionality of the "Generate Report" button when the "Date from" and "To" fields are filled.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Fill the "Date from" and "To" fields in "dd/mm/ yyyy" format. **4.** Select a period in the "Period" field. **5.** Click on the "Generate Report" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008 "Date from": 01/10/2007 "To": 30/09/2008	The report should be generated if records exist in that period. Otherwise, the message "No Record Found." should display.	Same as expected.		
STRP_R FR_324	**To verify the functionality of the "Export To Excel" button when the "Date from" and "To" fields are filled.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Fill in the "Date from" and "To" fields in "dd/mm/ yyyy" format. **4.** Select a period in the "Period" field. **5.** Click on the "Export To Excel" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008 "Date from": 01/10/2007 "To": 30/09/2008	The report should be generated if the records exist in that period. Otherwise, the message "No Record Found." should display.	Same as expected.		

(*Continued*)

Test case ID	Objective	Test steps	Test data	Expected results	Actual results	Test status (Pass/ Fail)	Bug ID
STRP_R FR_325	**To verify the functionality of the "Generate Report" button when only the "Date from" field is filled and the "To" field is left blank.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Fill in the "Date from" field in "dd/ mm/yyyy" format. **4.** Select a period in the "Period" field. **5.** Click on the "Generate Report" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008 "Date from": 01/10/2007	The report should be generated if the records exist from the date entered in the "Date from" field. Otherwise, the message "No Record Found." should display.	Same as expected.		
STRP_R FR_326	**To verify the functionality of the "Export To Excel" button when only the "Date from" field is filled and the "To" field is left blank.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Fill in the "Date from" field in "dd/ mm/yyyy" format. **4.** Select a period in the "Period" field. **5.** Click on the "Export To Excel" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008 "Date from": 01/10/2007	The report should be generated if the records exist from the date entered in the "Date from" field. Otherwise, the message "No Record Found." should display.	Same as expected.		
STRP_R FR_327	**To verify the functionality of the "Generate Report" button when only the "To" field is filled in and the "Date from" field is left blank.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Fill in the "To" field in "dd/mm/ yyyy" format. **4.** Select a period in the "Period" field. **5.** Click on the "Generate Report" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008 "To": 30/09/2008	The report should be generated if the records exist until the date entered in the "To" field. Otherwise, the message "No Record Found." should display.	Same as expected.		

(*Continued*)

Test case ID	Objective	Test steps	Test data	Expected results	Actual results	Test status (Pass/ Fail)	Bug ID
STRP_RFR_328	**To verify the functionality of the "Export To Excel" button when only the "To" field is filled in and the "Date from" field is left blank.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Fill in the "To" field in "dd/mm/yyyy" format. **4.** Select a period in the "Period" field. **5.** Click on the "Export To Excel" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008 "To": 30/09/2008	The report should be generated if the records exist untill the date entered in the "To" field. Otherwise, the message "No Record Found." should display.	Same as expected.		
STRP_RFR_329	**To verify the functionality of the "Generate Report" button when the "Date from" is greater than the "To Date."**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Fill in the "Date from" and "To" field in "dd/mm/yyyy" format. **4.** Select a period in the "Period" field. **5.** Click on the "Generate Report" button.	Period: April 1st - Sept 30th; 2007-2008 "Date from": 01/10/2008 "To" date: 30/09/2008	An error message should appear saying, "From Date can not be greater than To Date."	Same as expected.	**PASS**	**112387**
STRP_RFR_330	**To verify the functionality of the "Export To Excel" button when the "Date from" is greater than the "To" Date.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Fill in the "Date from" and "To" fields in "dd/mm/yyyy" format. **4.** Select a period in the "Period" field. **5.** Click on the "Export To Excel" button.	Period: April 1st - Sept 30th; 2007-2008 "Date from": 01/10/2008 "To" Date: 30/09/2008	An error message should appear saying "From Date can not be greater than To Date."	Same as expected.	**PASS**	**11238**

(*Continued*)

Test case ID	Objective	Test steps	Test data	Expected results	Actual results	Test status (Pass/ Fail)	Bug ID
STRP_R FR_331	**To verify the Max Length of the "Date from" field.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Enter more than 10 characters in the "Date from" field. **4.** Enter a valid date in the "To" field. **5.** Select a period in the "Period" field. **6.** Click on the "Generate Report" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008 "Date from": 01/10/2008	An error message saying "Date Format of Start Date is not valid." should appear with the "Back" button. On clicking the "Back" button, the "Return Filed Report" page should appear.	Same as expected.	**PASS**	
STRP_R FR_333	**To verify the functionality of the "Home" button at the "Return Filed Report" page.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed Report," "Return Filed Report" page appears. **3.** Fill all of the mandatory fields with valid data. **4.** Click on the "Home" quicklink.	NA	Homepage of the user appears.	Same as expected.	**PASS**	
STRP_R FR_334	**To verify the functionality of the "Home" button at the error message page.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed Report," "Return Filed Report" page appears. **3.** Leave the "Period" field unselected. **4.** Click on the "Generate Report" button. **5.** Click on the "Home" quicklink.	NA	Homepage of the user appears.	Same as expected.	**PASS**	

(*Continued*)

Test case ID	Objective	Test steps	Test data	Expected results	Actual results	Test status (Pass/ Fail)	Bug ID
STRP_R FR_335	**To verify the functionality of the "Cancel" button on the "Return Filed Report" page.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed Report," "Return Filed Report" page appears. **3.** Click on the "Cancel" button.	NA	Homepage of the user appears.	Same as expected.	**PASS**	
STRP_R FR_336	**To verify the values of the "Period" drop down.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed Report," "Return Filed Report" page appears. **3.** Click on the "Period" drop down.	NA	The "Period" drop down should display two values: **1.** April 1st - Sept 30th **2.** Oct 1st - March 31st	Same as expected.	**PASS**	
STRP_R FR_337	**To verify whether the fields are retaining values or not after the error message appears.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed Report," "Return Filed Report" page appears. **3.** Leave the "Period" field unselected. **4.** Click on the "Generate Report" button. **5.** An error message appears. **6.** Click on the "Back" button.	Loginid: student01 password: pass!23 "Date from": 30/09/2008	When we click on the "Back" button the user comes back to "Return Filed Report" page and all the previous filled values remain intact.	Same as expected.	**PASS**	**118564**

(*Continued*)

Test case ID	Objective	Test steps	Test data	Expected results	Actual results	Test status (Pass/ Fail)	Bug ID
STRP_R FR_338	**To verify the pagination on the report output section.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed Report," "Return Filed Report" page appears. **3.** Select a period in the "Period" field. **4.** Click on the "Generate Report" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008	If the report output section contains more than 10 records, the pagination takes place and the next 10 records will be visible on the next page.	Same as expected.	**PASS**	
STRP_R FR_339	**To verify the pagination on the report output section when the number of records are less than 10.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed Report," "Return Filed Report" page appears. **3.** Select a period in the "Period" field. **4.** Click on the "Generate Report" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008	There will be only one page of output section and all of the pagination links are disabled.	Same as expected.	**PASS**	
STRP_R FR_340	**To verify the pagination on the report output section when the records are equal to 10.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed Report," "Return Filed Report" page appears. **3.** Select a period in the "Period" field. **4.** Click on the "Generate Report" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008	There will be only one page of output section and all of the pagination links are disabled.	Same as expected.	**PASS**	

(*Continued*)

Test case ID	Objective	Test steps	Test data	Expected results	Actual results	Test status (Pass/ Fail)	Bug ID
STRP_R FR_341	**To verify the pagination on the report output section when the records are greater than 10.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed Report," "Return Filed Report" page appears. **3.** Select a period in the "Period" field. **4.** Click on the "Generate Report" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008	The next 10 records will be visible on the next page and the "Next" and "Last" links are clickable.	Same as expected.	**PASS**	
STRP_R FR_342	**To verify the number of records on each page in the report output section.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed Report," "Return Filed Report" page appears. **3.** Select a period in the "Period" field. **4.** Click on the "Generate Report" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008	Every page of the report output section should contain a maximum of 10 records.	Same as expected.	**PASS**	
STRP_R FR_343	**To verify the functionality of the "Next" button on the pagination.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed Report," "Return Filed Report" page appears. **3.** Select a period in the "Period" field. **4.** Click on the "Generate Report" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008	By clicking on the "Next" button, the next page of the report output section appears.	Same as expected.	**PASS**	

(*Continued*)

Test case ID	Objective	Test steps	Test data	Expected results	Actual results	Test status (Pass/ Fail)	Bug ID
STRP_R FR_344	**To verify the functionality of the "Last" button on pagination.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed Report," "Return Filed Report" page appears. **3.** Select a period in the "Period" field. **4.** Click on the "Generate Report" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008	By clicking on the "Last" button, the last page of the report output section appears.	Same as expected.	**PASS**	
STRP_R FR_345	**To verify the functionality of the "First" button on pagination.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed Report," "Return Filed Report" page appears. **3.** Select a period in the "Period" field. **4.** Click on the "Generate Report" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008	By clicking on the "First" button, the first page of the report output section appears.	Same as expected.	**PASS**	
STRP_R FR_346	**To verify the functionality of the "Prev" button on pagination.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed Report," "Return Filed Report" page appears. **3.** Select a period in the "Period" field. **4.** Click on the "Generate Report" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008	By clicking on the "Prev" button, the previous page of the report output section appears.	Same as expected.	**PASS**	

(*Continued*)

Test case ID	Objective	Test steps	Test data	Expected results	Actual results	Test status (Pass/ Fail)	Bug ID
STRP_R FR_347	**To verify the functionality of the "Go" button when the user enters a page number in text box.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Select a period in the "Period" field. **4.** Click on the "Generate Report" button. **5.** Fill in a page number in the "Go to Page" textbox and click on the "Go" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008 Go to Page: 2	The entered page number will appear and the text above the "First Prev Next Last" link will show the current page.	Same as expected.	**PASS**	
STRP_R FR_348	**To verify the functionality of the "Go" button when the user enters an alphanumeric value in the text box.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Select a period in the "Period" field. **4.** Click on the "Generate Report" button. **5.** Fill in an alphabetical character in the "Go to Page" textbox and click on the "Go" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008	Textbox does not accept the value and remains blank.	Same as expected.	**PASS**	
STRP_R FR_349	**To verify the text of the page number details of the pagination.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Select a period in the "Period" field.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008	The page number details of the report output section should show the current page number in the format "Page (current page) of (Total pages) Pages."	Same as expected.	**PASS**	

(*Continued*)

Test case ID	Objective	Test steps	Test data	Expected results	Actual results	Test status (Pass/ Fail)	Bug ID
		4. Click on the "Generate Report" button. **5.** Fill in a page number in the "Go to Page" textbox and click on the "Go" button.					
STRP_R FR_350	**To verify the availability of the "First" and "Prev" links on the pagination.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Select a period in the "Period" field. **4.** Click on the "Generate Report" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008	If the report output section contains more than 10 records and the user is at last page of the pagination then the "First" and "Prev" links on the pagination should be enabled.	Same as expected.	**PASS**	
STRP_R FR_351	**To verify the availability of the "Next" and "Last" links on the pagination.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Select a period in the "Period" field. **4.** Click on the "Generate Report" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008	If the report output section contains more than 10 records and the user is at first page of the pagination then the "Next" and "Last" links on the pagination will be enabled.	Same as expected.	**PASS**	
STRP_R FR_352	**To verify the availability of the "First," "Prev," "Next," and "Last" links on the pagination.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," the "Return Filed Report" page appears.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008	If the report output section contains more than 10 records and the user is neither on the first page nor on the last page of the pagination then all four links on the pagination page should be enabled.	Same as expected.	**PASS**	

(*Continued*)

Test case ID	Objective	Test steps	Test data	Expected results	Actual results	Test status (Pass/ Fail)	Bug ID
		3. Select a period in the "Period" field. **4.** Click on the "Generate Report" button.					
STRP_RFR_353	**To verify the sorting order of the records in the report output section page.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Select a period in the "Period" field. **4.** Click on the "Generate Report" button.	Loginid: student01 password: pass123 Period: April 1st - Sept 30th; 2007-2008	The output section of the report should be sorted on the alphabetical order of column "Name."	Same as expected.	**PASS**	
STRP_RFR_354	**To verify the functionality of the "Logout" button on the "Return Filed Report" page.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Click on the "Logout" button.	Loginid: student01 password: pass123	The Login page of the website is displayed.	Same as expected.	**PASS**	
STRP_RFR_355	**To verify the functionality of the quicklinks on left side on the "Return Filed Report" page.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Return Filed," "Return Filed Report" page appears. **3.** Click on any of the quicklinks on left side of the page.	Loginid: student01 password: pass123	The quicklinks should be clickable and the respective page should be displayed.	Same as expected.	**PASS**	

5.2. MONTHLY/QUARTERLY TAX PAID FORM

Test case ID	Objective	Test steps	Expected results	Test status (Pass/Fail)
STRP_MQF_30 1	**To verify the availability of the "Monthly/Quarterly Tax Paid Form" quicklinks to the student role.**	**1.** Login as student role. Homepage of the user appears. **2.** Observe the quicklinks appearing under the "To Be Reported By STRP" section.	"Monthly/Quarterly Tax Paid Form" quicklink should appear under the "To Be Reported By STRP" section.	**PASS**
STRP_MQF_30 2	**To verify the accessibility of the "Monthly/Quarterly Tax Paid Form."**	**1.** Login as student role. Homepage of the user appears. **2.** Click the "Monthly/ Quarterly Tax Paid Form" quicklink appearing under the "To Be Reported By STRP" section in quicklink.	The "Monthly/ Quarterly Tax Paid Form" page should appear.	**PASS**
STRP_MQF_30 4	**To verify the "STRP Details" at the "Monthly/Quarterly Tax Paid Form" page.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Monthly/ Quarterly Tax Paid Form," the "Monthly/ Quarterly Tax Paid Form" page appears.	**1.** The "STRP ID" should show the login ID of the logged in user. **2.** The "STRP Name" should show the name of the logged in user. **3.** The "STRP PAN Number" should show the PAN No. of the logged in user.	**PASS**
STRP_MQF_30 5	**To verify the functionality of the "Submit" button when no value is entered in the "Name of Assessee" field.**	**1.** Login as student role. Homepage of the user appears. **2.** Click the "Monthly/ Quarterly Tax Paid Form" quicklink appearing under the "To Be Reported By STRP" section on the homepage. **3.** Do not enter any value in the "Name of Assessee" field.	**1.** The "Monthly/ Quarterly Tax Paid Form" should not get submitted. **2.** The following error message should appear with the "Back" button: "Name of Assessee is mandatory." **3.** Clicking the "Back" button should take the user to homepage.	**PASS**

(*Continued*)

Test case ID	Objective	Test steps	Expected results	Test status (Pass/Fail)
		4. Enter valid values in all mandatory fields. **5.** Click the "Submit" button.		
STRP_MQF_306	**To verify the functionality of the "Submit" button when no value is entered in the "STC Code" field.**	**1.** Login as student role. Homepage of the user appears. **2.** Click the "Monthly/ Quarterly Tax Paid Form" quicklink appearing under the "To Be Reported By STRP" section on the homepage. **3.** Do not enter any value in the "STC Code" field. **4.** Enter valid values in all of the mandatory fields. **5.** Click the "Submit" button.	**1.** The "Monthly/ Quarterly Tax Paid Form" should not get submitted. **2.** The following error message should appear with the "Back" button: "STC Code is mandatory for valid Return Filed." **3.** Clicking the "Back" button should take the user to the homepage.	**PASS**
STRP_MQF_308	**To verify the functionality of the "Submit" button when no value is entered in the "Amount of Tax Payable" field.**	**1.** Login as student role. Homepage of the user appears. **2.** Click the "Monthly/ Quarterly Tax Paid Form" quicklink appearing under the "To Be Reported By STRP" section on the homepage. **3.** Do not enter any value in the "Amount of Tax Payable" field. **4.** Enter valid values in all of the mandatory fields. **5.** Click the "Submit" button.	**1.** The "Monthly/ Quarterly Tax Paid Form" should not get submitted. **2.** The following error message should come with the "Back" button: "Amount of tax payable is mandatory for valid Return Filed." **3.** Clicking the "Back" button should take the user to the homepage.	**PASS**
STRP_MQF_310	**To verify the functionality of the "Submit" button when the value in the "STC Code" field is entered**	**1.** Login as student role. Homepage of the user appears. **2.** Click the "Monthly/ Quarterly Tax Paid Form" quicklink	The form should get submitted successfully and the following confirmation message should appear: "Record has been saved successfully."	**PASS**

(*Continued*)

Test case ID	Objective	Test steps	Expected results	Test status (Pass/Fail)
	in the following format: STC code length: 15 characters. 1-5: alphabetical 6-9: numeral 10th: alphabetical 11-12: ST 13-15: numeral	appearing under the "To Be Reported By STRP" section on the homepage. **3.** Enter a value of the "STC Code" in the following format: STC code length: 15 characters. 1-5: alphabetical 6-9: numeral 10th: alphabetical 11-12: ST 13-15: numeral **4.** Enter valid values in all of the mandatory fields. **5.** Click the "Submit" button.		
STRP_MQF_311	**To verify the max length of the "Amount of Tax Paid" textbox.**	Specification not provided.		
STRP_MQF_312	**To verify the max length of the "Name of Assessee" textbox.**	Specification not provided.		

5.3. MONTHLY/QUARTERLY TAX PAID FORM

Test case ID	Objective	Test steps	Expected results	Test status (Pass/Fail)
STRP_MQF_301	**To verify the availability of the "Monthly/Quarterly Tax Paid Form" quicklinks to student role.**	**1.** Login as student role. Homepage of the user appears. **2.** Observe quicklinks appearing under the "To Be Reported By STRP" section.	"Monthly/Quarterly Tax Paid Form" quicklink should appear under the "To Be Reported By STRP" section.	**PASS**

(*Continued*)

Test case ID	Objective	Test steps	Expected results	Test status (Pass/Fail)
STRP_MQF_302	**To verify the accessibility of the "Monthly/Quarterly Tax Paid Form."**	**1.** Login as student role. Homepage of the user appears. **2.** Click the "Monthly/ Quarterly Tax Paid Form" quicklink appearing under the "To Be Reported By STRP" section in quicklink.	"Monthly/Quarterly Tax Paid Form" page should appear.	**PASS**
STRP_MQF_304	**To verify the "STRP Details" at the "Monthly/Quarterly Tax Paid Form" page.**	**1.** Login as student. Homepage of the user appears. **2.** Click on the quicklink "Monthly/ Quarterly Tax Paid Form," "Monthly/ Quarterly Tax Paid Form" page appears.	**1.** The "STRP ID" should show the login ID of the logged in user. **2.** The "STRP Name" should show the name of the logged in user. **3.** The "STRP PAN Number" should show the PAN No. of the logged in user.	**PASS**
STRP_MQF_305	**To verify the functionality of the "Submit" button when no value is entered in the "Name of Assessee" field.**	**1.** Login as student role. Homepage of the user appears. **2.** Click the "Monthly/ Quarterly Tax Paid Form" quicklink appearing under the "To Be Reported By STRP" section in quicklink on the homepage. **3.** Do not enter any value in the "Name of Assessee" field. **4.** Enter valid values in all of the mandatory fields. **5.** Click the "Submit" button.	**1.** The "Monthly/ Quarterly Tax Paid Form" should not get submitted. **2.** The following error message should come with the "Back" button: "Name of Assessee is mandatory." **3.** Clicking the "Back" button should take the user to the homepage.	**PASS**

(*Continued*)

Test case ID	Objective	Test steps	Expected results	Test status (Pass/Fail)
STRP_MQF_306	**To verify the functionality of the "Submit" button when no value is entered in the "STC Code" field.**	**1.** Login as student role. Homepage of the user appears. **2.** Click the "Monthly/ Quarterly Tax Paid Form" quicklink appearing under the "To Be Reported By STRP" section on the homepage. **3.** Do not enter any value in the "STC Code" field. **4.** Enter valid values in all of the mandatory fields. **5.** Click the "Submit" button.	**1.** The "Monthly/ Quarterly Tax Paid Form" should not get submitted. **2.** The following error message should come with the "Back" button: "STC Code is mandatory for valid Return Filed." **3.** Clicking the "Back" button should take the user to the homepage.	**PASS**
STRP_MQF_308	**To verify the functionality of the "Submit" button when no value is entered in the "Amount of Tax Payable" field.**	**1.** Login as student role. Homepage of the user appears. **2.** Click the "Monthly/ Quarterly Tax Paid Form" quicklink appearing under the "To Be Reported By STRP" section on the homepage. **3.** Do not enter any value in the "Amount of Tax Payable" field. **4.** Enter valid values in all of the mandatory fields. **5.** Click the "Submit" button.	**1.** The "Monthly/ Quarterly Tax Paid Form" should not get submitted. **2.** The following error message should come with the "Back" button: "Amount of tax payable is mandatory for valid Return Filed." **3.** Clicking the "Back" button should take the user to the homepage.	**PASS**
STRP_MQF_310	**To verify the functionality of the "Submit" button when value in the**	**1.** Login as student role. Homepage of the user appears.	The form should get submitted successfully and the following	**PASS**

(*Continued*)

Test case ID	Objective	Test steps	Expected results	Test status (Pass/Fail)
	"STC Code" field is entered in the following format: STC code length: 15 characters. 1-5: alphabetical 6-9: numeral 10th: alphabetical 11-12: ST 13-15: numeral	**2.** Click the "Monthly/ Quarterly Tax Paid Form" quicklink appearing under the "To Be Reported By STRP" section on the homepage. **3.** Enter the value of the "STC Code" in the following format: STC code length: 15 characters. 1-5: alphabetical 6-9: numeral 10th: alphabetical 11-12: ST 13-15: numeral **4.** Enter valid values in all of the mandatory fields. **5.** Click the "Submit" button.	confirmation message should appear: "Record has been saved successfully."	
STRP_MQF_311	**To verify the max length of the "Amount of Tax Paid" textbox.**	Specification not provided.		
STRP_MQF_312	**To verify the max length of the "Name of Assessee" textbox.**	Specification not provided.		

DATABASE TEST CASE

Test case ID	Objective	Test steps	Expected results
TRP_ST_ MQF_501	**To verify the "STRP Details" at the "Monthly/ Quarterly Tax Paid Form" page.**	Execute the below script: select VC_LGN_CD,VC_ USR_F_NM,VC_ USR_L_ NM,VC_TRP_PN_ NMBR from UM_TB_USR where VC_LGN_CD = 'student01'	The "STRP Details" at the "Monthly/Quarterly Tax Paid Form" page and the scripts output should exactly match.

5.4. MONTHLY /QUARTERLY TAX PAID FORM

POSITIVE FUNCTIONAL TEST CASES

Test case ID	Objective	Test steps	Expected results	Test status (Pass/Fail)
STRP_ MQTR_301	**To verify the availability of the "Monthly/Quarterly Tax Paid" report to student role.**	**1.** Login as student role. Homepage of the user appears. **2.** Observe the quicklinks on left side of the homepage.	The "Monthly/ Quarterly Tax Paid Report" quicklink should appear under the "View Reports" section.	**PASS**
STRP_ MQTR_302	**To verify the accessibility of the "Monthly/Quarterly Tax Paid" report through quicklinks.**	**1.** Login as student role. Homepage of the user appears. **2.** On the homepage, under the "View Reports" click on the "Monthly/Quarterly Tax Paid" link.	The "Monthly/ Quarterly Tax Paid Report" page should appear.	**PASS**
STRP_ MQTR_304	**To verify the functionality of the "Generate Report" button when no value is entered in the "Period" field.**	**1.** Login as student. **2.** Go to "View reports" and the "Monthly/Quarterly Tax Paid" quicklinks, the "Monthly/ Quarterly Tax Paid Report" page appears. **3.** Do not enter any value in the "Period" field. **4.** Select the "Date from" and "To" fields from the "Date" picker control. **5.** Click the "Generate Report" button.	Report should not get generated and the following error message should come with the "Back" button: "Select The Period." Clicking the "Back" button should take the user to the "Monthly/Quarterly Tax Paid Report" page. **Note: This ensures that the "Period" field is mandatory.**	**PASS**
STRP_ MQTR_305	**To verify the functionality of the "Generate Report" button when no value is entered in the "To" date field.**	**1.** Login as student. **2.** Go to "View reports" and the "Monthly/Quarterly Tax Paid" quicklinks. **3.** Select a period from the "Period" drop down. **4.** Do not enter any value in the "To" date field.	**1.** The report should get generated. **2.** All of the records of the user should appear in the "Reports" output section.	**PASS**

(*Continued*)

Test case ID	Objective	Test steps	Expected results	Test status (Pass/Fail)
		5. Select a valid date in the "Date from" field from the Date picker control. **6.** Click the "Generate Report" button.		
STRP_ MQTR_306	**To verify the functionality of the "Generate Report" button when no value is entered in the "Date From" field.**	**1.** Login as student. **2.** Go to "View Reports" and the "Monthly/Quarterly Tax Paid" quicklinks. **3.** Select a period from the "Period" drop down. **4.** Do not enter any value in the "Date from" field. **5.** Select a valid date in the "To" date field from the Date picker control. **6.** Click the "Generate Report" button.	**1.** The report should get generated. **2.** All of the records of the user should appear in the "Reports" output section.	**PASS**
STRP_ MQTR_310	**To verify the values appearing in the "Assessment Year" drop down.**	**1.** Login as student. **2.** Go to "View Reports" and the "Monthly/Quarterly Tax Paid" quicklinks, the "Monthly/ Quarterly Tax Paid Report" page appears. **3.** Click the "year" drop down.	**1.** The "Assessment Year" drop down should have the following values: **a.** 2007-2008 **b.** 2008-2009 **c.** 2009-2010 **2.** These values should be sorted by ascending order.	**PASS**

NEGATIVE FUNCTIONAL TEST CASES

Objective	Test steps	Test data	Expected results
To verify the functionality of the "Generate Report" button when a string of characters is entered in the "Date from" field.	**1.** Login as student. **2.** Go to "View Reports" and the "Monthly/ Quarterly Tax Paid" quicklinks, the "Monthly/ Quarterly Tax Paid Report" page appears.	UserID: student01 password: password Period: April 1st - Sept 30th; 2007-2008 "Date from:" yrtyryg "To:" 31/07/2009	**1.** Report should not get generated and the following error message should come with the "Back" button: "Date Format of Start Date is not valid."

(*Continued*)

Objective	Test steps	Test data	Expected results
	3. Enter a valid date in the "To" date field. **4.** Enter a string of characters in the "Date from" field. **5.** Select a period from the "Period" drop down. **6.** Select a valid assessment year. **7.** Click the "Generate Report" button.		**2.** Clicking the "Back" button should take the user to the "Monthly/Quarterly Tax Paid Report" page.
To verify the functionality of the "Generate Report" button when a string of characters is entered in the "To" date field.	**1.** Login as student. **2.** Go to "View Reports" and the "Monthly/Quarterly Tax Paid" quicklinks, "Monthly/Quarterly Tax Paid Report" page appears. **3.** Enter a valid date in the "Date from" field. **4.** Enter a string of characters in the "To" date field. **5.** Select a "period" from the "Period" drop down. **6.** Select a valid assessment year. **7.** Click the "Generate Report" button.	UserID: student01 password: password Period: April 1st - Sept 30th; 2007-2008 "Date from:" 12/12/2007 "To:" utufgh	**1.** Report should not get generated and the following error message should come with the "Back" button: "Date Format of End Date is not valid." **2.** Clicking the "Back" button should take the user to the "Monthly/Quarterly Tax Paid Report" page.

5.5. SERVICE WISE REPORT (ADMIN REPORT)

POSITIVE FUNCTIONAL TEST CASE

Test case ID	Objective	Test steps	Expected results	Test status (Pass/Fail)
STRP_RFR_316	**To verify the functionality of the "Generate Report" button when the "Date from" is greater than the "To" date.**	**1.** Login as admin. Homepage of the user appears. **2.** Click on the quicklink "Service Wise Report," "Service Wise Report" page appears. **3.** Fill the "Date from" and "To" fields in "dd/mm/yyyy" format. **4.** Select a service type from the "Service Type" drop down. **5.** Click on the "Generate Report" button.	An error message should appear saying, "From Date can not be greater than To Date."	**PASS**

(*Continued*)

Test case ID	Objective	Test steps	Expected results	Test status (Pass/Fail)
STRP_RFR_317	**To verify the functionality of the "Export to Excel" button when the "Date from" is greater than the "To" date.**	**1.** Login as admin. Homepage of the user appears. **2.** Click on the quicklink "Service Wise Report," "Service Wise Report" page appears. **3.** Select a service type from the "Service Type" drop down. **4.** Fill the "Date from" and "To" fields in "dd/mm/yyyy" format. **5.** Click on the "Export to Excel" button.	An error message should appear saying, "From Date can not be greater than To Date."	**PASS**
STRP_RFR_318	**To verify the Max Length of the "Date from" field.**	**1.** Login as admin. Homepage of the user appears. **2.** Click on the quicklink "Service Wise Report," "Service Wise Report" page appears. **3.** Select a service type from the "Service Type" drop down. **4.** Enter more than 10 characters in the "Date from" field. **5.** Enter a valid date in the "To" field. **6.** Click on the "Generate Report" button.	An error message saying, "Date Format of Start Date is not valid." should appear with the "Back" button. On clicking the "Back" button the "Service Wise Report" page should appear.	**PASS**
STRP_RFR_319	**To verify the Max Length of the "To" field.**	**1.** Login as admin. Homepage of the user appears. **2.** Click on the quicklink "Service Wise Report," "Service Wise Report" page appears. **3.** Select a service type from the "Service Type" drop down. **4.** Enter more than 10 characters in the "To" field. **5.** Enter a valid date in the "Date from" field. **6.** Click on the "Generate Report" button.	An error message saying, "Date Format of End Date is not valid." should appear with the "Back" button. On clicking the "Back" button the "Service Wise Report" page should appear.	**PASS**

NEGATIVE FUNCTIONAL TEST CASE

Test case ID	Objective	Test steps	Test data	Expected results
TRP_ST_ MQF_401	**To verify the availability of the "Service Wise Report" to admin role.**	**1.** Login as student. Homepage of the user appears.	Loginid: student01 password: pass 123	A quicklink "Service Wise Report" will not appear at the left side of the screen.
TRP_ST_ MQF_402	**To verify the functionality of the "Generate Report" button when the date format is "nnm/dd/yyyy" in any or both of the "Date from" and "To" textboxes.**	**1.** Login as admin. Homepage of the user appears. **2.** Click on the quicklink "Service Wise Report," "Service Wise Report" page appears. **3.** Select a service type in the "Service Type" field. **4.** Fill in the "Date from" in "mm/dd/yyyy" format. **5.** Click on the "Generate Report" button.	Loginid: admin password: pass 123 Service Type: Stock Broking Date from: 10/15/2007 To: 31/10/2008	The report should not be generated and give the error "Date Format of Start Date is not valid." and/or "Date Format of End Date is not valid."
TRP_ST_ MQF_403	**To verify the functionality of the "Generate Report" button when the date format is "yyyy/mm/dd" in any or both of the "Date from" and "To" textboxes.**	**1.** Login as admin. Homepage of the user appears. **2.** Click on the quicklink "Service Wise Report," "Service Wise Report" page appears. **3.** Select a service type in the "Service Type" field. **4.** Fill in the "Date from" field in "yyyy/ mm/dd" and "To" field in "dd/mm/yyyy" format. **5.** Click on the "Generate Report" button.	Loginid: admin password: pass 123 Service Type: Stock Broking Date from: 2007/10/01 To: 31/10/2008	The report should not be generated and give the error "Date Format of Start Date is not valid."

Test case ID	Objective	Test steps	Test data	Expected results	Actual results	Test status (Pass/Fail)
STRP_ RFR_302	**To verify the accessibility of the "STRP Wise Report" button.**	**1.** Login as admin. Homepage of the user appears. **2.** Click on the quickink "STRP Wise Report" under the "View Reports" heading.	Loginid: admin password: pass123	"STRP Wise Report" page appears.	Same as expected.	**PASS**
STRP_ RFR_322	**To verify the functionality of the "Export to Excel" button when a date is entered in the "Date From" field and "To" field is left blank.**	**1.** Login as admin. Homepage of the user appears. **2.** Click on the quicklink "STRP Wise Report," "STRP Wise Report" page appears. **3.** Select a period from the "Period" drop down. **4.** Fill in the "Date from" in "dd/mm/ yyyy" format. **5.** Click on the "Export to Excel" button.	Period: April 1st - Sept 30th; 2007-2008 "Date from:" 30/09/2006	Report should be generated for records Where the "Date of Filling Return" is **after** "30/09/2006."	The message "No Record Found." is displayed even if records exist after the "Date of Filling Return" as "30/09/2006."	**FAIL**
STRP_ RFR_324	**To verify the functionality of the "Generate Report" button when a date is entered in the "To" field and the "Date from" field is left blank.**	**1.** Login as admin. Homepage of the user appears. **2.** Click on the quicklink "STRP Wise Report," "STRP Wise Report" page appears. **3.** Select a period from the "Period" drop down. **4.** Fill in the "To" in "dd/mm/yyyy" format. **5.** Click on the "Generate Report" button.	Period: April 1st - Sept 30th; 2007-2008 "To:" 26/06/2009	Report should be generated for records where "Date of Filling Return" is until "26/06/2009."	Message "No Record Found." is displayed even if records exist till 'Date of Filling "Return" as "26/06/2009."	**FAIL**

(*Continued*)

Test case ID	Objective	Test steps	Test data	Expected results	Actual results	Test status (Pass/Fail)
STRP_ RFR_325	**To verify the functionality of the "Export to Excel" button when a date is entered in the "To" field and "Date from" field is left blank.**	**1.** Login as admin. Homepage of the user appears. **2.** Click on the quicklink "STRP Wise Report," "STRP Wise Report" page appears. **3.** Select a period from the "Period" drop down. **4.** Fill the "To" in "dd/mm/yyyy" format. **5.** Click on the "Export to Excel" button.	Period: April 1st - Sept 30th; 2007-2008 "To:" 26/06/2009	Report should be generated for the records where the "Date of Filling Return" is until "26/06/2009."	The message "No Record Found." is displayed even if records exist after the "Date of Filling Return" as "26/06/2009."	**Fail**

5.6. STRPS WISE REPORT (ADMIN REPORT)

NEGATIVE FUNCTIONAL TEST CASES

Objective	Test steps	Test data	Expected results
To verify the functionality of the"Generate Report" button when special characters are entered in the "To" date field.	**1.** Login as student. **2.** Go to "Reports" and the "STRP Wise Report" quicklinks, the "STRP Wise Report" page appears. **3.** Enter valid date in the "Date From" field. **4.** Enter special characters in "Date From" field. **5.** Select a period from the "Period" drop down. **6.** Select a valid assessment year. **7.** Click the "Export to Excel" button.	UserID: admin password: password Period: April 1st - Sept 30th; 2007-2008 "Date from": 12/12/2007 "To:" #$$@	**1.** Report should not get generated and the following error message should come with the "Back" button: "Date Format of End Date is not valid." **2.** Clicking the "Back" button should take the user to the "STRP Wise Report" page.

(*Continued*)

Objective	Test steps	Test data	Expected results
To verify the functionality of the "Generate Report" button when an "n" digit numeric value is entered in the "Name" field.	Prequisite: User has entered "n" number of characters in the "Name" field while submitting his or her STRP Wise Report. **1.** Login as student. **2.** Go to "Reports" and the "STRP Wise Report" quicklinks, "STRP Wise Report" page appears. **3.** Select a period from the "Period" drop down. **4.** Click the "Submit" button.		**1.** User record for the selected period should appear in the "Reports" output section. **2.** It should not break the page.

CONCLUSION

a. Advantages:

- Delivery of a quality product and software met all quality requirements.
- Website is developed within the time frame and budget.
- A disciplined approach to software development.
- Quality products lead to happy customers.
- Software is developed within the time frame and budget.

b. Limitations:

- Quality is compromised to deliver the product on time and within budget.
- It is a time consuming process.
- It requires a large number of employees that leads to an increase in the product cost.

CHAPTER 13

OBJECT-ORIENTED TESTING

Inside this Chapter:

13.0. Basic Unit for Testing, Inheritance, and Testing
13.1. Basic Concepts of State Machines
13.2. Testing Object-Oriented Systems
13.3. Heuristics for Class Testing
13.4. Levels of Object-Oriented Testing
13.5. Unit Testing a Class
13.6. Integration Testing of Classes
13.7. System Testing (with Case Study)
13.8. Regression and Acceptance Testing
13.9. Managing the Test Process
13.10. Design for Testability (DFT)
13.11. GUI Testing
13.12. Comparison of Conventional and Object-Oriented Testing
13.13. Testing Using Orthogonal Arrays
13.14. Test Execution Issues
13.15. Case Study—Currency Converter Application

The techniques used for testing object-oriented systems are quite similar to those that have been discussed in previous chapters. The goal is to provide some test design paradigms that help us to perform Object-Oriented Testing (OOT).

Object-oriented software centers on a class (the basic unit) and the inheritance and encapsulation that affect a class. However, procedural programming controls the flow between software functions. Testing these object-oriented features call for new strategies.

13.0. BASIC UNIT FOR TESTING, INHERITANCE, AND TESTING

The class is the smallest unit for testing. This is due to the following reasons

a. Methods are meaningless apart from their class.

b. We can design method tests only if we take a class into consideration.

c. Class behavior should be verified before a class is reused or promoted to a library.

The class clusters are the practical unit for testing. This is due to the following reasons:

a. It is typically impossible to test a class in total isolation.

b. It is not cost effective to develop a complete set of drivers and stubs for every class.

c. A smaller testing budget usually means larger clusters.

We have different types of classes such as application specific, general-purpose, abstract or parameterized (template) classes. Therefore we need different test requirements for each class.

What About the Objects?

Testing a class instance (an object) can verify a class in isolation. However, when untested objects of tested classes are used to create an application system, the entire system must be tested before these objects can be considered to be verified.

The major problems that we face during object-oriented testing are:

a. Dynamic (or late) binding requires a separate test.

b. Complex inheritance structures.

c. Interface errors because OO programs typically have many small components and, therefore, more interfaces.

d. Objects preserve state but the state control, i.e., the acceptable sequence of events, is typically distributed over an entire program. This leads to state control errors.

e. Encapsulation is not a source of errors but may be an obstacle to testing.

So we conclude that we now require developing testable state models and state-based test suites.

What About Inheritance and Testing?

Weyuker gave some testing axioms that establishes limits on the reusability of test cases. These axioms are discussed below.

Axiom-1: *Antiextensionality.* The same function may be computed many ways, so test cases that cover method A do not necessarily cover method B, even if A and B are functionally equivalent.
Axiom-2: *Antidecomposition.* Test cases that cover a client class do not necessarily result in coverage of its server objects.
Axiom-3: *Anticomposition.* Tests that are individually adequate for each method in a class are not collectively adequate for the class. The individual test suites do not guarantee that all the class interactions will be tested.
Axiom-4: *General multiple change.* Even when two methods have the same structure (flow graph), test cases that cover method A do not necessarily cover B.

Let us now consider what these axioms indicate for testing under inheritance.

Case I: Extension

Suppose we change some methods in class A. Clearly,

- We need to retest the changed methods.
- We need to retest interaction among changed and unchanged methods.
- We need to retest unchanged methods, if data flow exists between statements, in changed and unchanged methods.

But, what about unchanged subclass B, which inherits from A? By anticomposition:

"The changed methods from A also need to be exercised in the unique context of the subclass."

Now, suppose we add or modify subclass B, without any change to superclass A. By antidecomposition and anticomposition:

"We need to retest inherited methods, even if they weren't changed." For example, say we develop a class acct, test it and add it to our class library.

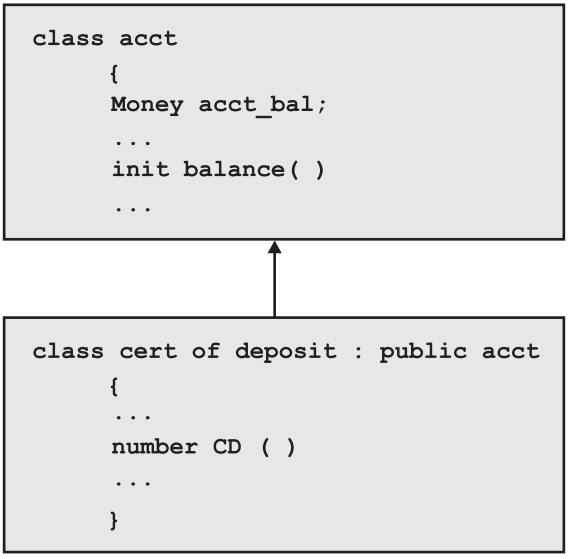


FIGURE 13.1 Extension Case.

Here, we will not find the error unless we retest both number (DC) and the inherited, previously tested, init balance( ) member functions.

Case II: Overriding

Suppose we add a subclass method B-1 which now overrides superclass method A-1. The new method should be tested.

Would the superclass test cases for A-1 necessarily provide the same coverage for the method B-1? The answer is:

a. No (by antiextensionality).

b. If the data domains and the control paths are only slightly different, the superclass test cases will probably not provide the same coverage.

This means that additional test cases are needed at each level of the inheritance hierarchy.

For example, suppose we develop a classes account and cert of deposit, test them, and then add both to our class library.

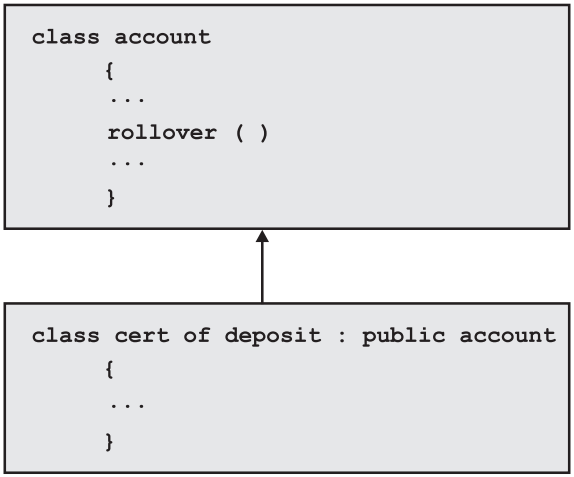


FIGURE 13.2

Later, we specialize cert of deposit to have its own rollover ( ). That is,

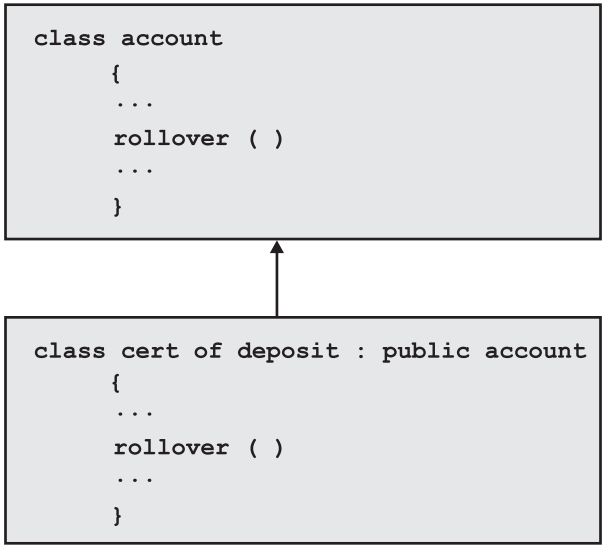


FIGURE 13.3 Overriding Case.

It is likely that both the specification and implementation of the new method will be different, so reusing the test cases for account : : rollover ( ) won't give us the coverage we need.

SOLVED EXAMPLES

EXAMPLE 13.1. Consider the following code for the shape class hierarchy.

```
Class Shape {
   private :
        Point reference_point;
   public :
   void put_reference_point (Point);
   point get_reference_point ( );
   void move_to (point);
   void erase ( );
   virtual void draw ( ) = 0;
   virtual float area ( );
        shape (point);
        shape ( );
   }
Class triangle : public shape {
   private :
        point vertex 2;
        point vertex 3;
   public :
        point get_vertex1 ( );
        point get_vertex2 ( );
        point get_vertex3 ( );
        void set_vertex1 (point);
        void set_vertex2 (point);
        void set_vertex3 (point);
   void draw ( );
   float area ( );
   Triangle ( );
   Triangle (point, point, point);
}
Class Equi Triangle : Public Triangle
   {
   public
        float area ( );
```

```
        equi triangle ( );
        equi triangle (point, point, point);
    }
```

What kind of testing is required for this class hierarchy?

SOLUTION. We can use method-specific retesting and test case reuse for the shape class hierarchy.

Let
D = denote, develop, and execute test suite
R = Reuse and execute superclass test suite
E = Extend and execute superclass test suite
S = Skip, super's test adequate
N = Not testable

Then, we get the following table that tells which type of testing is to be performed.

Method-Specific Test Reuse

The method-specific retesting and test case reuse for the shape class hierarchy is shown below.

Flattened class interface	Method type	Super	Method testing strategy												
Shape															
put_reference_point	Base		D												
get reference_point	Base			D											
move_to	Base				D										
erase	Base					D									
draw	Abstract Base						N								
area	Abstract Base							N							
Shape ( )	Constructor													D	
Shape (....)	Constructor														D
Triangle															
put_reference_point	Inherited	Shape	S												
get_reference_point	Inherited	Shape		S											

(*Continued*)

Flattened class interface	Method type	Super	Method testing strategy												
move_to	Inherited	Shape			S										
erase	Inherited	Shape				S									
draw	Overriding						D								
area	Overriding							D							
get_vertex1	Specialized								D						
get_vertex2	Specialized									D					
get_vertex3	Specialized										D				
set_vertex1	Specialized											D			
set_vertex2	Specialized												D		
set_vertex3	Specialized													D	
Triangle	Constructor														D
Triangle (....)	Constructor														D
Equi-Triangle															
put_reference_point	Inherited	Shape	S												
get_reference_point	Inherited	Shape		S											
move_to	Inherited	Shape			S										
erase	Inherited	Shape				S									
draw	Inherited	Triangle					R								
area	Overriding							E							
get_vertex1	Inherited	Triangle							S						
get_vertex2	Inherited	Triangle								S					
get_vertex3	Inherited	Triangle									S				
set_vertex1	Inherited	Triangle										S			
set_vertex2	Inherited	Triangle											S		
set_vertex3	Inherited	Triangle												S	
Equi-Triangle	Constructor														D
Equi-Triangle (....)	Constructor														D

D Develop and execute test suite.
R Reuse and execute superclass test suite.
E Extend and execute superclass test suite.
S Skip, super's tests adequate.
N Not testable.

Method-Specific Test Reuse

Each kind of derivation calls for a different method-specific test strategy.

		Pure extension (5)	Extension	Overriding	Specialization
Extent of subclass testing		None	Minimal (2)	Full (1)	Full
Source of black-box tests	Reuse superclass method tests?	NA	Yes	Maybe (4)	NA
	Develop new subclass method tests?	NA	Maybe (3)	Yes	Yes
Source of white-box tests	Reuse superclass method tests?	NA	Yes	No	NA
	Develop new subclass method tests?	NA	Maybe (3)	Yes	Yes

NA = Not Applicable

1. By antiextensionality (different implementation of the same function requires different tests).
2. By anticomposition (individual method test cases are not sufficient to test method interactions).
3. New test cases are needed when inherited features use any locally defined features.
4. Superclass test cases are reusable only to the extent that the super and subclass have the same pre- and post-conditions.
5. A pure extension subclass is an exception.
   - The subclass consists of entirely new features.
   - There are no interactions between the new and inherited features.

13.1. BASIC CONCEPTS OF STATE MACHINES

A state machine is an abstract, formal model of sequential computation. It is defined as a system whose output is determined by both current and past input. Although most classes retain some representation of previous messages, only certain classes are sensitive to the sequence of past message. A state machine can model the behavior of such classes. They are also called sequential machines.

State models have many uses in software engineering when sequence is important. Classes (objects) are readily modeled by state machines. It is the state of an object that may limit the messages it accepts. It typically determines its response to messages. The state determined message response patterns are called the behavior of an object. We can easily produce powerful test suites for behavior verification from a state model of class.

A system is said to exhibit state-based behavior when identical inputs are not always accepted and when they are accepted may produce different outputs. State machines are an engineering application of mathematical models known as finite automata.

State-Based Behavior

A state machine accepts only certain sequences of input and rejects all others. Each accepted input/state pair results in a specific output. State-based behavior means that the same input is not necessarily always accepted and when accepted, does not necessarily produce the same output.

This simple mechanism can perform very complex control tasks. Examples of sequentially constrained behavior include:

a. GUI interface control in MS-Windows

b. Modem controller

c. Device drivers with retry, restart, or recovery

d. Command syntax parser

e. Long-lived database transactions

f. Anything designed by a state model

The central idea is that sequence matters. Object behavior can be readily modeled by a state machine.

A state machine is a model that describes behavior of the system under test in terms of events (inputs), states, and actions (outputs).

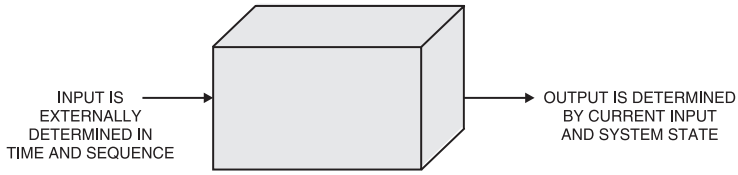


FIGURE 13.4

A state machine has four building blocks:

1. **State:** An abstraction which summarizes the information concerning past inputs that is needed to determine the behavior of the system on subsequent inputs.
2. **Transition:** Any allowable change from one state to another.
3. **Event:** An input, an interval of time, or other condition that causes a transition.
4. **Action:** The result or output that follows a transition.

Its graphic representation is shown in Figure 13.5.

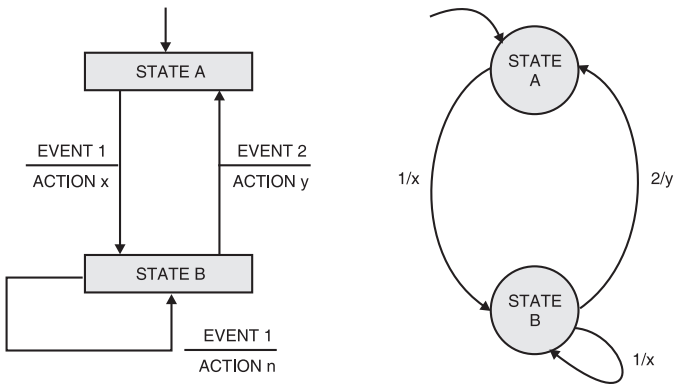


FIGURE 13.5 State Transition Diagrams.

- A transition takes the system from one state to another state.
- A transition has a pair of states, the accepting state and the resultant state.
- A machine may be in one state at a time.
- The current state refers to the active state.

A mechanism of a state machine has several steps. They are as follows:

Step 1. The machine begins in the initial state.
Step 2. The machine waits for an event for an indefinite interval.
Step 3. An event presents itself to the machine.
Step 4. If the event is not accepted in the current state, it is ignored.
Step 5. If the event is accepted in the current state, the designated transition is said to fire. The associated output action (if any) is produced and the state designated as the resultant state becomes the current state. The current and the resultant state may be the same.
Step 6. The cycle is repeated from Step 2 unless the resultant state is the final state.

UML notations of state-transition diagram (STD) is given next.

It is a graphic representation of a state machine.

- Nodes represent states.
- Arrows represent transitions.
- The annotations on the edge represent events and actions.

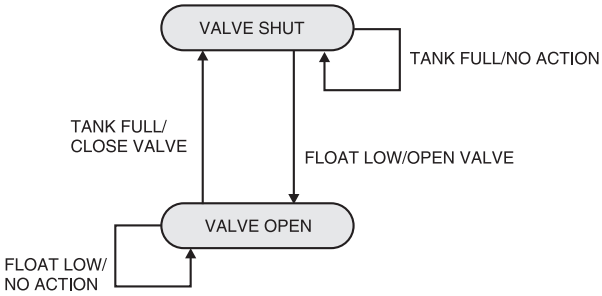


FIGURE 13.6 STD of Water Tank System.

A Water-Tank Example—STD

We draw STD of a water tank system with a valve. This valve can be in one of the two states—shut or open. So, we have the following:

State Tables/State Transition Table

A state to state table shows current state (rows) by next state (columns). Cells with an event/action pair define transactions. State-transition diagrams (STDs) are useful with a relatively small number of states (up to 20 states). For more states, state transition tables are compact and ease systematic examination.

Properties of Finite State Automata

Some properties of finite automata have practical implications for testing. They are as follows:

1. Any two states are equivalent if all possible event sequences applied to these states result in identical output action sequences. That is, if we start in either state and apply all possible combination of events up to and including those that result in a final state, we cannot tell from which state we started by comparing output actions.

   A minimal state machine has no equivalent states. A model with equivalent states is at best redundant and probably incorrect or incomplete in some more serious way.

2. A state, S_i, is reachable from another state, S_j, when a legal event sequence takes the machine from S_i to S_j.

   When some state is said to be reached without specifying a starting state, the initial state is assumed.

   Reachability analysis is concerned with determining which states are reachable.

   Non reachable states arise for several reasons:

   a. **Dead state:** No other state is reachable from it.

   b. **Dead loop:** No state outside of the loop may be reached.

   c. **Magic state:** It has no inbound transition but provides transition to other states. It is an extra initial state.

3. A dead state/loop is appropriate in some rare cases but generally is an error.

4. A magic state is always an error.

Mealy/Moore Machines

There are two main variants of state models (named for their developers).

Moore Machine:

- Transitions do not have output.
- An output action is associated with each state. *States are active.*

Mealy Machine:

- Transitions have output.
- No output action is associated with state. *States are passive.*
- In software engineering models, the output action often represents the activation of a process, program, method, or module.

Although the Mealy and Moore models are mathematically equivalent, the Mealy type is preferred for software engineering.

A passive state can be precisely defined by its variables. When the same output action is associated with several transitions, the Mealy machine provides a more compact representation.

Conditional/Guarded Transitions

Basic state models do not represent conditional transitions. This is remedied by allowing a Boolean conditions on event or state variables.

Consider a state model for stack class. It has three states: empty, loaded, and full.

We first draw the state machine model for a STACK class, without guarded transitions. The initial state is "Empty."

What is a guard? It is a predicate expression associated with an event. Now, we draw another state machine model for STACK class, with guarded transitions.

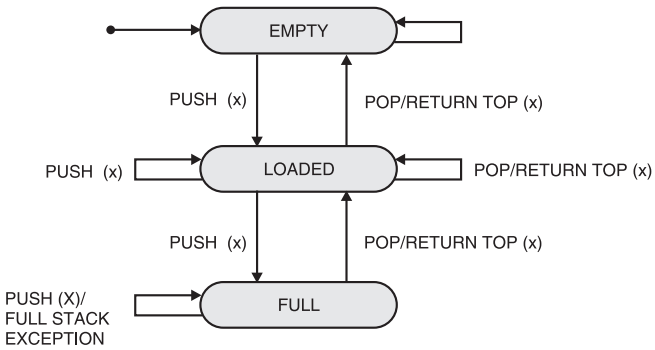


FIGURE 13.7 State Model of a Stack Without Guards.

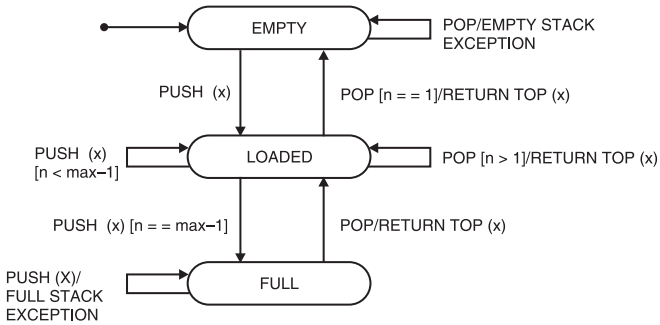


FIGURE 13.8 State Machine Model of Stack With Guards.

Conditions/guards are Boolean expressions on:

a. Variables of the class under test.

b. Parameters of the messages sent to the class under test.

There are two kinds of conditions:

a. *Acceptance conditions* specify the conditions under which a message is accepted.

b. *Resultant conditions* specify the conditions under which one of the several possible states may result from the same message.

EXAMPLE 13.2. Draw a state model of a two-player volley ball game.

SOLUTION.

- The game starts.
- The player who presses the start button first gets the first serve. The button press is modeled as the player-1 start and player-2 start events.
- The current player serves and a volley follows. One of three things end the volley.
- If the server misses the ball, the server's opponent becomes the server.
- If the server's opponent misses the ball, the server's score is incremented and gets another chance.
- If the server's opponent misses the ball and the server's score is at the game point, the server is declared winner.

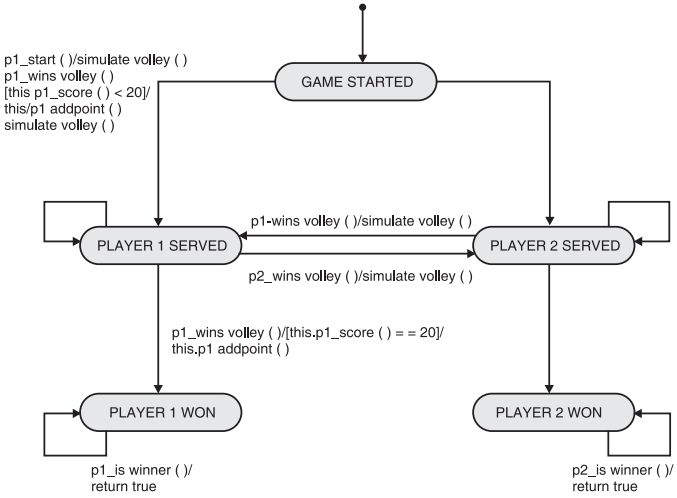


FIGURE 13.9

Here, 20 is the game point, so a score of 21 wins.

Design Guidelines–Basic Requirements

1. One state must be designated as the initial state.
2. At least, one state must be designated as the final state. If not, assumptions about termination should be made explicit.
3. There are no equivalent states. (Equivalent states produce exactly the same response for all events visible to the machine in question.)
4. Every defined event and every defined action must appear in at least one transition.
5. There should be an explicit definition of the mechanism to handle events which are implicitly rejected.

Limitations of the Basic State Models

1. State diagrams can be easily drawn or read up to 20 states. Tabular representation helps but is not suitable for large states.
2. Concurrency cannot be modeled. Only a single state may be active, so the basic model cannot accommodate two or more simultaneous transition processes. Only one state may be designated as the current state. For example, tasks running on separate computers in a client/server systems.

3. With even a moderately large number of states, a state diagram becomes a spaghetti ball.
4. Basic models do not offer any mechanisms for partitioning or hierarchic abstraction.

   Statecharts extend the basic model to remove these deficiencies.

What Are Statecharts?

Statecharts use a graphic shorthand to avoid enumeration of all states. They overcome the basic machine's scalability and concurrency limitations.

Properties of Statecharts

1. They use two types of state: group and basic.
2. Hierarchy is based on a set-theoretic for malism (hypergraphs).
3. Easy to represent concurrent or parallel states and processes.

Therefore, we can say that:

Statecharts = State diagram
+
Depth
+
Orthogonality
+
Broadcast Communication

A basic state model and its equivalent statechart are shown below:

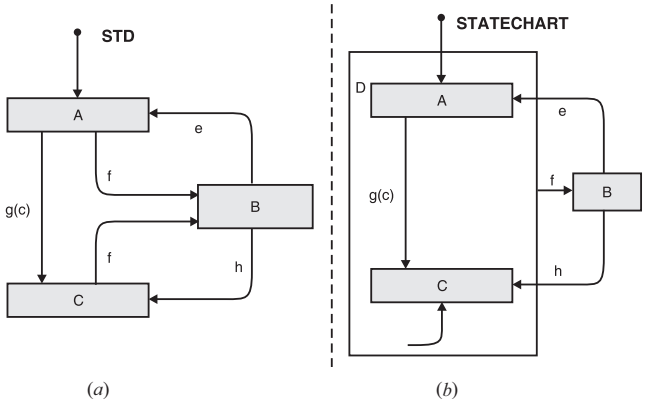


FIGURE 13.10

We have already discussed the STD. We now discuss its equivalent, statechart. In the Figure 13.10(b), we observe the following:

1. State D is a super state. It groups states A and C because they share common transitions.
2. State A is the initial state.
3. Event f fires the transition AB or CB, depending on which state is active.
4. Event g fires AC but only if C is true (a conditional event).
5. Event h fires BC because C is marked as the default state.
6. The unlabelled transition inside of state D indicates that C is the default state of D.

Statecharts can represent hierarchies of single-thread or concurrent state machines.

- Any state may contain substates.
- The substate diagram may be shown on a separate sheet.
- Decomposition rules are similar to those used for data flow diagrams (DFD). Orthogonal superstates can represent several situations.
- The interaction among states of separate classes (objects).
- The non interaction among states of separate processes which proceed independently. For example, "concurrent," "parallel," "multi-thread," or "asynchronous" execution.

Statecharts have been adapted for use in many OOA/OOD methodologies including:

- Booch OOD
- Object modelling technique (OMT)
- Object behavior analysis (OBA)
- Fusion
- Real-time object-oriented modeling (ROOM)

EXAMPLE 13.3. Consider a traffic light control system. The traffic light has five states—red, yellow, green, flashing red, and OFF. The event, "power on," starts the system but does not turn on a light. The system does a self test and if no faults are recognized, the no fault condition becomes true. When a reset event occurs and the no fault condition holds, the red on event is generated. If a fault is raised in any state, the system raises a "Fault" event and returns to the off state. Draw its

a. *State transition diagram.*

b. *State chart.*

SOLUTION. (a) We first draw its state transition diagram (STD) shown in Figure 13.11.

The unlabeled transition inside the cycling state shows that red is the default state of cycling.

An aggregration of states is a superstate and the model within an aggregation is a process. The entire model is also a process, corresponding to the entire system under study.

The traffic light model has three processes: the traffic light system, the on process, and the cycling process.

The state enclosed in the undivided box are mutually exclusive indicating that the system may be in exactly one of these states at a time. This is called *XOR decomposition.*

The two superstates "on" and "cycling" show how transitions are simplified.

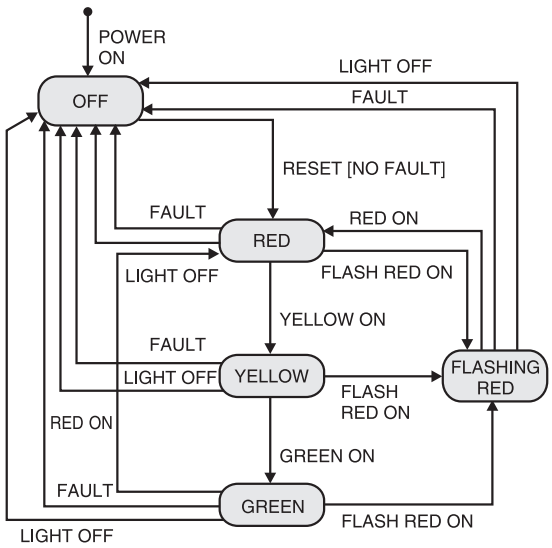


FIGURE 13.11 STD for Traffic Light.

- Superstate "on" means the system is either cycling or flashing red.
- Superstate cycling groups are the state red, yellow, and green because they share common transitions.
- Off is the initial state, reached by the power on event.
- The event flashing red on fires the transition red-flashing red on, yellow-flashing red on, or green-flashing red on depending on which state is active.
- The event reset fires off-on, but only if no fault is true. This is a guarded transition.

- The event reset fires off-red on because red on is marked as the only default state with both superstates on and cycling.
- In a state chart, a state may be an aggregate of other states (a superstate) or an atomic state.
- In the basic state model, one transition arrow must exist for each transition having the same event, the same resultant state but different accepting states. This may be represented in a statechart with a single transition from a superstate.

Figure 13.12 shows the statechart for the traffic light system.

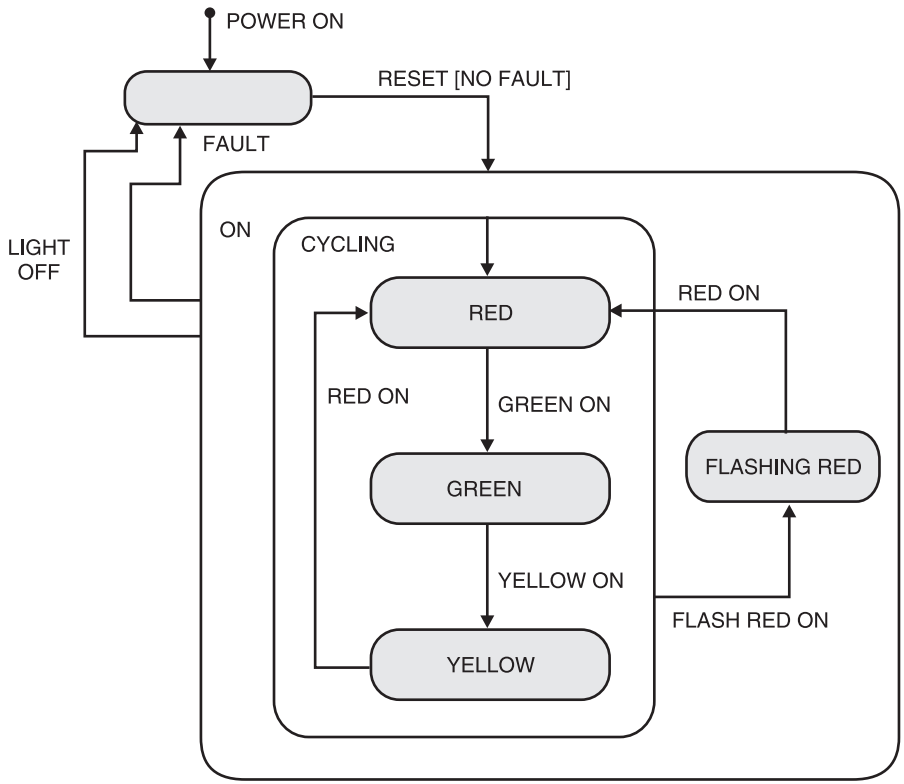


FIGURE 13.12 Statechart for the Traffic Light.

Concurrent states are represented by divided superstates. This is called *AND decomposition*. That is, one state or superstate in each partition is active.

EXAMPLE 13.4. Consider the behavior of two automotive control systems: Crusive control and antilock brake control. These systems share a common event (brake applied) but are otherwise independent. Draw its statechart.

SOLUTION. Please note the following points here.

- To be in the moving state means that both substates cruise control and antilock brake control are active.
- The substates of cruise control (off, suspended, and active) are mutually exclusive as are the substates of antilock brake control (free wheeling, braking, and modulating).

Multiple transitions are used to model constructor behavior. Subclass constructors typically percolate up the hierarchy so a complete flattened model must show the chain of events that happens upon construction of a subclass.

Concatenation

Concatenation involves the formation of a subclass that has no locally defined features other than the minimum requirement of a class definition.

State Space

A subclass state space results from the two factors:

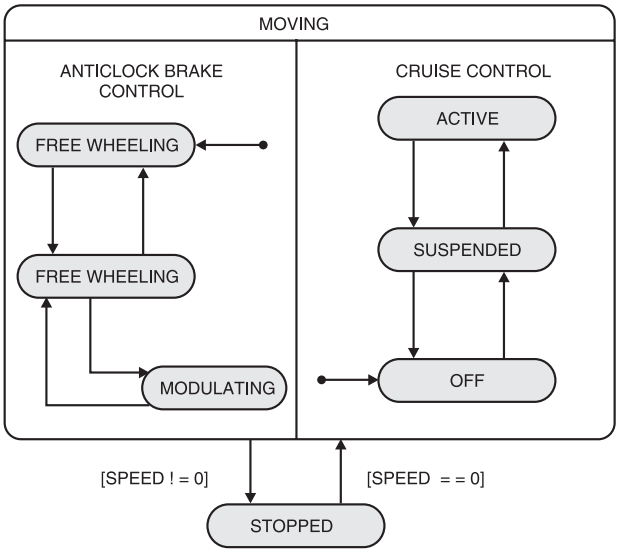


FIGURE 13.13

a. The superclass state space is inherited to the extent that superclass instance variables are visible.

b. The subclass adds state space dimensions to the extent that it defines or overrides instance variables.

For a subclass to be well-formed and testable:

a. *Orthogonal composition:* Each subclass inherits all of the non private (public) superclass features without redefinition and redefines some new local features. Subclass extensions must be orthogonal. The inherited state space must not be spindled, folded, or mutilated by the extensions.

b. The *inherited state space* must be the same or smaller. The subclass should not redefine inherited instance variables in any other way.

c. Superclass private variables must be orthogonal in the state space formed by inheritable variables. In other words, the effect of superclass private variables on the superclass state space must be additive.

TIPS *Class hierarchies that do not meet these conditions are very likely to be buggy.*

How State Machines Fail?

i. State machines fail due to control faults like:

a. A missing or incorrect transition (the resultant state is incorrect).

b. A missing or incorrect event (a valid message is ignored).

c. A missing or incorrect action.

d. An extra, missing, or corrupt state (behavior becomes unpredicatable).

e. A sneak path (a message is accepted when it should not be).

f. An illegal message failure (an unexpected message causes a failure).

g. A trap door (the implementation accepts undefined messages).

Missing transition:

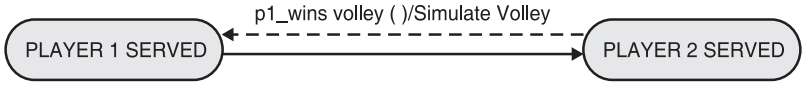


FIGURE 13.14

Player-2 loses the volley but continues as server.

Incorrect transition/resultant state: After player-2 missed, the game resets.

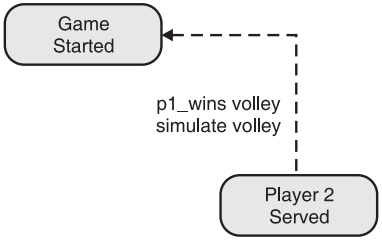


FIGURE 13.15

Missing action:

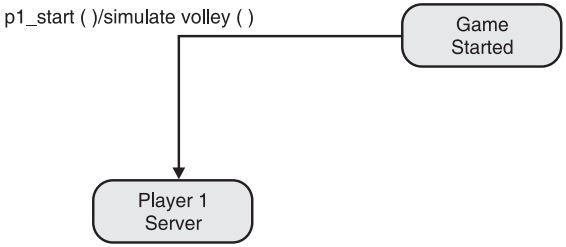


FIGURE 13.16

No volley is generated. System will wait indefinitely.

ii. State machines fail due to an incorrect actions like:

- **a.** Sneak path
- **b.** Corrupt state
- **c.** Illegal message failure
- **d.** Trap doors

Sneak path: The implementation accepts an event that is illegal or unspecified for a state. For example, when player-2 is serving, player-2 can win if his or her start button is pressed.

Corrupt state: The implementation computes a state that is not valid.

Player-1 is serving at game point and player-2 misses, the game crashes and cannot be restarted.

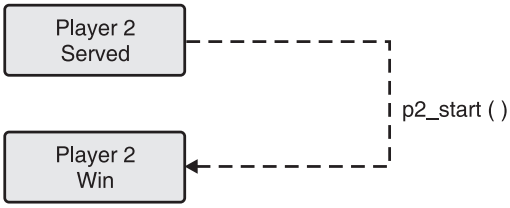


FIGURE 13.17

Illegal message failure: The implementation fails to handle an illegal message correctly. Incorrect output is produced, the state is computed or both.

For example, if player-1 presses the player select button after serving, the game crashes and can't be restarted.

Trap door: The implementation accepts an event that is not defined in the specification. For example, when player-1 is serving, player-1 can win any time by pressing the scroll lock key. A trapdoor can result from:

1. Obsolete features that were not removed when a class was revised.
2. Inherited features that are inconsistent with the requirements of a subclass.
3. Undocumented features added by the developer for debugging purposes.

EXAMPLE 13.5. Consider a class TwoPlayerGame. This class acts as a base class. The class ThreePlayerGame is derived from this class. Draw its class hierarchy and statecharts at class scope. Also, draw its flattened transition diagram for ThreePlayerGame.

SOLUTION. We draw its class hierarchy first. This diagram is known as a class diagram. Both of them are shown in Figure 13.18. Its flattened diagram is shown in Figure 13.19.

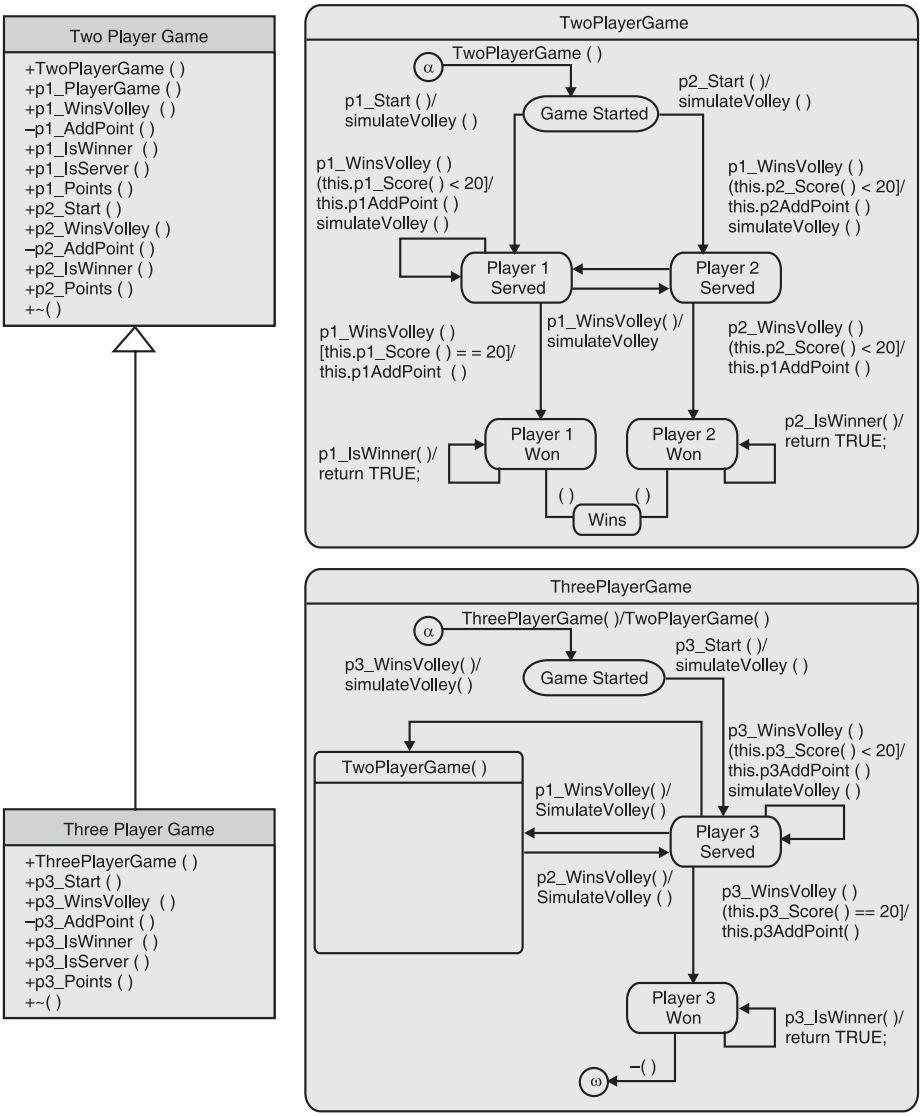


FIGURE 13.18 ThreePlayerGame Class Hierarchy and Statecharts at Class Scope.

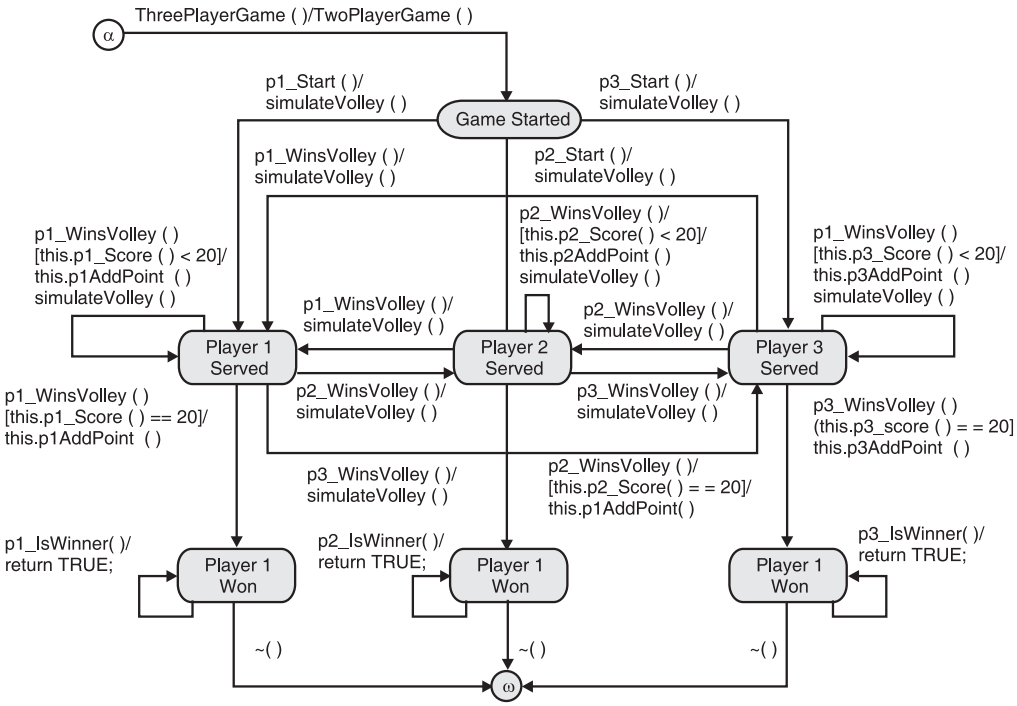


FIGURE 13.19 Flattened Transition Diagram, Three Player Game.

EXAMPLE 13.6. Write an algorithm that produces a transition tree from a state transition diagram (STD). Draw a transition tree for three player game.

SOLUTION.

1. The intial state is the root node of the tree.
2. Examine the state that corresponds to each non terminal leaf node in the three and each outbound transition on this state. At least one new edge will be drawn for each transition. Each new edge and node represents an event and resultant state reached by an outbound transition.
   - **a.** If the transition is unguarded, draw one new branch.
   - **b.** If the transition guard is a simple predicate or a complex predicate composed of only AND operators, draw one new branch.
   - **c.** If the transition guard is a complex predicate using one or more OR operators, draw a new branch for each truth value combination that is sufficient to make the guard TRUE.

3. For each edge and node drawn in Step 2:
   a. Note the corresponding transition event, guard, and action on the new branch.
   b. If the state that the new node represents is already represented by another node (anywhere in the diagram) or is a final state, mark this node as terminal because no more transitions are drawn from this node.
4. Repeat Steps 2 and 3 until all leaf nodes are marked terminal.

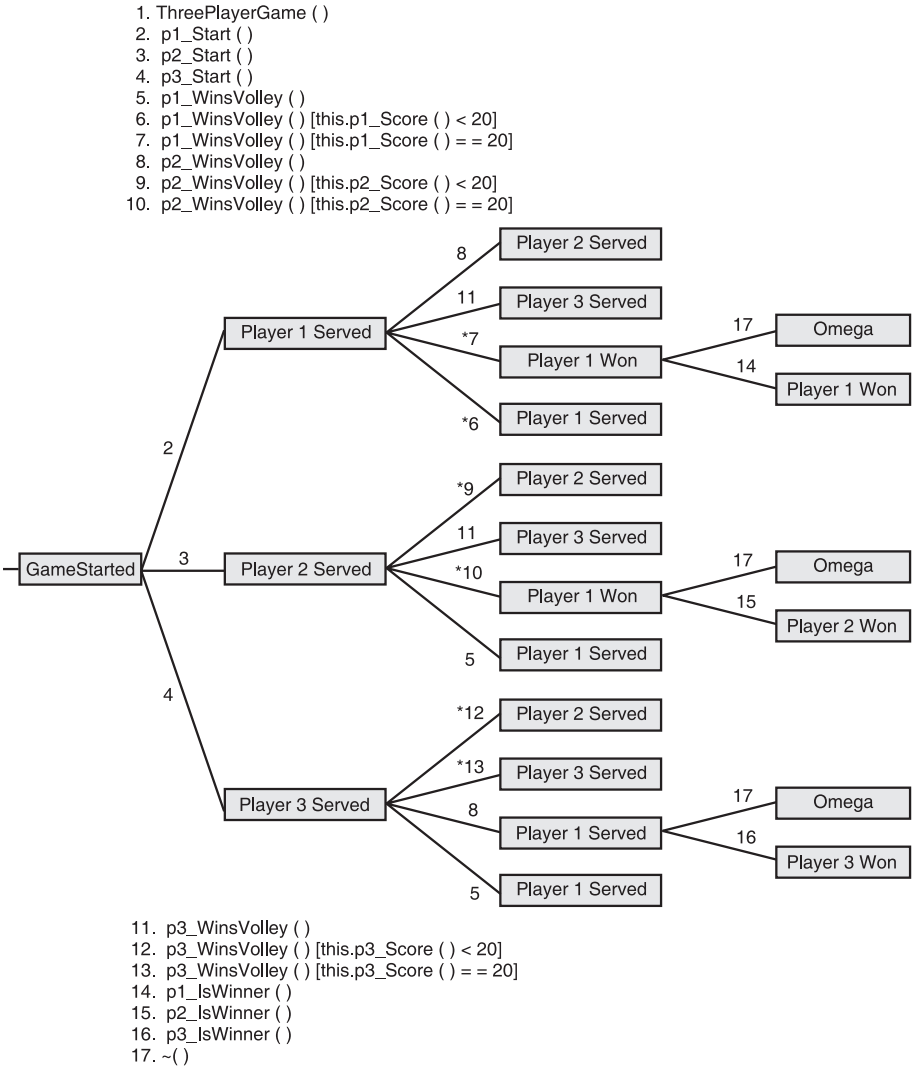


FIGURE 13.20 Transition Tree for Three Player Game.

EXAMPLE 13.7. For a three player game, what conformance test suite will you form? Derive the test cases.

SOLUTION.

TABLE 13.1 Conformance Test Suite for Three Player Game.

	Test case input		Expected result	
TCID	**Event**	**Test condition**	**Active**	**State**
1.1	Three Player Game			Game Started
1.2	p1_start		simulateVolley	Player 1 Served
1.3	p2_WinsVolley		simulateVolley	Player 2 Served
2.1	Three Player Game			Game Started
2.2	p1_start		simulateVolley	Player 1 Served
2.3	p3_WinsVolley		simulateVolley	Player 3 Served
3.1	Three Player Game			Game Started
3.2	p1_start		simulateVolley	Player 1 Served
3.3				Player 1 Served
3.4	p1_WinsVolley	p1_Score = = 20		Player 1 Won
3.5	dtor			omega
4.1	Three Player Game			Game Started
4.2	p1_start		simulateVolley	Player 1 Served
4.3				Player 1 Served
4.4	p1_WinsVolley	p1_Score = = 20		Player 1 Won
4.5	p1_IsWinner		return TRUE	Player 1 Won
5.1	Three Player Game			Game Started
5.2	p1_start		simulateVolley	Player 1 Served
5.3				Player 1 Served
5.4	p1_WinsVolley	p1_Score = = 19	simulateVolley	Player 1 Served
6.1	Three Player Game			Game Started

(Continued)

TCID	Test case input		Expected result	
	Event	Test condition	Active	State
6.2	p2_start		simulateVolley	Player 2 Served
6.3				Player 2 Served
6.4	p2_WinsVolley	p2_Score = = 19	simulateVolley	Player 2 Served
7.1	Three Player Game			Game Started
7.2	p2_start		simulateVolley	Player 2 Served
7.3	p3_WinsVolley		simulateVolley	Player 3 Served
8.1	Three Player Game			Game Started
8.2	p2_start		simulateVolley	Player 2 Served
8.3				Player 2 Served
8.4	p2_WinsVolley	p2_Score = = 20		Player 2 Won
8.5	dtor			omega
9.1	Three Player Game			Game Started
9.2	p2_start		simulateVolley	Player 2 Served
9.3				Player 2 Served
9.4	p2_WinsVolley	p2_Score = = 20		Player 2 Won
9.5	p2_IsWinner		return TRUE	Player 2 Won
10.1	Three Player Game			Game Started
10.2	p2_start		simulateVolley	Player 2 Served
10.3	p2_WinsVolley		simulateVolley	Player 1 Served
11.1	Three Player Game			Player 3 Served
11.2	p3_start		simulateVolley	Player 3 Served
11.4	p3_WinsVolley	p3_Score = = 19	simulateVolley	Player 3 Served
12.1	Three Player Game			Game Started
12.2	p3_start		simulateVolley	Player 3 Served
12.3				Player 3 Served

(Continued)

TCID	Test case input		Expected result	
	Event	Test condition	Active	State
12.4	p3_WinsVolley	p3_Score = = 20		Player 3 Won
12.5	dtor			omega
13.1	Three Player Game			Game Started
13.2	p3_start		simulateVolley	Player 3 Served
13.3				Player 3 Served
13.4	p3_WinsVolley	p3_Score = = 20		Player 3 Won
13.5	p3_IsWinner		return TRUE	Player 3 Won
14.1	Three Player Game			Game Started
14.2	p3_start		simulateVolley	Player 3 Served
14.3	p2_WinsVolley		simulateVolley	Player 2 Served
15.1	Three Player Game			Game Started
15.2	p3_start		simulateVolley	Player 3 Served
15.3	p1_WinsVolley		simulateVolley	Player 1 Served

EXAMPLE 13.8. For a three player game, what sneak path test suite will you form? Derive the test cases.

SOLUTION.

TABLE 13.2 Sneak Path Test Suite for Three Player Game.

TCID	Test case		Expected result		
	Setup sequence	Test state	Test event	Code	Action
16.0	Three Player Game	Game Started	Three Player Game	6	Abend
17.0	Three Player Game	Game Started	p1_WinsVolley	4	Illegal Event Exception
18.0	Three Player Game	Game Started	p2_WinsVolley	4	Illegal Event Exception
19.0	Three Player Game	Game Started	p3_WinsVolley	4	Illegal Event Exception

(Continued)

TCID	Test case		Expected result		
	Setup sequence	Test state	Test event	Code	Action
20.0	10.0	Player 1 Served	Three Player Game	6	Abend
21.0	5.0	Player 1 Served	p1_start	4	Illegal Event Exception
22.0	10.0	Player 1 Served	p2_start	4	Illegal Event Exception
23.0	5.0	Player 1 Served	p3_start	4	Illegal Event Exception
24.0	1.0	Player 2 Served	Three Player Game	6	Abend
25.0	6.0	Player 2 Served	p1_start	4	Illegal Event Exception
26.0	1.0	Player 2 Served	p2_start	4	Illegal Event Exception
27.0	6.0	Player 2 Served	p3_start	4	Illegal Event Exception
28.0	7.0	Player 3 Served	Three Player Game	6	Abend
29.0	2.0	Player 3 Served	p1_start	4	Illegal Event Exception
30.0	7.0	Player 1 Served	p2_start	4	Illegal Event Exception
31.0	2.0	Player 2 Served	p3_start	4	Illegal Event Exception
32.0	4.0	Player 1 Won	Three Player Game	6	Abend
33.0	4.0	Player 1 Won	p1_start	4	Illegal Event Exception
34.0	4.0	Player 1 Won	p2_start	4	Illegal Event Exception
35.0	4.0	Player 1 Won	p3_start	4	Illegal Event Exception
36.0	4.0	Player 1 Won	p1_WinsVolley	4	Illegal Event Exception

(Continued)

TCID	Test case		Expected result		
	Setup sequence	Test state	Test event	Code	Action
37.0	4.0	Player 1 Won	p2_WinsVolley	4	Illegal Event Exception
38.0	4.0	Player 1 Won	p3_WinsVolley	4	Illegal Event Exception
39.0	9.0	Player 2 Won	Three Player Game	4	Illegal Event Exception
40.0	9.0	Player 1 Won	p1_start	4	Illegal Event Exception
41.0	9.0	Player	p2_start	4	Illegal Event Exception
42.0	9.0	Player	p3_start	4	Illegal Event Exception
43.0	9.0	Player 2 Won	p1_WinsVolley	4	Illegal Event Exception
44.0	9.0	Player 2 Won	p2_WinsVolley	4	Illegal Event Exception
45.0	9.0	Player 2 Won	p3_WinsVolley	4	Illegal Event Exception
46.0	13.0	Player 3 Won	Three Player Game	6	Abend
47.0	13.0	Player 3 Won	p1_start	4	Illegal Event Exception
48.0	13.0	Player 3 Won	p2_start	4	Illegal Event Exception
49.0	13.0	Player 3 Won	p3_start	4	Illegal Event Exception
50.0	13.0	Player 3 Won	p1_WinsVolley	4	Illegal Event Exception
51.0	13.0	Player 3 Won	p2_WinsVolley	4	Illegal Event Exception
52.0	13.0	Player 3 Won	p3_WinsVolley	4	Illegal Event Exception
53.0	12.0	omega	any	6	Abend

13.2. TESTING OBJECT-ORIENTED SYSTEMS

Conventional test case designs are based on the process they are to test and its inputs and outputs. Object-oriented test cases need to concentrate on the state of a class. To examine the different states, the cases have to follow the appropriate sequence of operations in the class. Class becomes the main target of 00 testing. Operations of a class can be tested using the conventional white-box methods and techniques (basis path, loop, data flow) but there is some notion to apply these at the class level instead.

13.2.1. IMPLEMENTATION-BASED CLASS TESTING/WHITE-BOX OR STRUCTURAL TESTING

Implementation-based tests are developed by analyzing how a class meets its responsibilities. Implementation-based tests are developed by analyzing source code. This is also called structural, white-box, clear-box, or glass-box testing.

The main approaches to implementation-based testing are:

- Control flow testing
- Path coverage
- Data flow testing

These techniques are further classified as follows:

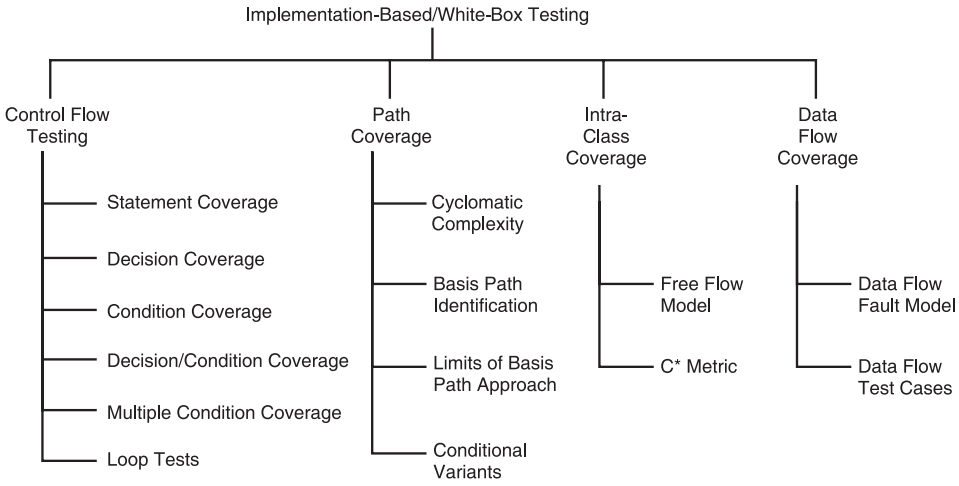


FIGURE 13.21 Types of White-Box Testing Techniques.

We will discuss these techniques one by one.

(I) Control Flow Testing

What Is Coverage Coverage is a key concept. It is a measure of test completeness. It is denoted by Cx. It is the percent of test cases identified by technique x that were actually run. So,

$$Cx = \frac{\text{Total Components Tested}}{\text{Total Number of Components}} \times 100$$

Following are some basis coverage metrics:

C_1 or Statement Coverage

- 100 percent statement coverage means every statement has been executed by a test at least once.
- C1 coverage (i.e., 100% statement coverage) is the minimum required by IEEE standard 1008, the standard for software unit testing. It has been IBM's corporate standard for nearly 30 years.
- C1 coverage is the absolute minimum for responsible testing. It is the *weakest* possible coverage.

C_2 or Decision Coverage

- 100% decision coverage means that every path from a predicate has been executed at least once by a test.
- For structured programs, C1 and C2 are equivalent. That is, you will cover all statements in a structured program if you cover all branches.

Control Graphs: Control flow faults account for 24% of all conventional software faults. For example,

Correct code	**Fault**
a = myobject.get ( )	a = myobject.get ( )
if (a = = b)	if (a = b) .....

Structured programming and small methods decrease the opportunity for these kinds of faults but do not eliminate them. Implementation-based testing can reveal subtle faults in control flow, and provides a basis for coverage analysis.

Consider the following C++ code snippet:

```
1 Void abx (int a, int b, int x)
  {
  if (a > 1) && (b = = 0)
2 x = x/a;
```

```
3 if (a = = 2) | | (x > 1)
4 x = x + 1;
5 cout < < x;
  }
```

We draw its control graph as it is our main tool for test case identification shown in Figure 13.22.

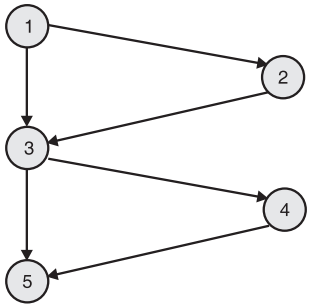


FIGURE 13.22 Flowgraph for C++ Code.

Statement coverage for this C++ code: It simply requires that a test suite cause every statement to be executed at least once.

We can get 100% C_1 coverage for the abx method with one test case.

Test case	Path	Test values			
		a	b	x	x′
SI.1	1-2-3-4-5	2	0	2	2

C_1 is not sufficient to discover common errors in logic and iteration control.

- Suppose the condition in node-3 is incorrect and should have seen

$$x > 0$$

There are many C_1 test sets that would miss this error.

- Suppose path 1-3-5 has a fault (x is not touched on this path). There are many C_1 test sets that would miss this fault.

Statement coverage is a very weak criterion. However, on average, adhoc testing covers only half of the statements.

Predicate Testing

A predicate is the condition in a control statement: if, case, do while, do until, or for. The evaluation of a predicate selects a segment of code to be executed.

There are four levels of predicate average [Myers]:

- Decision coverage
- Condition coverage
- Decision/condition coverage
- Multiple condition coverage

Each of these subsumes C_1 and provides greater fault detecting power. There are some situations where predicate coverage does not subsume statement coverage:

- Methods with no decisions.
- Methods with built-in exception handlers, for example, C++ try/throw/catch.

Decision Coverage

We can improve on statement coverage by requiring that each decision branch be taken at least once (at least one true and one false evaluation). Either of the test suites below provides decision (C_2) coverage for the abx ( ) method.

TABLE 13.3 Test Suite for Decision Coverage of abx ( ) Method (function).

Test suite	Test case	Path	Test values			
			a	b	x	x′
TS1	D1.1	1-2-3-4-5	4	0	8	3
	D1.2	1-3-5	0	0	0	0
TS2	D2.1	1-2-3-5	3	0	3	1
	D2.2	1-3-4-5	2	1	1	2

However, C_2 is not sufficient to discover common errors in logic and iteration control. For example, we can meet C_2 without testing the path where x is untouched (1-3-5) as test suite D_2 shows. Suppose condition-4 in statement number 3 is incorrectly typed as $x < 1$ then neither of the C_2 test suites would detect this fault.

Decision Coverage

It does not require testing all possible outcomes of each condition. It improves on this by requiring that each condition be evaluated as true or false at least once. There are four conditions in the abx ( ) method.

Condition	True domain	False domain
$a > 1$	$a > 1$	$a \leq 1$
$b = 0$	$b = 0$	$b \neq 0$
$a = 2$	$a = 2$	$a \neq$
$x > 1$	$x > 1$	$x \leq 1$

Either of the following test suites will force at least one evaluation of every condition. They are given below:

Test suite	Test case	Path	Test values			
			a	b	x	x′
TS1	C1.1	1-2-3-4-5	2	0	4	3
	C1.2	1-3-5	1	1	1	1
TS2	C2.1	1-3-4-5	1	0	3	4
	C2.2	1-3-4-5	2	1	1	2

However, this is not sufficient to discover common errors in logic and iteration control. For example, it is possible to skip a branch and still cover the conditions. This is shown by test suite C_2. Because condition coverage can miss nodes, it can miss faults.

Decision/Condition Coverage

Condition coverage does not require testing all possible branches. Decision/condition coverage could improve on this by requiring that each condition be evaluated as true or false at least once and each branch be taken at least once.

However, this may be infeasible due to a short-circuit Boolean evaluation.

Most programming languages evaluate compound predicates from left to right, branching as soon as a sufficient Boolean result is obtained. This allows statements like

```
if (a = 0) or (b/a > c) then .....
```

to handle a = 0 without causing a "divide-by-zero" exception. This can prevent execution of compound conditions.

We are now in a position to have decision/condition coverage for the abx ( ) method with two test cases.

Test case	Path	Test values			
		a	b	x	x′
DC1.1	1-2-3-4-5	2	0	4	3
DC1.2	1-3-5	1	1	1	1

But due to short-circuit evaluation, the false domains of

`b ≠ 0 and x ≤ 1 are not checked.`

Multiple Condition Coverage

The preceding problems occur because we have not exercised all possible outcomes of each condition for each decision.

Multiple condition coverage requires test cases that will force all combinations of simple conditions to be evaluated. If there are n simple conditions, there will be, at most, 2^n combinations. Multiple condition coverage subsumes all lower levels of predicate coverage.

We now derive the test cases based on multiple condition coverage for the abx ( ) method. They are given below.

TABLE 13.4 Test Cases for abx ( ) Method/Function.

Test case	Path	Test values			
		a	b	x	x′
M1.1	1-2-3-4-5	2	0	4	3
M1.2	1-3-4-5	2	1	1	2
M1.3	1-3-4-5	1	0	2	3
M1.4	1-3-5	1	1	1	1

Please note the following observations:

a. We usually don't need 2^n test cases for multiple condition coverage.

b. Multiple condition coverage does not guarantee path coverage. For example, path 1-2-3-5 is not exercised.

c. If a variable is changed and then used in a predicate, reverse data flow tracing may be needed to establish test cases values. This is called *path sensitization*.

The following table shows how each condition is covered by the M test suite (for the abx ( ) method).

a	b	x	Test Case
> 1	= 0	dc	M1.1
	≠ 0	dc	M1.2
≤ 1	= 0	dc	M1.3
	Impossible due to short circuit		
	≠ 0	dc	M1.4
	Impossible due to short circuit		
= 2	dc	> 1	M1.1
	dc	≤ 1	M1.2
≠ 2	dc	> 1	M1.3
	dc	≤ 1	M1.4

So, there are 8 possible conditional variants. And we are able to exercise all 8 with only 4 test cases.

Impossible Conditions and Paths

Some predicate domains or programming structures may render a path impossible. There are at least four causes of impossible conditions and paths.

1. **Boolean short circuits,** as discussed earlier.
2. **Contradictory or constraining domains.**

   Given:

   ```
   if (x > 2) && (x < 10) then ....
   ```

   Only three conditions can be obtained. x cannot be simultaneously less than 3 and greater than 9, so the false-false case is impossible.

3. **Redundant Predicates.**

   ```
   if (x)      <A>
   else        <B>
   endif
   if not (x)  <C>
   else        <D>
   endif
   ```

For the same value of x, paths <A> <C> and <B> <D> are not possible.

Because the predicates could be merged, you may have found a fault or at least a questionable piece of code.

4. **Unstructured Code.**
   - Acceptable: Exception handling, break.
   - Not acceptable: Anything else.

Consideration for Loops

Loops also pose some unique test problems. Loops may be simple loops, nested loops, serial loops, and spaghetti loops. We shall discuss each one by one.

a. **Simple loops:** All loops should get at least two iterations. This is the minimum needed to detect data initialization/use faults. We focus on boundary conditions because they are a frequent source of loop control faults.

   A simple loop is either a do, while, or repeat-until form with a single entry and a single exit. A complete test suite will exercise the domain boundary of the loop control variable such as (minimum-1), minimum (possibly zero), (minimum +1), typical, (maximum –1), maximum, (maximum +1), and so on.

b. **Nested loops:** A nested loop is a simple loop contained in another simple loop. Multiplicative testing is overkill. For example, if we use the five basic loop test cases then:

   i. We would have 25 test cases for a doubly nested loop.

   ii. We would have 125 test cases for a triply nested loop.

   Beizer suggests that we should start with the innermost loop first and then go to the outer loops to test nested loops.

   Beizer suggests the following procedure:

   Step 1. Test the inner loop first, and the outer loop last.
   Step 2. Run the five basic tests on the inner loop:

   a. Min, min +1, typical, max –1, max

   b. Set all outer loop controls to minimum values

   c. Add excluded and out of range checks, if needed

Step 3. For the next loop out:

- Run min, min +1, typical, max –1, max.
- Set the inner and outer loop controls to typical values.
- Add excluded and out of range checks, if needed.

Step 4. Repeat Step-3 for each inner loop.
Step 5. When all loops check out individually, set all values at max and test all loops together. With two-level loops, this requires 12 test cases, 16 with three-level, and 19 with four-level.

c. **Serial loops:** Two or more loops on the same control path are serial loops. To test them

- If there are any define/use data relationships between the loops, treat them as if they were nested.
- Use data flow analysis to determine test cases.
- If there is no define/use relationship between the loops, they can be tested in isolation.

d. **Spaghetti loops:** It is a loop which has more than one entry or exit point. Such loops, if nested, can result in very complex control flow paths.

To test them, we can follow one of the following methods:

a. Reject such loops and return the code to the developer for correction.

b. Design another equivalent single-entry/single-exit loop. Develop your test cases from this design, then add test data for every wired path you can think of.

(II) Path Coverage Testing

The number of unique independent paths through a non directed graph is given by graph theory. The number of independent paths is called the cyclomatic complexity of the graph, denoted by V(G). They are called basis paths because they correspond to the basis vector in a matrix representation of an undirected graph.

This number is not defined for directed (e.g., control flow) graphs. However, it has become a popular metric for the complexity and test coverage.

There are three methods to compute V(G). They are as follows:

a. V(G) = e – n + 2 (e = edges, n = nodes)

b. V(G) = Number of enclosed regions +1

c. V(G) = Number of predicate nodes +1

These expressions hold for single-entry, single-exit program complexity, C.

Basis Paths. Consider the following function foo ( ).

```
1  Void foo (float y, float a*, int n)
         {
         float x = sin (y);
         if (x > 0.01)
2        z = tan (x);
         else
3             z = cos (x);
4  for (int i = 0; i < x ; ++i) {
6           a[i] = a[i] * z;
        cout << a[i];
        }
7       cout << i;
        }
```

Its flowgraph is shown in Figure 13.23.

Here, Number of edges (e) = 8
Number of nodes (n) = 7

$\therefore$ $V(G) = e - n + 2 = 8 - 7 + 2 = 3.$

This indicates there are at least 3 entry/exit paths in this graph:

Path 1: 1-2-4-5-7
Path 2: 1-3-4-5-7
Path 3: 1-3-4-5-6-7

or they can be:

Path 1: 1-2-4-5-7
Path 2: 1-3-4-5-7
Path 3: 1-2-4-5-6-7

These are called basis paths.

This technique has already been discussed in previous chapters.

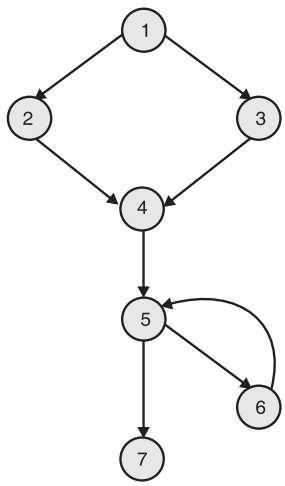


FIGURE 13.23

(III) Data Flow Testing

We have already discussed this technique in earlier chapters.

(IV) Intra Class Coverage

It further involves two techniques:

a. The FREE flow model

b. The C* metric (or V*(G) metric).

We shall discuss these techniques one by one.

a. **The FREE flow graph:** We have seen several ways to identify paths for single-entry, single-exit code. However, a *class* typically contains many single-entry, single-exit (SESE) code segments. In C++, each member function is a SESE segment. The methods in a class access the same instance variables and must cooperate for the correct execution under all possible activation sequences. What is the control and data flow for the entire class? To answer such questions, we need:

a. A state model for the class under test.

b. A control flow graph for each method.

We will use the FREE state model.

- State is the result of method activation.
- A single state may be computed by one or several methods.
- We use a concrete, aggregate model of state.

Events are either:

- Method activation.
- Interrupts that change the state of an instance variable, i.e., we are testing an actor class.

Action are either:

- Message response (values returned to the client).
- Messages sent to servers or other interfaces.

The FREE flow graph is based on the following observation:

- Methods compute state.
- A single transition edge may be replaced by the flow graph of the method which causes the transition.

A FREE flow graph is constructed in two steps:

1. Method graphs are substituted for transitions.

2. Edges are added to link the state exit/method entry and method exit/state entry nodes.

Test case derivation: We now have a graph model of the entire class. We can identify all intra-class control paths. We can identify all intra-class du paths for instance variables. We can apply all the preceding test case techniques at the class level.

b. **The C* metric:** We know that V(G) or C of a graph, G, is given by e – n + 2. Similarly, for the FREE flow graph the class complexity is represented by C* or V*(G). It is the minimum number of intra-class control paths.

$$\therefore \quad C^* = E - N + 2$$
$$E = e_m + 2m$$
$$N = n_m + n_s$$

where e_m = Total edges in all inserted subgraphs
m = Number of inserted subgraphs
n_m = Total nodes in all inserted subgraphs
n_s = Nodes in state graph (states)

Thus, we have

$$C^* = e_m + 2m - n_m - n_s + 2$$

Limitations of Implementation-Based Testing/White-Box Testing of Classes

It provides many useful approaches to develop and evaluate unit test cases. The limitations are as follows:

1. Exhaustive path testing is impossible.
2. Implementation-based testing requires code analysis skills and knowledge of test case design techniques.
3. Without automated support, it is hard to scale up.
4. Available testing tools are useful but not ideal.
5. Coverage metrics are guidelines not absolutes.
6. Even a high coverage test suite cannot prove the absence of faults, validate requirements, or test missing functions.
7. Each code-based coverage approach has "blind spots" and is sensitive to the particular test values selected.
8. Code-based testing is tautological.

13.2.2. RESPONSIBILITY-BASED CLASS TESTING/BLACK-BOX/FUNCTIONAL SPECIFICATION-BASED TESTING OF CLASSES

Responsibility-based class testing is the process of devising tests to see that a class correctly fulfills its responsibilities. These tests are derived from specifications and requirements. They are not derived from implementation details. This is also called black-box, functional, or specification-based testing.

We categorize this type of testing as follows:

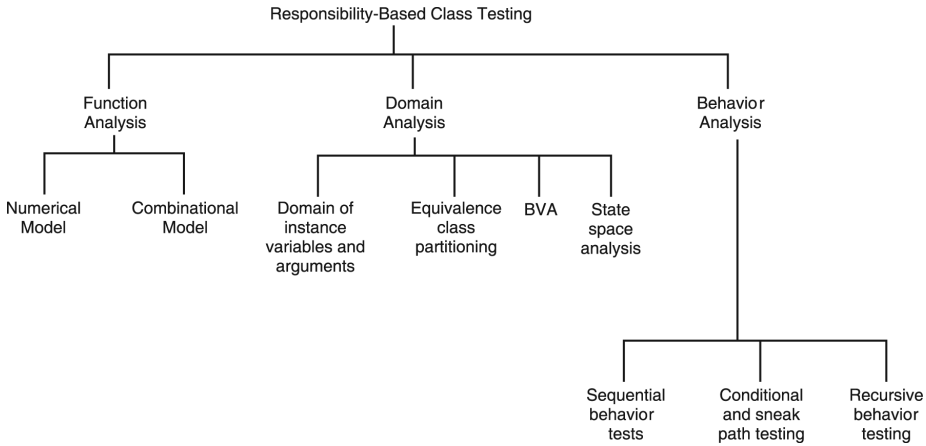


FIGURE 13.24

We need to consider three main facets of a class and its methods to develop responsibility test cases. They are:

i. **Functional Analysis**

- What kind of function is used to transform method inputs into outputs?
- Are inputs externally or internally determined?

ii. **Domain Analysis**

- What are valid and invalid message inputs, states, and message outputs?
- What values should we select for test cases?

iii. **Behavior Analysis**

- Does the sequence of method activation matter or not?
- When sequence matters, how can we select efficient testing sequence?

We will discuss each of these one by one.

Overview of the Approach

Responsibility-based testing has four main steps:

Step 1. Select a functional model for each method: numerical, combinational, or compound.

a. Model numerical functions with an equation.

b. Model combinatorial functions with a decision table.

c. Develop heuristic tests for compound functions.

Step 2. Identify test case data for each method test case using domain analysis.

Step 3. Select an activation model for the method and the class. Develop an sequential activation plan.

Step 4. Interleave functional test cases with the activation plan.

Function types. A method computes a result. In this sense, it is like a mathematical function. For testing purposes, it is useful to consider three kinds of functions. They are as follows:

Numerical: A numerical function is specified or can be modeled by a mathematical formula.

Combinational: A combinational function is specified or can be modeled by a decision table, Boolean equations, or equivalent conditions, rules or logic.

General: A general function has responsibilities that are not easily modeled as numerical or combinational functions.

A Testable Function

- Must be independently invocable and observable.
- Is typically a single responsibility and the collaborations necessary to carry it out.
- Often corresponds to a specific user command or menu action.

Testable functions should be organized into a hierarchy

- Follow the existing design.
- Develop a test function hierarchy.
- Testable functions should be small, "atomic" units of work.

A method is a typically testable function for a class-level test. A use case is a typically testable function for a cluster-level or system test.

For numerical functions, how many test cases do we need?

- At least two.
- If there are n variables, we need (n + 1) test cases.

What values should we use?

- Each value in the test suite must be unique.
- Do not use 1 (1.0) or 0 (0.0) for input values.
- Select values that are expected to produce a non-zero result.
- If a variable is returned by a message, the message must be forced to provide a non-result.

This test will reveal all errors in which one or more coefficients is incorrect.

Test cases are defined in *two steps:*

Step 1. Prepare a matrix with one more column than there are variables. This extra column is filled with "1," which is used to check the determinant but not as part of the test case.

	Variable 1	Variable 2	.....	Variable n	
Test Case 1					1
Test Case 2					1
… …					1
Test Case n + 1					1

Step 2. Find values for the variable matrix such that the determinant of the entire matrix is not zero. This requires that

- No row or column consists entirely of zeros.
- No row or column is identical.
- No row or column is an exact multiple of any other.

So, for F = (10 * x) – (x + y) ^ 3, we have

Test case	x	y	Expected result
1	3	7	–970
2	5	11	–4046
3(n + 1)	7	13	–7930

To select test values for variables

- Use prime numbers for integer variables.
- For rational-valued variables, select values which meet the above matrix constraints by inspection. If the matrix is too large to check by inspection, then compute the determinant.
- If the variables are bounded (e.g., $100 < x < 1000$) then develop additional tests for boundary values.

Alternatively, we can select output values and solve for the test points. We must:

- Test cases for inequalities in n-variables and should have at least (n + 2) tests. The function should be exercised at least one on-point and one off-point.
- Test cases for non-linear functions should have points falling on either side of the points of inflexion.
- Boolean functions (i.e., variables with a {0/1} or {true/false} domain should be validated by combinational analysis).
- The expected results must be established by an oracle or a trustworthy source of correct values.
- Inspections and acceptance testing are the only way to detect design errors.

What About the Combinational Functions?

A combinational function is one where:

a. Output is determined by current input.

b. Previous inputs do not change the output.

c. This is like a simple spin lock which will open when the correct number is dialed in regardless of the dialing order.

We model combinational functions with a decision table. A decision table has conditions and actions. When all of the individual conditions on one line are true, the corresponding action is taken. There can be many conditions for each action. A condition must resolve to a Boolean variable.

Cells irrelevant for a rule are labelled DC, for "don't care." Rules can be given in either rows or columns.

For example, we have already drawn a decision table for some problems in previous chapters. Now, let us see how to find combinational faults.

Although combinational functions should ignore prior inputs and give the same results regardless of input sequence, try varying the order of the

input variables. The evaluation of the conditions is typically hard-coded in a case or switch construct which may have assumed a particular order of input.

A decision table with n conditions can have 2^n actions, however ....

- It is not unusual to see fewer than 2^n actions.
- This often results from "don't care" (DC) or "impossible" conditions.
- DCs should be implemented by default processing by which they are treated as an error or ignored without causing a failure.
- As a practical matter, DCs often result in failures.

So, we must not ignore DCs. We must expand the DC conditions to the full 2^n set. We test each of this expanded condition in the same way we would test regular conditions.

(II) Domain Analysis

Test case data is identified by domain analysis. For each testable function, we must:

a. Identify all parameters, arguments, variables, database conditions, set points, toggles, etc.

b. Identify equivalence classes or partitions for each variables of interest.

c. Identify boundary values for the partition.

Domain analysis can reveal design faults. It is good at revealing missing logic faults—holes in the specification or the implementation.

Domains–Input v/s Output

A domain is a set defined over a method's arguments or instance variables. A typical input domain is astronomically large. For a method that accepts a 10 character (8-bit) input string, there are

$$2^8 \times 2^8 \times 2^8 \times 2^8 \times 2^8 \times 2^8 \times 2^8 \times 2^8 \times 2^8 \times 2^8 = 2^{80}$$

possible input combinations.

Even if you were very fast and could run and check 1000 tests per second and work 24 hours a day, non stop, you would need about twice the current estimated age of the universe to complete the tests. Remember that exhaustive testing is impossible.

Because we can't test all values, which ones should we test? The answer may be an equivalence class test. But before discussing these techniques we must know the differences between the input and output domain.

An *input domain* and domain values are relatively easy to define. So, we have

a. Any input domain is the entire range of valid values for all external and internal inputs to the method under test.

b. Private instance variables should be treated as input variables.

c. The domain must be modeled as a combinational function if there are dependencies among several input variables.

An *output domain* may be more difficult to identify. So, we have

a. The output domain of an arithmetic function is the entire range of values that can be computed by the function.

b. Output domains can be discontinuous–they may have "holes" or "cliffs."

c. The output domain of a combination function is the entire range of values for each action stub.

Next, we discuss a technique under domain analysis which is a type of responsibility-based/ black-box testing method. This technique is popularly known as *equivalence class partitioning.*

An equivalence class is a group of input values which require the same response. A test case is written for members of each group and for non-members. The test case designates the expected response.

NOTE *When speaking of an equivalence class, class is used in the mathematical sense and not as a source code construct.*

Equivalence class testing is based on the assumption that if a method fails for one member of a equivalence class, it is likely to fail for all members of that class. This reduces the size of the input space to manageable proportions.

How are Equivalence Classes Identified?

We have already discussed this issue in previous chapters. System requirements are the source of domains, equivalence classes, and boundary values. There is no algorithm for exactly determining "good" and "bad" partitions. Any available information can be used. As quoted by Hamlet—

"The goal is to make the resulting classes so narrow that each aspect of the program, of the specification, of development, each programmer concern etc. is separated into a unique class."

The steps are:

1. Define valid and invalid partitions for each input variable.
2. Further subdivide the valid partition.

At least one separately executable test case is written for each equivalence class. The second technique is known as boundary value analysis (BVA). As with equivalence classes, there is no general algorithm for exactly determining boundary values. The domain boundary depends on what the variable represents and its constraints.

For a range of (n, m), the boundaries may lie

- Inside upper and lower edges, n, n + 1, m – 1, m.
- Outside upper and lower edges, n – 1, m + 1.

For counters, boundaries are often related to capacity limits. Say, we can store from 1 to 255 objects.

- We would have boundary values at 0, 1, 255, and 256.

For ordered collections, boundaries occur at the start, end, etc. Like first, last, empty, full top, bottom, etc.

For output variables, boundaries can suggest some tests.

- Try tests that produce exactly the minimum and maximum output values.
- Try tests that exceed the minimum and maximum output values by the smallest increment permitted by the data type.

It is the type of function that suggests boundary values.

For a numeral with floating-point arguments

- Try test cases that should cause over flow and under flow.
- Try tests to check the finest resolution. For example, with floating-point operations, [(1.0/ 10.0) * 10.0] never really equals 1.

For combinational functions, we identify all combinations of on-points. A test case may be prepared for each on-point set.

Another technique is state space analysis. The number of corners in a state space depends upon the state variable constraints. A state space boundary may be defined by:

- Complex but well-formed mathematical relations (sphere, torus, polyhedra, etc.).
- Arbitrarily complex and irregular constraints.

- Research which indicates that a test set consisting of (n + 2) points with least one off-point and one on-point is needed to probe a non linear boundary.

To probe a state space, we adapt the nested loop approach to select points. With two state variables this produces 15 test points, 22 for three, and 29 with four.

(III) Behavior Analysis

Behavior is a specified or observed sequence of accepted messages and responses. There are three behavior patterns of interest for testing.

a. **Independent:** The method output is totally independent of previous activation. We can use domain, numerical, or combinational techniques.

b. **Sequential:** The method output depends on specific prior sequences of activations and can be modeled by a finite state machine.

c. **Recursive:** Method output depends on previous activations but there are no constraints on order of activation.

It is of three types:

a. Sequential behavior testing

b. Conditional and sneak path testing

c. Recursive behavior testing

Now, we shall discuss these techniques one by one.

a. **Sequential behavior testing:** To develop test cases for sequential behavior:

- Develop a state model.
- Messages are modeled as transition events.
- Outbound messages are modeled transition actions.
- Generate the state path tree.
- Tabulate events and actions along each path to form test cases.
- Develop test data for the path by domain analysis on events and actions.

Consider a simple state model of a STACK.

What kinds of sequential behavior faults can occur?

The faults that can occur are:

a. Missing or incorrect transition.

b. Missing or incorrect event.

c. Missing or incorrect action.

d. Extra, missing, or corrupt state.

We have already discussed this type of testing in Section 13.1 of this chapter. In the table below we list various state control faults for the 2 player game already discussed.

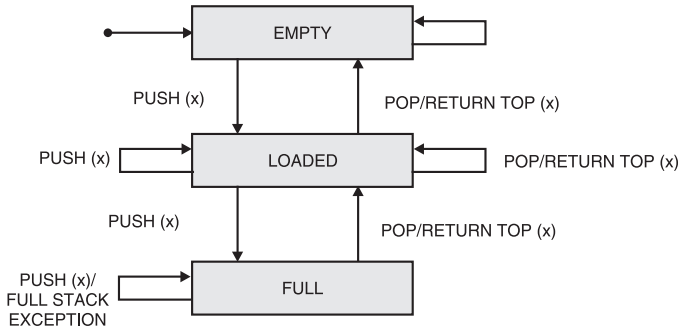


FIGURE 13.25

Event	Action	Resultant state	Error description
OK	OK	OK Wrong Extra	Normal Transition Incorrect State Corrupted State
Reject Legal	OK	OK Wrong Extra	Missing Transition
Accept Illegal	Wrong	OK Wrong Extra	Sneak Path Sneak path to corrupt state
Accept Undefined	Undefined	OK Wrong Extra	Trap door with incorrect output. Trap door with incorrect output to corrupt state

b. **Conditional transition tests:** With conditional transitions, we need to be sure that all conditional transition outcomes have been exercised.

- During conformance testing, we want a conditional transition to fire when we send a message which meets its condition.
- For each conditional transition, we need to develop a truth table for the variables in the condition.
- An additional test case is developed for each entry in the truth table which is not exercised by the conformance tests.
- This test is added to the transition tree.

Event Path Sensitization

- Values (or states) necessary for conditional expression must be determined by analysis.
- We can use the same approach taken to identify path conditions for source code.

But this testing is not complete. We also need to do *sneak path testing*.

What Is a Sneak Path?

- An *illegal transition* is present when the class-under-test (CUT), in some valid state, accepts a message which is not explicitly specified for that state.
- An *illegal message* is an otherwise valid message which should not be accepted given the current state of the class; if accepted, an illegal transition results.
- A *sneak path* is the bug in the CUT which allows an illegal transition.

We test for sneak paths by sending illegal messages. For each specified state, a sneak path is possible for each message *not* accepted in that state.

The expected response to a sneak path is usually:

- The message should be rejected in an appropriate manner.
- The state of the object should be unchanged after rejecting the illegal event.

Strict black-box state-based testing approaches assume that only externally visible I/O sequences are available to the tester. The tester must determine which state is actually obtained by applying a distinguishing sequence and observing the resulting output. This increases the number of tests by a large amount.

With object-oriented software, we assume an internal state can be determined by

a. A method activation trace (activation is equated with state).

b. State reporting capability in the CUT.

c. Built-in reporting in the CUT.

This is a gray-box assumption.

And last but not the least, another testing technique is recursive behavior testing. We will discuss it following.

c. Recursive behavior testing: Some classes are purposely designed to respond to all methods in any state. This is often true of "framework" or "foundation" classes. Classes that represent the application domain typically have sequential constraints.

The recursive behavior test strategy uses paired sequences of methods on instances of the same class [Doong]. In this type of testing—

a. Each sequence is picked to yield the same results.

b. Each member of the pair is executed.

c. The results are compared.

d. If the results are not equivalent, the test has succeeded in revealing a fault.

For example, in the table below, test cases 1 and 2 run a pair of sequences on stack objects A and B, which should yield the same result in all four sequences, that is, 5 on the top. So, we have

Test	Test sequence
1. Stack A	Create, push (5), push (6), pop
Stack B	Create, push (5)
2. Stack A	Create, push (5), push (6), pop, top
Stack B	Create, push (5), top

13.3. HEURISTICS FOR CLASS TESTING

An *abstract class* is typically not directly instantiated. Instead, subclasses specialize or provide an implementation. A strategy for testing C++ *abstract classes* can be given below:

1. Develop specification-based tests from abstract class specification.
2. Test all member functions that don't call pure virtual methods.
3. Test all member functions that do call pure virtual methods. This may require stubs for the derived class implementation.
4. Do state/sequential tests of new and inherited methods.

Like abstract classes, the templates cannot be directly instantiated or tested. Once instantiated, all other basis testing techniques apply. The strategy for testing template classes is given below:

1. Prepare test cases, and test at least one instantiation.
   - A single test suite can be reused to test each type with the necessary changes per type and additional special tests per type.
   - Does any possible type suggest special or different test cases?
   - Do any pre- or post-conditions change due to any possible type?
2. Select a type that exercises all class features. If you can't then this is probably a poorly designed template.
3. Decide on how these additional types may be tested. Testing one type may be sufficient if
   - Only =, = =, and ! = operators are used.
   - There are no parameters of T* or T& in the public interface of the template class.
   - This means no polymorphic bindings can be made.

If either of the above conditions is not present, then each possible type should be instantiated and tested.

A *collection class* houses a group of individual objects of the same type, for example, a list, stack, queue, array, table, etc. The responsibility of this class is to manage the collection according to some scheme. The test strategy must be based on this scheme.

There are several patterns useful for testing collections. They are discussed below.

Sequential Collection Test Pattern

A sequential collection contains a variable number of items. Items are added to the "end," e.g., strings, buffers, ASCII files, etc. You must "walk" the entire collection in sequence to access a particular element. For a sequential collection like a stack or queue, we have the test suite given below.

Collection operation	Input size	Collection state	Expected result
1. Add	Single element	empty	added
	Single element	not empty	added
	Single element	capacity-1	added
	Single element	full	reject
	Several elements, sufficient to overflow	not empty	reject
	Several elements	capacity-1	reject
	Null element	empty	no action
	Null element	not empty	no action
2. Update/ Replace	Several elements, sufficient to overflow by 1	not empty	reject
	Several elements, sufficient to reach capacity	not empty	accept
	Several elements, sufficient to reach capacity	not empty	accept
	Several elements, sufficient to reach capacity-1	not empty	accept
	Several elements, fewer than in updated collection	not empty	accept, check clean-up

Ordered Collection Test Pattern

In an ordered collection, elements are added, accessed, and removed using some ordering scheme—stack, queue, tree, list, etc. All of the sequential test patterns can be applied with the necessary changes in expected results. The following test patterns verify position-dependent behavior.

Collection operation	Input	Element position	Collection state	Expected result
All operations	Single item Single item	First Last	Not empty Not empty	Added Added
Delete/ Remove	Single item Single item	dc dc	Single element Empty	Deleted Reject

Pairwise Operand Test Pattern

Operations may be defined that use collections as operands. Suppose there are two collections, A and B. Then,

- A and B should be individually verified first.
- A and B are populated before running the test.

The following test patterns verify size-dependent operation processing and not the semantics of the operation—

Collection operation	Size of A	Size of B	Expected result
All operations	0 (empty) 0 (empty) N	0 (empty) N M	Operation specific Operation specific Accept
All operations requiring same size A and B	N N N + 1	N N + 1 1	Accept Reject Reject

where $N > 1$ and $N \neq M$.

What About Arrays and Pointers?

Try the following as inputs to the pointer calculation

- A negative, zero, or very large value
- The lower-bound value –1
- The lower-bound value

- A normal value
- The upper-bound
- The upper-bound +1

Try formula verification tests with array elements initialized to unusual data patterns.

- All elements zero or null
- All elements one
- All elements same value
- All elements maximum value, all bits on, etc.
- All elements except one are zero, one, or max

We treat each sub-structure with a special role (header, pointer vector, etc.), as a separate data structure.

The pair-wise operand test pattern may also be applied to operators with array operands.

Relationship test patterns: Collection classes may implement relationships (mapping) between two or more classes. We have various ways of showing relationships. For example

Entity-relationship model:

FIGURE 13.26

Coad-Yourdon diagram for the above case is

FIGURE 13.27

While its Booch class diagram is

FIGURE 13.28

We can now model valid instances of relations with combinational analysis.

Person	Owns	Dog	Result
1	0	DC	Okay
	1	0	Error
		1	Okay
		*	Okay
		M	Okay

Relationships can be tested by considering likely scenarios:

- Person buys dog
- Dogs runs away
- Person sells dog
- Dog dies

We can use the relationship's cardinality parameters to systematically identify test cases that verify correct implementation of relationships.

Keyed Collection Test Patterns

Some collections (e.g., Dictionary in Smalltalk) provide a unique identifier for each member of a collection.

Application classes that provide file or database wrappers may also rely on keyed access. The basic test patterns for keyed collections follow.

Collection operation	Key value	Item state	Expected result
Create	Any	Not present	Accept
Read	Any	Not present	Reject
Update	Any	Not present	Reject
Delete	Any	Not present	Reject
Create	Any	Present	Reject
Read	Any	Present	Accept
Update	Any	Present	Accept
Delete	Any	Present	Accept

Keyed collections must provide a wide range of paired operations. Item-to-item processing is often faulty.

Collection classes are well-suited to sequential constraint or recursive equivalence test patterns on operations.

Exception Testing

Exception handling (like Ada exceptions, C++ try/throw/catch) add implicit paths. It is harder to write cases to activate these paths. However, exception handling is often crucial for reliable operations and should be tested. Test cases are needed to force exceptions.

- File errors (empty, overflow, missing)
- I/O errors (device not ready, parity check)
- Arithmetic over/under flows
- Memory allocation
- Task communication/creation

How can this testing be done?

1. **Use patches and breakpoints:** Zap the code or data to fake an error.
2. **Use selective compilation:** Insert exception-forcing code (e.g., divide-by-zero) under control of conditional assembly, macro definition, etc.
3. **Mistune:** Cut down the available storage, disk space, etc. to 10% of normal, for example, saturate the system with compute bound tasks. This can force resource related exceptions.
4. **Cripple:** Remove, rename, disable, delete, or unplug necessary resources.
5. **Pollute:** Selectively corrupt input data, files, or signals using a data zap tool.

Suspicion Testing

There are many situations that indicate additional testing may be valuable [Hamlet]. Some of those situations are given below:

1. A module written by the least experienced programmer.
2. A module with a high failure rate in either the field or in development.
3. A module that failed an inspection and needed big changes at the last minute.
4. A module that was subject to a late or large change order after most of the coding was done.
5. A module about which a designer or programmer feels uneasy.

These situations don't point to specific faults. They may mean more extensive testing is warranted. For example if n-tests are needed for branch coverage, use 5n instead to test.

Error Guessing

Experience, hunches, or educated guesses can suggest good test cases. There is no systematic procedure for guessing errors. According to Beizer, "logic errors and fuzzy thinking are inversely proportional to the probability of a path's execution."

For example, for a program that sorts a list, we could try:

- An empty input list.
- An input list with only one item.
- A list where all entries have the same value.
- A list that is already sorted.

We can try for weird paths:

- Try to find the most tortuous, longest, strongest path from entry to exit.
- Try "impossible" paths, and so on.

The idea is to find special cases that may have been overlooked by more systematic techniques.

Historical Analysis

Metrics from past projects or previous releases may suggest possible trouble spots. We have already noted that C metric was a good predictor of faults. Coupling and cohesion are also good fault predictors. Modules with high coupling and low cohesion are 7 times more likely to have defects compared to modules with low coupling and high cohesion.

Class Testing Summary

The main steps to test class methods are given below:

Step 1. Instantiate a test object of the class-under-test (CUT).
Step 2. Set the object-under-test (OUT) and supplier objects to the state specified by the test case.
Step 3. Set the parameters of the method-under-test (MUT) to the values specified by the test case.
Step 4. Send the test message to the MUT.
Step 5. Check for compliance with the expected results of the test case:

a. The returned values of the MUT.
b. The resultant state of the OUT.
c. The state of the message parameters.
d. Exceptions (if any) thrown or raised.
e. Message sent to other objects (or stubs) are correct in sequence and content.

Six Principles of Effective Testing

Brain Marick offers six principles of effective testing. They are as follows:

Principle 1: Most errors are not very creative. Because errors tend to follow patterns, methodical checklist testing has a high payoff.

Principle 2: Faults of omission activated by unanticipated special cases are the most difficult to find and should therefore get greater attention.

Principle 3: Specification faults, especially omissions, are more dangerous and harder to find than code faults.

Principle 4: At every stage of testing, mistakes are inevitable. Later stages should compensate for this.

Principle 5: Code coverage is a good approximation of test quality. Because it is an approximation, coverage should be viewed as a useful indicator, not a guarantee.

Principle 6: Good tests emphasize variety and complexity. So,

- Use the same test case patterns in different ways and in different contexts.
- Try to combine test cases in random, unusual, strange, or weird ways. This increases the chance of revealing a sneak path, a surprise, or an omission.

13.4. LEVELS OF OBJECT-ORIENTED TESTING

Three or four levels of object-oriented testing are used depending on the choice of what constitutes a unit. If individual operations or methods are considered to be units, we have four levels: operation/method, class, integration, and system testing.

1. With this choice, *operation/method testing* is identical to unit testing of procedural software.
2. *Class and integration testing* can be renamed as intraclass and interclass testing. The second level then, consists of testing interactions among previously tested operations/methods.
3. *Integration testing* is the major issue of object-oriented testing and must be concerned with testing interactions among previously tested classes.
4. *System testing* is conducted at the port event level and is identical to system testing of traditional software. The only difference is where system level test cases originate.

13.5. UNIT TESTING A CLASS

Classes are the building blocks for an entire object-oriented system. Just as the building blocks of a procedure-oriented system have to be unit tested individually before being put together, so the classes have to be unit tested. This is due to several reasons. Some of them are given below:

1. Reusability of the code is the focal point of the object-oriented system. A class is also reused many times. A residual defect in a class can, therefore, potentially affect every instance of reuse.
2. Many defects get introduced at the time a class gets defined. If these defects are not caught in time then they may go into the clients of these classes. Thus, the fix for the defect would have to be reflected in multiple places giving rise to inconsistencies.
3. A class may have different features. Different clients of the class may pick up different pieces of the class. No one single client may use all the pieces of the class. Thus, unless the class is tested as a unit first, there may be pieces of a class that may never get tested.
4. A class encapsulates data and functions. If the data and functions do not work in synchronization at a unit test level, it may cause defects that are potentially very difficult to narrow down later.
5. Because object-oriented systems support inheritance, the building blocks are thoroughly tested stand alone, defects arising out of these context may surface magnified many times later in the cycle.

The question is now whether we can apply conventional methods to test our classes? Yes, some of the methods for unit testing apply directly to testing classes. For example:

1. Every class has certain variables. The techniques of BVA and equivalence class partitioning discussed earlier in black-box testing can be applied to make sure that the most effective test data is used to find as many defects as possible.
2. Not all member-functions in C++ or methods in JAVA are exercised by all the clients. The methods of *function coverage* that were discussed in white-box testing can be used to ensure that every method (function) is exercised.
3. Every class will have functions/methods that have some procedural logic. So, the techniques of *condition coverage, branch coverage, code*

complexity, and so on that we discussed during white-box testing can be used to make sure that branch coverage is complete and to increase the software maintainability.

4. Because a class is meant to be instantiated multiple times by different clients, the various techniques of *stress testing and system and acceptance testing* can be performed for early detection of stress-related problems such as memory leaks.

Also note that we have already discussed *state-based testing* in Section 13.2.2 of this chapter. This technique is useful for black-box testing of classes. Because a class is a combination of data and methods that operate on the data, in some cases, it can be visualized as an object going through different states. The messages that are passed to the class act as inputs to trigger the state transition. It is useful to capture this view of a class during the design phase so that testing can be more natural.

How to Test Classes?

In order to test an instantiated object, messages have to be passed to various methods. In what sequence does one pass the messages to the objects? One of the methods that is effective for this purpose is the alpha-omega method.

Principles of the Alpha-Omega Method

The alpha-omega method works on the following principles:

1. Test the object through its life cycle from birth to death, that is, from instantiation to destruction. An instance gets instantiated by a constructor method; then the variables are initialized. These values may get modified also. Finally, destructors are used to destroy objects.
2. Test simple methods first and then more complex methods. This is because object-oriented programming languages support inheritance and thus more complex methods will be built upon the simpler methods.
3. Test the methods from private through public methods. The private access specifier reduces the dependencies in testing and gets the building blocks in a more robust state before they are used by clients.
4. Send a message to every method at least once. This ensures that every method is tested at least once.

The following steps are followed during the alpha-omega method:

Step 1. Test the constructor methods first. When multiple constructors are used, all should be tested individually.

Step 2. Test the get methods or access or methods (methods that retrieve the values of variables in an object for use by the calling programs). This ensures that the variables in the class definition are accessible by the appropriate methods.

Step 3. Test the methods that modify the object variables. There are methods that test the contents of variables, methods that set/update the contents of variables, and methods that loop through various variables.

Step 4. Finally, the object has to be destroyed and when the object is destroyed, no further accidental access should be possible. Also, all the resources used by the object instantiation should be released. These tests conclude the lifetime of an instantiated object.

Which Unit to Select—Method or Class?

Let us consider this one by one as two separate cases.

Case 1: Methods as units: Superficially, this choice reduces object-oriented unit testing to traditional (procedural) unit testing. A method is nearly equivalent to a procedure so all the traditional functional and structural testing techniques should apply. Unit testing of procedural code requires stubs and a driver test program to supply test cases and record the results. Similarly, if we consider methods as 0-0 units, we must provide stub classes that can be instantiated and a "main program" class that acts as a driver to provide and analyze test cases. Also, it is found that nearly as much effort will be made to create the proper stubs as in identifying test cases. Another important consequence is that much of the burden is shifted to integration testing. In fact, we can identify two levels of integration testing: intraclass and interclass integration.

Case 2: Classes as units: Testing a class as a unit solves the intraclass integration problem but it creates other problems. One has to do with various views of a class. In the static view, a class exists as source code. This is fine if all we do is code reading. The problem with the static view is that inheritance is ignored but we can fix this by using fully flattened classes. We might call the second view, the compile-time view, because this is when the inheritance actually "occurs." The third view is the execution-time view, when objects of classes are instantiated.

Testing really occurs with the third view but we still have some problems. For example, we cannot test abstract classes because they cannot be instantiated. Also, if we are using fully flattened classes, we will need to "unflatten" them to their original form when our unit testing is complete. If we do not use fully flattened classes, in order to compile a class, we will need all of the other classes above it in the inheritance tree. One can imagine the software configuration management (SCM) implications of this requirement.

The class as a unit makes the most sense when little inheritance occurs and classes have what we might call internal control complexity. The class itself should have an "interesting" state-chart and there should be a fair amount of internal messaging.

13.6. INTEGRATION TESTING OF CLASSES

Of the three main levels of software testing, integration testing is the least understood. This is true for both traditional and object-oriented software. As with traditional procedural software, object-oriented integration testing presumes complete unit-level testing.

Both unit choices have implications for object-oriented integration testing. If the operation/ method choice is taken, two levels of integration are required:

1. One to integrate operations into a full class.
2. One to integrate the class with other classes.

This should not be dismissed. The whole reason for the operation-as-unit choice is that the classes are very large and several designers were involved.

Assuming that a class is the basic unit choice, once the unit testing is complete two steps must occur:

1. If flattened classes were used, the original class hierarchy must be restored.
2. If test methods were added, they must be removed.

Once we have our "integration test bed" we need to identify what needs to be tested. As with our traditional software integration, static and dynamic choices can be made. We can address the complexities introduced by polymorphism in a purely static way: test messages with respect to each polymorphic context. The dynamic view of object-oriented integration testing is more interesting.

In addition to addressing excapsulation and polymorphism in object-oriented testing, another question that arises is what order do we put the classes together for testing? The various methods of integration like top-down, bottom up, big bang, and so on can all be applicable here. Please note the following points about object-oriented systems.

1. Object-oriented systems are inherently meant to be built out of small, resuable components. Hence, integration testing will be even more critical for object-oriented systems.
2. There is typically more parallelism in the development of the underlying components of object-oriented systems, thus the need for frequent integration is higher.
3. Given the parallelism in development, the sequence of availability of the classes will have to be taken into consideration while performing integration testing. This would also require the design of stubs and harnesses to simulate the function of yet-unavailable classes.

There are four basic strategies for integration:

1. Client/Supplier: The structure of class/supplier classes can be used to guide integration.
2. Thread based: The thread of messages activated to service a single user input or external event studied to decide the order of integration.
3. Configuration based: The thread of messages activated to service a single user input or external event studied to decide the order of integration.
4. Hybrid strategy: A mix of top-down, bottom-up, or big-bang integration can be used.

Before we discuss these methods, we must define some terminology used during integration testing. It is useful to distinguish several kinds of objects—

a. **Actor:** An object that changes state or uses other objects without receiving a message from another application object. An actor is never a recipient of a message from another object.
b. **Agent:** An object that accepts and sends object messages. An agent typically causes other objects to be created and used.
c. **Server:** An object that accepts messages from other objects but does not send messages.

We will now discuss these techniques one by one.

1. Client/Supplier Integration

This integration is done by "users." The following steps are followed:

Step 1. We first integrate all servers, that is, those objects that do not send messages to other application objects.

Step 2. Next we integrate agents, i.e., those objects that send and receive messages. This first integration build consists of the immediate clients of the application servers.

Step 3. There may be several agent builds.

Step 4. Finally, we integrate all *actors*, i.e., application objects that send messages but do not receive them.

This technique is called *client/supplier integration* and is shown in Figure 13.29.

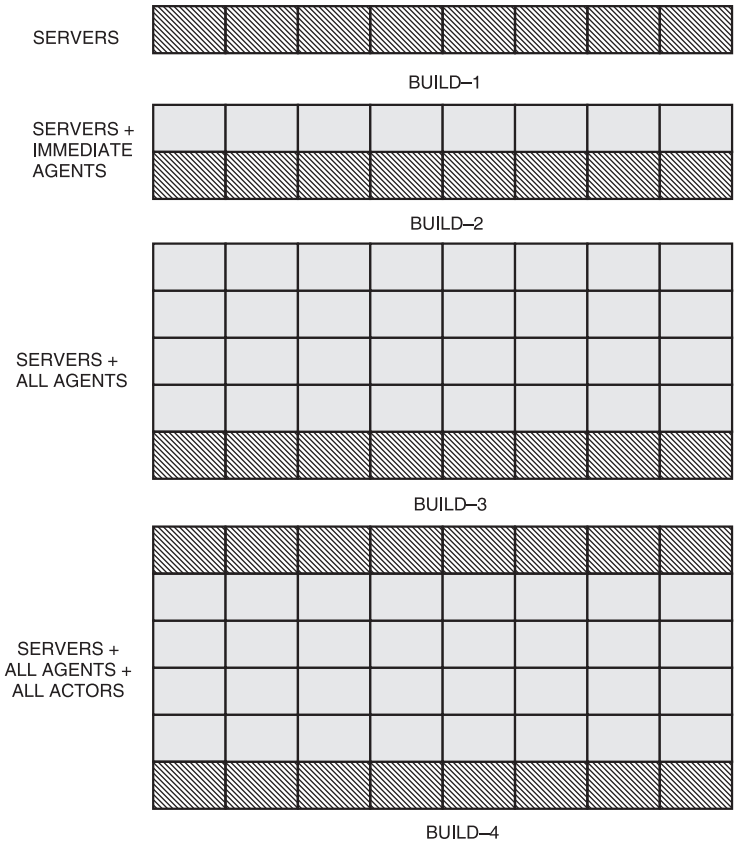


FIGURE 13.29 Client/Supplier Integration.

2. Thread Integration

Thread integration is integration by end-to-end paths. A use case contains at least one, possibly several threads.

Threaded integration is an incremental technique. Each processing function is called a thread. A collection of related threads is often called a build. Builds may serve as a basis for test management. The addition of new threads for the product undergoing integration proceeds incrementally in a planned fashion. System verification diagrams are used for "threading" the requirements.

3. Configuration Integration

In systems where there are many unique target environment configurations, it may be useful to try to build each configuration. For example, in a distributed system, this could be all or part of the application allocated to a particular node or processor. In this situation, servers and actors are likely to encapsulate the physical interface to other nodes, processors, channels, etc. It may be useful to build actor simulators to drive the physical subsystem in an controllable and repeatable manner.

Configuration integration has three main steps:

1. Identify the component for a physical configuration.
2. Use message-path or thread-based integration for this subsystem.
3. Integrate the stabilized subsystems using a thread-based approach.

4. Hybrid Strategy

A general hybrid strategy is shown in the following steps:

a. Do complete class testing on actors and servers. Perform limited bottom-up integration.

b. Do top-down development and integration of the high-level control modules. This provides a harness for subsequent integration.

c. Big-bang the minimum software infrastructure: OS configuration, database initialization, etc.

d. Achieve a high coverage for infrastructure by functional and structural testing.

e. Big-bang the infrastructure and high-level control.

f. Use several message path builds or thread builds to integrate the application agents.

Integration Test Checklist

The completion of the following checklist will reveal many interface faults and establish the basis for effective system testing.

1. **Set up infrastructure:**

   1.1 Configure environment.
   1.2 Create/Initialize files.
   1.3 Connect I/O devices, establish minimal handshaking.
   1.4 Compile, link, build, make, or assemble the entire system from a single library under SCM control.

2. **Verify executability:**

   2.1 Run all batch job streams to completion with minimal input.
   2.2 Bring up all tasks in an online system. Navigate all menu paths and perform normal shut down.
   2.3 Bring up all tasks in an embedded system. Accept normal signal inputs, produce all output at nominal values. Perform normal shut down.

3. **Verify minimal cooperative functionality:**

   3.1 Run through primary end-to-end threads: All event-response paths.
   3.2 Run through all cycles: on/off, polling loop, hourly, daily, weekly, monthly, etc.

Integration testing should not try for extensive coverage as this is the goal of unit and system test. Do good unit testing or make unit tests an explicit and separate part of your integration strategy. Don't stress the system. Do just enough thread and cycle testing to reveal interface and integration faults. Focus on stabilizing the system so that you can do efficient system testing.

13.7. SYSTEM TESTING (WITH CASE STUDY)

According to Myer, if you do not have written objectives for your product, or if your objectives are unmeasurable, then you cannot perform a system test.

Object-oriented systems are by design meant to be built using smaller sensable components (i.e., the classes). Due to this heavy reusability of

components, system testing becomes even important for object-oriented systems. This is due to the following reasons:

1. A class may have different parts not all of which are used at the same time. When different clients start using a class, they may be using different parts of a class and this may introduce defects at a later (system testing) phase.
2. Different classes may be combined together by a client and this combination may lead to new defects that are uncovered.
3. An instantiated object may not free all of its allocated resources, thus causing memory leaks and such related problems, which will show up only in the system testing phase.

Please note that proper entry and exit criteria should be set for the various test phases before system testing so as to maximize the effectiveness of system testing.

We describe four types of system testing.

1. **Functional testing:** A requirement is a capability, a feature, or function that the system must provide. Every explicitly stated requirement must be tested. We must, however, take care of the following:

   - Traceability from requirements to test cases is necessary.
   - Each test case, at all levels, should indicate which requirement (if any) it implements.
   - The test is simple: can you meet the requirement?

   Well-written, testable line-item requirements are a necessity. We must take care of the following:

   - If you don't have line-item requirements, use the primitives in your requirements model like PDLs.
   - If you don't have line-item requirements, extract a list of requirements from any and all available system documentation like user manuals, contracts, marketing literature, etc.

2. **Scenario-based system testing:** The notion of the "average" user or usage scenario can be misleading or too vague to construct useful tests. We need to define specific scenarios to explore dynamic behavior. For example:

   - Customer/user oriented definition.
   - Revalidates the product.

- Focuses development on customer requirements.
- Early customer involvement can build relationships.
- Real-world orientation.

A scenario is—who, what, when, where, how, and why scenarios have three dimensions—the users (system for embedded applications), the tasks they perform, and the environment for different tasks and users.

Who, how, and why?

- Define the significant types, classes, and categories of users.

What and when?

- Define from the user's point of view the tasks that use the system.
- A user task must have specific time characteristics (when): interval, cycle, duration, frequency, etc.

Where and how?

- Define all the variations for context of tasks and users.

Users

The following questions can help to identify user categories:

Who?

- Who are the users?
- Can you find any dichotomies?
  - Big company versus small
  - Novice versus experienced
  - Infrequent versus heavy user
- Experience: Education, culture, language, training, work with similar systems, etc.

Why?

- What are their goals in performing the task—what do they want?
- What do they produce with the system?

How?

- What other things are necessary to perform the task?
  - Information, other systems, time, money, materials, energy, etc.
- What methods or procedures do they use?

Environment

The user/task environment (as well as the OS or computer) may span a wide range of conditions.

Consider any system embedded in a vehicle. Anywhere the vehicle can be taken is a possible environment.

What external factors are relevant to the user? To the system's ability to perform? For example, buildings, weather, electromagnetic interference, etc.

What internal factors are relevant to the user? To the system's ability to perform? For example, platform resources like speed, memory, ports, etc., AC power system loading, multitasking.

With scenarios categories in hand, we can focus on specific test cases. This is called an *operational profile*.

An *activity* is a specific discrete interaction with the system. Ideally, an activity closely corresponds to an event-response pair. It could be a subjective definition but must have a start/stop cycle. We can refine each activity into a test by specifying:

- Probability of occurrence.
- Data values derived by partitioning.
- Equivalence classes are scenario-oriented.

Scenarios are a powerful technique but have limitations and require a concentrated effort. So, we have the following suggestions:

- User/customer cooperation will probably be needed to identify realistic scenarios.
- Scenarios should be validated with user/customer or a focus group.
- Test development and evaluation requires people with a high level of product expertise who are typically in short supply.
- Generate a large number of test cases.
- Well-defined housekeeping procedures and automated support is needed if the scenarios will be used over a long period of time by many people.

Next we consider a *case study* of ACME Widget Co.

We will illustrate the *operational profile* (or specific test cases) with the ACME Widget Co. order system.

Users

- There are 1000 users of the ACME Widget order system.
- Their usage patterns differ according to how often they use the system. Of the total group, 300 are experienced, and about 500 will use the system on a monthly or quarterly basis. The balance will use the system less than once every six months.

Environment

- Several locations have significantly different usage patterns.
- Plant, office, customer site, and hand-held access.
- Some locations are only visited by certain users. For example, only experienced users go to customer sites.

Usage

- The main user-activities are order entry, order inquiry, order update, printing a shipping ticket, and producing periodic reports.
- After studying the usage patterns, we find proportions vary by user type and location.
- For example, the infrequent user will never print a shipping ticket but is likely to request periodic reports.

Some scenarios are shown in Table 13.5.

TABLE 13.5 Showing Scenario Probability (p).

User type	P_1	Location	P_2	Activity	P_3	Scenario probability (p)
Experienced	0.3	Plant	0.80	Inquiry	0.05	0.0120
	0.3	Plant	0.80	Update	0.05	0.0120
	0.3	Plant	0.80	Print Ticket	0.90	0.2160
	0.3	Office	0.10	Order Entry	0.70	0.0210
	0.3	Office	0.10	Update	0.20	0.0060
	0.3	Office	0.10	Inquiry	0.10	0.0030
	0.3	Customer Site	0.10	Order Entry	0.10	0.0030
	0.3	Customer Site	0.10	Update	0.20	0.0060
	0.3	Customer Site	0.10	Inquiry	0.70	0.0210
Cyclical	0.5	Plant	0.10	Inquiry	0.05	0.0025
	0.5	Plant	0.10	Update	0.05	0.0025
	0.5	Plant	0.50	Print Ticket	0.90	0.0450
	0.5	Office	0.50	Order Entry	0.30	0.0025
	0.5	Office	0.50	Update	0.20	0.0450
	0.5	Office	0.50	Inquiry	0.50	0.0750
	0.5	Hand Held	0.40	Order Entry	0.95	0.1900
	0.5	Hand Held	0.40	Update	0.02	0.0040
	0.5	Hand Held	0.40	Inquiry	0.03	0.0060

(Continued)

User type	P_1	Location	P_2	Activity	P_3	Scenario probability (p)
Infrequent	0.2	Plant	0.05	Report	0.75	0.0075
	0.2	Plant	0.05	Update	0.15	0.0015
	0.2	Plant	0.05	Inquiry	0.10	0.0010
	0.2	Office	0.95	Inquiry	0.60	0.1140
	0.2	Office	0.95	Update	0.10	0.0190
	0.2	Office	0.95	Report	0.30	0.0570
						1.0000

There are two main parts in an operational profile: usage scenarios and scenario probabilities or:

Operational Profile = Scenarios + Probabilities

The factor proportions are multiplied to get scenario probability (p). That is,

$$p = p_1 \times p_2 \times p_3 \qquad \text{(from the above table)}$$

For example, looking at the first factor—user type experienced in column-1 of Table 13.5, we observe that

a. About 30% are experienced users.

b. The second factor (location) shows that 80% of the experienced group uses the system in the plant.

c. Nine times in 10 an experienced user in the plant will use the system to print a shipping ticket.

The column on the far right gives the probability of this scenario. Out of all users of the system, about 22% will be by experienced users in the plant to print a shipping ticket.

NOTE *The individual scenario probabilities must add to 1.0.*

If we sort the table in probability order, we have a prioritized test system strategy; we sort it in descending order of the scenario probability, p. Thus, we get Table 13.6.

TABLE 13.6 Sorted Scenario Probability (p) for Prioritization.

User type	p_1	Location	p_2	Activity	p_3	Scenario probability (p)
Experienced	0.3	Plant	0.80	Print Ticket	0.90	0.2160
Cyclical	0.5	Hand Held	0.40	Order Entry	0.95	0.1900
Cyclical	0.5	Office	0.50	Inquiry	0.50	0.1250
Infrequent	0.2	Office	0.95	Inquiry	0.60	0.1140
Cyclical	0.5	Office	0.50	Order Entry	0.30	0.0750
Infrequent	0.2	Office	0.95	Report	0.30	0.0570
Cyclical	0.5	Office	0.50	Update	0.20	0.0500
Cyclical	0.5	Plant	0.10	Print Ticket	0.90	0.0450
Experienced	0.3	Office	0.10	Order Entry	0.70	0.0210
Experienced	0.3	Customer Site	0.10	Inquiry	0.70	0.0210
Infrequent	0.2	Office	0.95	Update	0.10	0.0190
Experienced	0.3	Plant	0.80	Update	0.05	0.0120
Experienced	0.3	Plant	0.80	Inquiry	0.05	0.0120
Infrequent	0.2	Plant	0.05	Report	0.75	0.0075
Cyclical	0.5	Hand Held	0.40	Inquiry	0.03	0.0060
Experienced	0.3	Customer Site	0.10	Update	0.20	0.0060
Experienced	0.3	Office	0.10	Update	0.20	0.0060
Cyclical	0.5	Hand Held	0.40	Update	0.20	0.0060
Experienced	0.3	Customer Site	0.10	Order Entry	0.10	0.0030
Experienced	0.3	Office	0.10	Inquiry	0.10	0.0030
Cyclical	0.5	Plant	0.10	Inquiry	0.05	0.0025
Cyclical	0.5	Plant	0.10	Update	0.05	0.0025
Infrequent	0.2	Plant	0.05	Update	0.15	0.0015
Infrequent	0.2	Plant	0.05	Inquiry	0.10	0.0010
						1.0000

The operational profile is a framework for a complete test plan. For each scenario, we need to determine which functions of the system under test will be used.

An activity often involves several system functions; these are called "runs." Each run is a thread. It has an identifiable input and produces a distinct output.

For example, the experienced/plant/ticket scenario might be composed of several runs.

- Display pending shipments.
- Display scheduled pickups.
- Assign carrier to shipment.
- Enter carrier landing information.
- Print shipment labels.
- Enter on-truck timestamp.

Some scenarios may be low probability but have high potential impact. For example, suppose ACME Widget is promoting order entry at the customer site as a key selling feature. So, even though this accounts for only 3 in a thousand uses, it should be tested as if it was a high-priority scenario.

This can be accomplished by adding a weight to each scanario:

Weights	Scenario
+2	Must test, mission/safety critical
+1	Essential functionality, necessary for robust operation
+0	All other scenarios

The operational profile maximizes system reliability for a given testing budget. According to J.D. Musa, *"testing driven by an operational profile is very efficient because it identifies failures on average, in order of how often they occur. This approach rapidly increases reliability— reduces failure intensity—per unit of execution time because the failures that occur most frequently are caused by the faulty operations used most frequently.*

User will also detect failures in order of their frequency, if they have not already been in test."

3. **Performance testing:** Performance is the behavior of the system with respect to goals for time, space, cost, and reliability.

Performance testing requires:

- Clear, objective test criterion
- Controlled, well-instrumented test-bed
- Data reduction support
- Automated test drivers

We will look at two aspects of performance test:

1. Performance objectives.
2. Considerations for running performance tests.

Software performance measurement parameters are throughput, response time, and utilization. We will discuss their benefits, definitions, and examples in the table below:

Objective	Benefit	Definition	Example
1. Throughput	Productivity	Number of tasks accomplished per unit time.	Transactions per second
2. Response time	Responsiveness	Time elapsed between input arrival and delivery of output at sink.	Check to display delay
3. Utilization	Component availability	Ratio of time busy/available for a component over a fixed interval.	Server utilization

The following example shows the difference between throughput, response time, and utilization. The same tasks are run on similar processors but executed in a different order. Tasks arrive at the same time.

Time and space cost	Processor–A		Processor–B	
	Task time	Response time	Task time	Response time
	1	1	5	5
	2	3	4	9
	3	6	3	12
	4	10	2	14
	5	15	1	15
Totals	15	35	15	55
Average response		7		11
Worst case response		15		15
Throughput	5		5	
Utilization	100%		100%	

Service-level objectives target performance features visible to users under various operational scenarios:

- Peak load performance.
- Average load performance.
- Worst-case performance.
- Availability profile.

Average or worst-case tolerance depends on the application. Worst-case performance is relevant for mission critical systems like air traffic control. Average performance is often the focus of data processing systems.

Performance objectives specify unambiguous, quantifiable targets for system behavior. A system performance requirement can be stated in terms of response time, throughput, or utilization.

This objective must be translated into a constraint on software execution time or resource use. Response time is the dominant performance issue for many applications. Response time constraints are of the form:

Elapsed Execution Time < Response Time Objective

There are several variations of performance testing. They are given below one by one.

a. Volume test: This is simple quantity saturation. Lots of input, with no constraint on time under normal system loading.

b. Background test: This is an attempt to find faults related to concurrent, parallel, or multitasking. We try to get several things going concurrently to see if we can force:

- Resource contention resolution faults.
- Scheduling and deadlock faults.
- Race conditions.

We try to sample loading levels without stressing the system. This is good at detecting:

- Basic timing problems.
- Faults with re-entrant code.
- Data integrity faults.

This sets the stage for stress testing.

c. Stress test: The idea of a stress test is to "break" the system. That is, we want to see what happens when the system is pushed beyond design limits.

4. Configuration testing: Some system products are intended for use in a wide range of platform installations.

For example, consider a typical PC package. It must run on several computer makes and models, OS versions, and support every printer and monitor sold.

Configuration testing should be done when everything else is stable.

Test cases are a matter of selecting a subset of configuration permutations:

a. Identify all possible configuration variables and domains, requirements, and marketing.

b. Select a subset using domain analysis.

c. Pay attention to "special" or "custom" configurations promised to specific customers or users.

At a minimum, the test suite should cover a normal duty cycle for the devices in configuration:

- Start-up/run/shut down
- Start-up/device failure/recovery/run/shut down

The system testing approaches that we have discussed can be used to investigate other relevant categories also. Some of them are listed below:

- Security
- Restart and recovery
- Parallel
- Reliability
- Installation
- User documentation
- Operator procedures

13.8. REGRESSION AND ACCEPTANCE TESTING

As we know *reusability of code* is the focal point of object-oriented programming. So changes to any one component could have potentially unintended side-effects on the clients that use the component. Also, because of cascaded effects of changes resulting from properties like inheritance, it makes sense to catch the defects as early as possible.

What Is Regression Testing?

It is defined as the selective retesting of system or component after changes have been made.

Or

It is defined as the process to verify absence of unintended effects.

Or

It is defined as the process of verifying compliance with all old and new requirements.

Approach used

- Reveal faults in new or modified modules. This requires running new test cases and typically reusing old test cases.
- Reveal faults in unchanged modules. This requires re-running old test cases.
- Requires reusable library of test suites.

Scope of regression test

- Regression test suite with functional or interface subsets.
- Run entire test suite periodically.
- Run subset on every new integration.

The selection of subset and frequency requires evaluation of time/cost effectiveness trade offs.

When to do?

- Periodically, every three months.
- After every integration of fixes and enhancements.
- Frequency, volume, and impact must be considered.

What to test?

- Changes result from new requirements or fixes.
- Analysis of the requirements hierarchy may suggest which subset to select.
- If new modules have been added, you should redetermine the call paths required for CI coverage.

Possible test suites

- **Maximal:** Test changed component, all ancestors, and descendants.
- **Minimal:** Test changed component and immediate ancestors.
- Old test cases may need to be revised.

Acceptance Testing

On completion of the developer administered system test, three additional forms of system testing may be appropriate.

a. Alpha test: Its main features are:
   1. It is generally done "in-house" by an independent test organization.
   2. The focus is on simulating real-world usage.
   3. Scenario-based tests are emphasized.
b. Beta test: It's main features are:
   1. It is done by representative groups of users or customers with prerelease system installed in an actual target environment.
   2. Customer attempts routine usage under typical operating conditions.
   3. Testing is completed when failure rate stabilizes.
c. Acceptance test: Its main features are:
   1. Customer runs test to determine whether or not to accept the system.
   2. Requires meeting of the minds on acceptance criterion and acceptance test plan.

13.9. MANAGING THE TEST PROCESS

A comprehensive test plan package is described in IEEE 83b, which is an accepted industry standard for test planning. It recommends the following main documents:

1. The test plan: It defines the features to be tested and excluded from testing. It covers approach, deliverables, suspend/resume criteria, environmental needs, responsibilities, staffing and training, schedule, risks and contingencies, and approvals.

2. The test design: It defines the features/functions to test and the pass fail criterion. It designates all test cases to be used for each feature/functions.
3. The test cases: It defines the items to be tested and provides traceability to SRS, SDD. User operations or installation guides. It specifies the input, output, environment, procedures, intercase dependencies of each test case.
4. Test procedures: It describes and defines the procedures necessary to perform each test.

Each item, section, and sub-section should have an identifying number and designate date prepared and revised, authors, and approvals.

How should we go about testing a module, program, or system? What activities and deliverables are necessary? A general approach is described in IEEE 87a, an accepted industry standard for unit testing. It recommends four main steps:

Step 1. Prepare a testing plan: Document the approach, the necessary resources, and the exit criterion.

Step 2. Design the test:
- 2.1 Develop an architecture for the test, organize by goals.
- 2.2 Develop a procedure for each test case.
- 2.3 Prepare the test cases.
- 2.4 Package the plan per IEEE 82a.
- 2.5 Develop test data.

Step 3. Test the components:
- 3.1 Run the test cases.
- 3.2 Check and classify the results of each test case:
  - 3.2.1 Actual results meet expected results.
  - 3.2.2 Failure observed:
    - Implementation fault.
    - Design fault.
    - Undetermined fault.
  - 3.2.3 Unable to execute test case:
    - Fault in test specification or test data.
    - Fault in test procedure.
    - Fault in test environment.

Step 4. Prepare a test summary report: Identify the component, document observed variances from specifications, summarize test case results, evaluate component usability, summarize testing activities, and obtain necessary approvals.

A Test Plan Schema

A detailed test case schema is presented in [Berard] is given below. This schema is a tree. So, there will be typically many branches and leaves for a single root (object test case). The template of a test plan schema is given next.

```
Object Test Case
     Object Id
     Test case ID
     Purpose
     List of test case steps
          test case step
               list of states
                    state transition
                        expected state
                        actual state
               list of messages or operations
                    name and parameters
                        message name
                        input parameters
                            name
                            value
                            position
                            type
                        output parameters
                            expected value
                            actual value
                            name
                            value
                            position
                            type
               list of exceptions raised
                    expected exception
                    actual exception
```

How much testing is enough?

Three general types of exit criteria have been proposed:

1. **Sufficient coverage:** Testing ceases when:
   - statement coverage is achieved
   - decision coverage is achieved

- path coverage is achieved
- all du-path coverage is achieved
- call path coverage is achieved

You will probably not be able to do 100% predicate coverage. 85% is practical goal.

2. **Failure rate stabilization:** Testing ceases when:
   - the rate of failures versus CPU test time is low and steady.
3. **Residual fault threshold:** Testing ceases when:
   - the ratio of detected faults to the estimated number of total faults becomes acceptable.

The economic, ethical, legal, and practical considerations for an application to determine adequate testing. They are summarized below:

Type	Example	Appropriate testing
Life critical	Avionics, medical, motion control	Parallel development, comparison tests
Mission critical	Weapon control systems, secure communications, process control	Independent V&V
Asset management	Wire transfer, product inventory, payables/receivables	Security testing
External market	Communications, OS, GUI	Alpha, beta
Internal users	Order processing	Formal and informal
One-shot project tools	Test drivers, command files	Informal
Personal use	Tools, games	Hack and go

Choice of Standards

The planning aspects are proactive measures that can have an across-the-board influence on all testing projects.

Standards comprise an important part of planning in any organization. Standards are of two types:

1. External standards
2. Internal standards

External standards are standards that a product should comply with, are externally visible, and are usually stipulated by external consortia. From a testing perspective, these standards include standard tests supplied by external consortia and acceptance tests supplied by customers.

Internal standards are standards formulated by a testing organization to bring in consistency and predictability. They standardize the processes and methods of working within the organization. Some of the internal standards include

1. Naming and storage conventions for test artifacts
2. Document standards
3. Test coding standards
4. Test reporting standards

These standards provide a competitive edge to a testing organization. It increases the confidence level one can have on the quality of the final product. In addition, any anomalies can be brought to light in a timely manner.

Testing requires a robust infrastructure to be planned upfront. This infrastructure is made up of three essential elements. They are as follows:

a. A test case database
b. A defect repository
c. SCM repository and tool

A test case database captures all of the relevant information about the test cases in an organization.

A defect repository captures all of the relevant details of defects reported for a product. It is an important vehicle of communication that influences the work flow within a software organization.

A software configuration management (SCM) repository/CM repository keeps track of change control and version control of all the files/entities that make up a software product. A particular case of the files/entities is test files.

13.10. DESIGN FOR TESTABILITY (DFT)

Design for testability (DFT) is a strategy to align the development process for maximum effectiveness under either a reliability-driven or resource-limited regime.

A reliability-driven process uses testing to produce evidence that a pre-release reliability goal has been met.

A resource-limited process views testing as a way to remove as many rough edges from a system as time or money permits.

Testability is important in either case. It

- Reduces cost in a reliability-driven process.
- Increases reliability in a resource-limited process.

Object-oriented systems present some unique obstacles to testability as well sharing many with conventional development.

This cost and difficulty can be reduced by following some basic design principles and planning for test.

Broadly conceived, software testability is a result of many factors.

Some of them are:

a. Characteristics of the representation.

b. Characteristics of the implementation.

c. Built-in test capabilities.

d. The test-suite.

e. The test support environment.

f. The software process in which testing is conducted.

We will now discuss a fishbone chart to consider testability relationships.

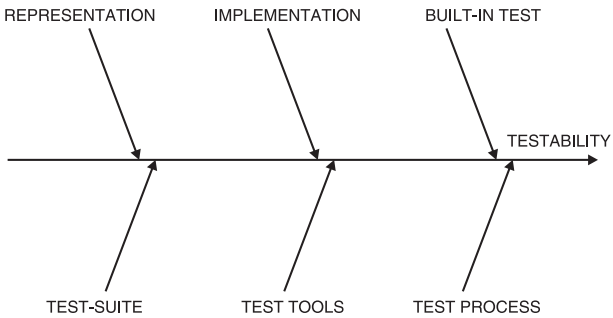


FIGURE 13.30 Fishbone Chart.

1. **Representation:** The presence of a representation and its usefulness in test development is a critical testability factor. This is because

   a. Testing without a representation is simply experimental prototyping.

   b. It cannot be decided that a test has passed or failed without an explicit statement of expected result.

c. It may force production of a partial representation as part of test plan.

There are many approaches to developing object-oriented representations, generically known as object-oriented analysis (OOA) and object-oriented design (OOD).

2. **Implementation:** An object-oriented program that complies with generally accepted principles of OOP poses the fewest obstacles to testing.

Structural testability can be assessed by a few simple metrics. A metric may indicate testability, scope of testing, or both.

For example, with high coupling among classes, it is typically more difficult to control the class-under-test (CUT), thus reducing testability.

The effect of all intrinsic testability metrics is same:

- relatively high value = decreased testability
- relatively low value = increased testability

Scope metrics indicate that the number of tests is proportional to the value of the metric.

A good design always improves testability but some applications may be inherently hard to test. Testability metrics provide information useful for resolving design trade-offs. They do not represent some kind of score to be maximized or minimized.

3. **Built-in test:**

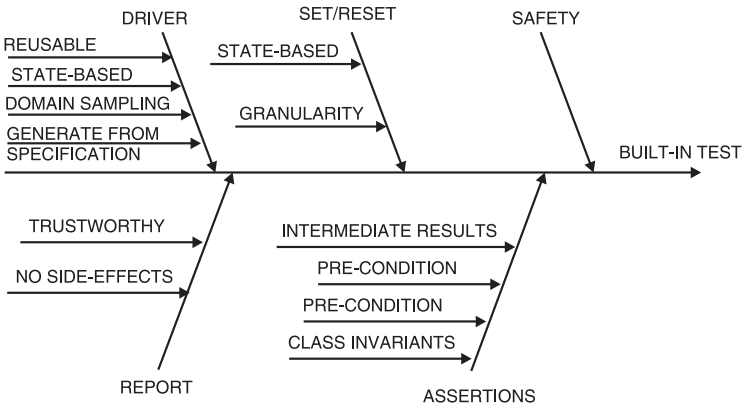


FIGURE 13.31 Fish Bone Chart.

Built-in test features are shown in Figure 13.31 and are summarized below:

1. Assertions automate basic checking and provide "set and forget" runtime checking of basic conditions for correct execution.
2. Set/Reset provides controllability.
3. Reporters provide observability.
4. A test suite is a collection of test cases and plan to use them. IEEE standard 829 defines the general contents of a test plan.
5. Test tools require automation. Without automationless testing, greater costs will be incurred to achieve a given reliability goal. The absence of tools inhibits testability.
6. Test process: The overall software process capability and maturity can significantly facilitate or hinder testability. This model follows the key process ability of the defined level for software product engineering.

13.11. GUI TESTING

The main characteristics of any graphical user interface (GUI) application is that it is event driven. Users can cause any of several events in any order. Many GUIs have an event sequence that is not guided. One benefit of GUI applications to testers is that there is little need for integration testing. Unit testing is typically at the "button level"; that is, buttons have functions and these can be tested in the usual unit-level sense. The essence of system-level testing for GUI applications is to exercise the event-driven nature of the application. Unfortunately, most of the models in UML are of little help with event-driven systems. The main exception is behavioral models, specifically statecharts and their simpler case, finite-state machines (FSMs).

13.12. COMPARISON OF CONVENTIONAL AND OBJECT-ORIENTED TESTING

Although there is basically no difference in the method of testing, whether one is testing a non-object-oriented application or an object-oriented application; still the following aspects applicable mostly for object-oriented

applications may be kept in mind and testing tactics accordingly devised. The main points of comparisons are given below:

1. Object-oriented application development requires object-oriented analysis (OOA) and object-oriented design (OOD) and very often the code is generated from the design models-OOM (object-oriented modelling). This overwhelming importance of OOAD (object-oriented analysis and design) and OOM requires that testing lay special stress on verifying analysis and models, that being critical.
2. Unit testing in OOT context would mean testing not an isolated method or function or procedure, but a class as a whole.
3. In non-OOT, we independently test Module-X and Module-Y and then under integration testing, integrate them together and test mainly for the correctness of their interface. In OOT, however, Module-X and Module-Y may share many components and may also have classes inherited from common base class. One of the distingushing OOT approaches would therefore be to do layered-testing. In layered-testing we test base classes (bottom layer) first, followed by the next layer and so on; that is, proceed from independent classes to dependent classes.
4. A properly designed OOApp (object-oriented application) is an excellent candidate for black-box testing, as each component/class ought to behave as a black-box with well defined sets of inputs and outputs.
5. Model base testing (MBT) that is a black-box testing strategy is greatly facilitated by OOM.
6. For systematic testing, partition testing at the class level can be used. Partition testing can be categorized under:
   a. **State-based partitioning:** Operations that change the state of the class (like credit, debit, etc.) and not those that merely fetch an information (like date of maturity, balance, etc.).
   b. **Attribute-based partitioning:** Class operations based on attributes, that is operations, that use a given attribute (like balance, etc.).
   c. **Function-group based partitioning:** Some examples of such partitioning are open, close, transact, report, etc.

13.13. TESTING USING ORTHOGONAL ARRAYS

Although some applications have a small number of input values, listing all of the possible combinations may increase the number of test cases. One technique to reduce the number of test cases is to apply a combinational method known as orthogonal array testing (OAT). This method can be used for both normal software applications as well as object-oriented software.

To apply the technique, the software must have independent sets of states. The goal is to pair each state (from one set) with every other state (from another set) at least once. We need to identify unique pairs of states.

Let us consider an example of a bookstore application. Further assume that there are three classes:

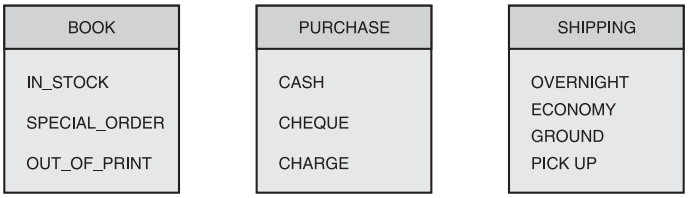


FIGURE 13.32 Classes and States in a Bookstore.

Each of these three classes have a finite number of possible states in an object-oriented application. Whereas in a procedural application, if we have three procedures, each will have arguments with a finite set of values.

Now, from the previous figure we find that two classes (book and purchase) have three states each and one class has four states, so testing every combination of states requires $3 \times 3 \times 4 = 36$ test cases. Selecting a state from each of the two classes (known as the pair-wise combinations) only requires 12 test cases as shown in table below, thereby reducing the number of possible test cases.

TABLE 13.7 An Orthogonal Array.

Test case	Book	Purchase	Shipping
1.	in-stock	cash	overnight
2.	in-stock	check	economy
3.	in-stock	charge	ground
4.	in-stock	cash	pick-up

(Continued)

Test case	Book	Purchase	Shipping
5.	special-order	check	overnight
6.	special-order	check	economy
7.	special-order	cash	ground
8.	special-order	check	pick-up
9.	out-of-print	charge	overnight
10.	out-of-print	cash	economy
11.	out-of-print	check	ground
12.	out-of-print	charge	pick-up

Test case 1 in the table above says to test the combination that has book set to "in-stock," purchase set to "cash," and shipping set to "overnight."

How to implement this test case?

In a procedural language, the test consists of passing the parameters "in-stock," "cash," and "overnight" to the procedures order-book, purchase-book, and ship-book, respectively. However, in object-oriented programming, this test consists of having an object of class book, in state "in-stock," sends a message that passes an object of class purchase, in state "cash," to an object of class shipping in state "overnight."

Test cases 9-12 have some improbable combinations. It shows "out-of-print" state of the book. A user cannot purchase a book which is not of store/print. A software tester may find this a non-feasible combination and may remove it from the set of test cases. On the other hand, executing such a test ensures that the application traps this condition and returns the proper error message.

When reducing the number of test cases, the possibility of missing a crucial combination always exists. One can use risk analysis to identify such crucial tests and give them priorities.

Orthogonal array testing is a powerful technique to use any time a system presents a small input domain with a large number of permutations.

13.14. TEST EXECUTION ISSUES

After defining the test cases, the next step is to formulate an environment in which to execute the tests. This requires special test software, known as a driver. It invokes the software-under-test (SUT). The typical driver consists of the following features:

a. Packaging input and data

b. Invoking the software-under-test (SUT)

c. Capturing the resulting outcome

Creating a driver to test a class calls for additional considerations. The driver must take care of encapsulation so as to control and observe a class's internal data and methods. There are several ways to achieve this, such as:

- Providing a test case method for each class
- Creating a parallel class, which appears identical to the original except for the addition of code needed for testing
- Creating a child class that inherits the methods to be tested

All of these methods, alter the class and thus modify the actual application. Consequently the tested implementation may not be identical to the released code. It is important that the testing environment be as close as possible to the deployment environment, otherwise, you are not fully testing the end product.

Drivers for testing classes can take on many forms. The testers must analyze the trade offs when deciding on a driver scheme. Creating effective and reusable test drivers takes significant planning and effort, even if commercial tools are used.

13.15. CASE STUDY—CURRENCY CONVERTER APPLICATION

Problem Statement: The currency converter application coverts U.S. dollars to any of the four currencies: Brazilian real, Canadian dollars, European euros, and Japanese yen. The user can revise inputs and perform repeated currency conversion. This program is an event-driven program that emphasizes code associated with a GUI. The GUI is shown in Figure 13.33.

The user selects the country whose currency equivalent is desired using radio buttons which are mutually exclusive. When the compute button is

clicked, the result is displayed. The clear button will clear the screen. Clicking on the quit button ends the application.

Now, we will perform the following on this GUI application:

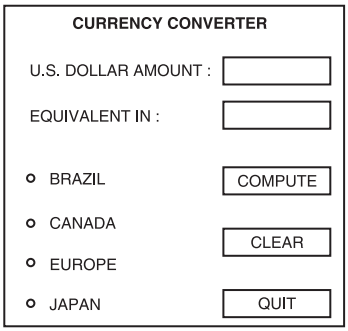


FIGURE 13.33

1. **GUI testing of this application:** To test a GUI application, we begin by identifying all of the user input events and all of the system output events. Some of them are given below:

Input events		Output events	
inp1	Enter U.S. dollar amount	op1	Display U.S. dollar amount
inp2	Click on a country button	op2	Display currency name
inp2.1	Click on Brazil	op2.1	Display Brazilian reals
inp2.2	Click on Canada	op2.2	Display Canadian dollars
inp2.3	Click on Europe	op2.3	Display European euros
		op2.4	Display Japanese yen
inp2.4	Click on Japan	op2.5	Display ellipsis
inp3	Click on compute button	op3	Indicate selected country
inp4	Click on clear button	op4	Reset selected country
inp5	Click on quit button	op5	Display foreign currency value
inp6	Click on OK in error message	op6	Must select a country : Error message

2. **Unit testing of currency conversion program:** The command buttons (i.e., Compute, Clear, and Quit) on the form have an event-driven code attached to them. Hence, they are sensible places to perform unit-level testing. Unit testing of the compute button should also consider invalid U.S. dollar amount entries such as non-numerical inputs, negative inputs, and very large inputs.

What are the best methods for performing unit-level testing?

One method is to run test cases from a specially coded driver that would provide values for input data and check output values against expected values.

A second method is to use the GUI as a test bed. This looks like system-level unit testing, which seems oxymoronic, but it is workable. It is system level in the sense that test case inputs are provided via system-level user input events. And the test case result comparisons are based on system-level output events. This works for small applications but it does beg some serious questions such as the computation is correct but a fault occurs in the output software. Another problem is that it will be harder to capture test execution results. It also has some other problems like repeating a set of test cases is time consuming.

Hence, unit testing with a test driver looks preferable.

Unit testing with the U.S. dollar amount text box can be done. It is language dependent. If it is implemented in VB then there is little need to unit test the test box. But if we use an OOP language then we would need to verify that the inputs observed by the keyboard handler are correctly displayed on the GUI and are correctly stored in the object's attributes.

3. **Integration testing of currency conversion program:** Whether integration testing is suitable for this problem depends on how it is implemented. Three main choices are available for the compute button.

First choice concentrates all logic in one place and simply uses the status of the option buttons as in IF tests. Such IF tests would be thoroughly done at the unit level. So, there is little need for integration testing.

The second and more object-oriented choice would have methods in each option button object that send the exchange rate value for the corresponding country in response to a click event. In this case, we notice that as units, there is very little to test. That is why the choice of methods as units depends on size. Everything of interest is at the integration level and at that level we have two concerns—are the option/radio buttons sending the right exchange rate values and is the equivalent amount calculation correct?

The third choice is in the visual basic style. The code for VB includes a global variable for the exchange rate. There are event procedures for each option button. The result is very similar to the pure object-oriented version. Unit testing could be replaced by simple code reading for the option button

event procedures and the testing of the command compute procedure is similarly trivial.

Note that in all the three variations given previously there is little need for integration testing. This is true for small GUI applications only. By "small" we mean to say an application that is implemented by one person.

4. **System testing for currency conversion program:** As we have already discussed, unit and integration testing are minimally needed for small GUI applications. The onus therefore shifts to system testing.

   System testing may be

   a. UML-based system testing.

   b. Statechart-based system testing.

In the first step, also called project inception, the customer/user describes the application in very general terms. Three types of system functions are identified—evident, hidden, and frill. Evident functions are the obvious ones. Hidden functions might not be discovered immediately and frills are the "bells and whistles" that so often occur. The table below lists the system functions for the currency converter application.

Reference no.	Function	Category
R1	Start application	Evident
R2	End application	Evident
R3	Input U.S. dollar amount	Evident
R4	Select country	Evident
R5	Perform conversion calculation	Evident
R6	Clear user inputs and program outputs	Evident
R7	Maintain EX-OR relationship among countries	Hidden
R8	Display country flag images	Frill

We first discuss UML-based system testing.

Our formulation lets us be very specific about system-level testing. There are four levels with corresponding coverage metrics for GUI applications. Two of these are naturally dependent on the UML specification.

First Level: To test the system functions by developing an incidence matrix as shown below:

TABLE 13.8 Use Case Incidence with System Functions.

Expanded essential use cases	R1	R2	R3	R4	R5	R6	R7
1	×	—	—	—	—	—	—
2	—	×	—	—	—	—	—
3	—	—	×	×	×	—	—
4	—	—	×	×	×	—	—
5	—	—	×	×	×	—	×
6	—	—	×	×	—	×	×
7	—	—	×	—	×	—	—
8	—	—	×	—	×	—	—
9	—	—	—	—	×	—	—

Examining the incidence matrix, we can see several possible ways to cover the seven system functions. One way would be to derive test cases from real-use cases that correspond to extended essential-use cases 1, 2, 5, and 6. These will need to be real-use cases as opposed to the expanded essential-use cases. The difference is that specific countries and dollar values are used, instead of the higher level statements such as "click on a country button" and enter a dollar amount.

The process of deriving system test cases from real-use cases is mechanical. The use case preconditions are the test-case preconditions and the sequences of actor actions and system responses map directly into sequences of user input events and system output events.

The set of extended essential-use cases 1, 2, 5, and 6 is a nice example of a set of regression test cases because test-case 5 in the above table covers four system functions and test-case 6 covers 3.

Second Level: To develop test cases from all of the real-use cases. We can develop system level test cases from the real-use case based on extended essential-use case. For example, this has been applied on real-use-case-3 (RUC-3) that is obtained from extended essential-use-case-3. So, our RUC-3 is

	RU3 Normal usage	
	Actor(s)	User
	Preconditions	txtDollar has focus
	Type	Primary
	Description	The user inputs a U.S. $10 and selects the European option; the application computes and displays the equivalent: 9.30 euros
Sequence	Actor action	System response
	1. User enters 10 from keyboard	**2.** 10 appears in txt Dollar
	3. User clicks on European button	**4.** Euros appears in IblEquiv
	5. User clicks cmd-compute button	**6.** 9.30 appears in IbIEq Amt
Alternative reversed	Actions 1 and 3 can be reversed and consequently responses 4 and 6 will be sequence	
Cross-reference	R3, R4, R5	
Post-conditions	Cmd clear has focus	

This is RUC-3. Based on this real-use case, we derive system-level test cases also. They are given below.

SysTC3	Normal usage ($ amount entered)	
Test operator	Rajiv Chopra	
Pre-conditions	txtDollar has focus	
Test operator	Tester inputs	Expected system response
sequence	**1.** Enters 10 from keyboard	**2.** Observe 10 appears in txtDollar
	3. Click on the European button	**4.** Observe euros appears in lblEquiv
	5. Clicks cmdcompute button	**6.** Observe 9.30 appears in Ibl Eq. Amt
Post-conditions	cmdclear has focus	
Test Result	Pass/Fail	
Data Run	September 6, 2007	

Third Level: To derive test cases from the finite state machines derived from a finite state machine description of the external appearance of the GUI. This is shown below:

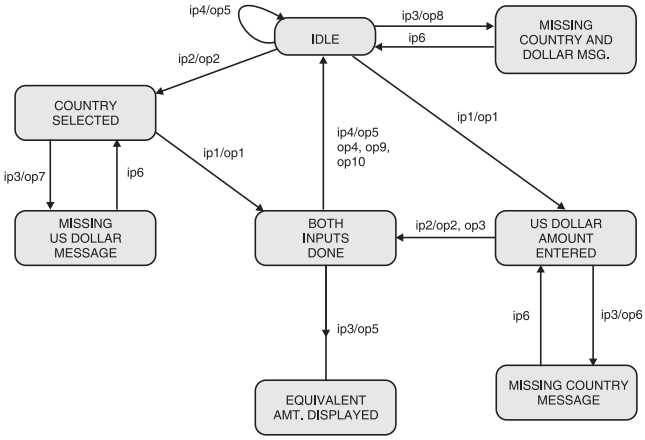


FIGURE 13.34 GUI Finite State Machine.

A test case from this formulation is a circuit. A path in which the start node is the end node is usually an idle state. Nine such test cases are shown in the table below. The numbers in the table show the sequence in which the states are traversed by the test case. The test cases, TC1 to TC9, are as follows:

State	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9
Idle	1	1	1	1	1	1	1	1	1, 3
Missing country and dollar message							2	2	
Country selected		2	2, 4			2			4, 6
U.S. dollar amount entered				2	2, 4		2		
Missing U.S. dollar msg			3						5
Both inputs done		3	5	3	5	3	3		7
Missing country msg					3				
Equivalent amount displayed		4	6	4	6				
Idle	2	5	7	5	7	1	1	1	1

FIGURE 13.35 Test Cases Derived from FSM.

Note that these numbers in columns TC1-TC9 show the sequence in which states are traversed by the test case. Many other cases exist but this shows how to identify them.

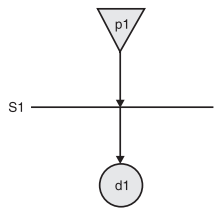
Fourth Level: To derive test cases from state-based event tables. This would have to be repeated for each state. We might call this the exhaustive level because it exercises every possible event for each state. However, it is not truly exhaustive because we have not tested all sequences of events across states. The other problem is that it is an extremely detailed view of system testing that is likely very redundant with integration and even unit-level test cases.

Now, we will discuss statechart-based system testing.

Statecharts are a fine basis for system testing. The problem is that Statecharts are prescribed to be at the class level in UML. There is no easy way to compose Statecharts of several classes to get a system-level Statechart. A possible solution is to translate each class-level Statechart into a set of event-driven petri nets (EDPNs) to describe threads to be tested. Then the atomic system functions (ASFs) and the data places are identified. Say, For our GUI-application they are as follows:

Atomic system functions	Sense click on Compute button
S1 : Store US dollar amount S2 : Sense click on Brazil S3 : Sense click on Canada S4 : Sense click on Europe S5 : Sense click on Japan S6 : Sense click on Compute button	S7 : Sense click on Clear button S8 : Sense click on Quit button **Data places** are d1 : US dollar amount entered d2 : Country selected

FIGURE 13.36 ASFs and Data Places.



The next step in building an EDPN description of the currency conversion GUI is to develop the EDPNs for the individual atomic system functions. For example,

where p1 was "Enter US dollar amount" and d1 is "US dollar amount entered."

Then, system-level threads are built up by composing atomic system functions into sequences. We are finally in a position to describe various sets of system-level test cases for the currency conversion GUI. The lowest level is to simply exercise every atomic system function. This is a little artificial because many atomic system functions (ASFs) have data outputs that are not visible system-level outputs. Even worse, the possibility always exist that an ASF has no port outputs, like with S1 : Store US dollar amount.

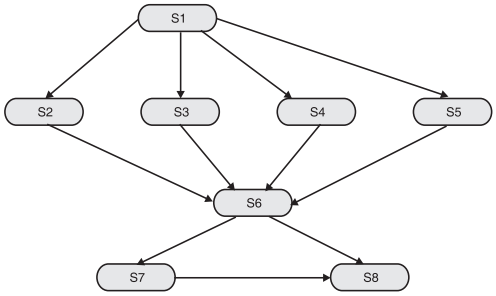


FIGURE 13.37 Directed Graph of ASF Sequences.

At the system testing level, we cannot tell if an amount is correctly stored, although we can look at the screen and see that the correct amount has been entered.

The next level of system testing is to exercise a "suitable" set of threads. By suitable we mean that we should exercise a set of threads that:

- Uses every atomic system function.
- Uses every port input.
- Uses every port output.

Beyond this we can look at a directed graph of ASFs as shown in Figure 13.37.

This is only a partial graph showing mainline behavior. The thread

<S1, S2, S6, S7>

is one of the 16 paths. Half of these end with the clear button clicked and the other half end with the quit button.

Consider the set, T, of threads {T1, T2, T3, T4} where their ASF sequences are given below:

T1 = <S1, S4, S6, S7>
T2 = <S1, S2, S6, S7>
T3 = <S1, S3, S6, S7>
T4 = <S1, S5, S7, S8>

The threads (T1-T4) given above, in set, T, have the following coverages:

- Every ASF
- Every port input
- Every port output

As such, set T, constitutes a reasonable minimum level of system testing for the currency conversion GUI. Similarly, we can go into more detail by exploring some next-level user behavior.

SUMMARY

Various key object-oriented concepts can be tested using some testing methods and tools. They are summarized below:

Key object-oriented concept	Testing methods and tools
1. Object orientation	▪ Tests need to integrate data and methods more tightly.
2. Unit testing of classes	▪ BVA, equivalence partitioning to test variables. ▪ Code coverage methods for methods of a class. ▪ Alpha-Omega method of exercising methods. ▪ State diagram to test states of a class. ▪ Stress testing to detect memory leaks.
3. Abstract classes	▪ Requires retesting for every new implementation of the abstract class.
4. Encapsulation and inheritance	▪ Requires unit testing at class level and incremental class testing when encapsulating. ▪ Inheritance introduces extra context; each combination of different contexts has to be tested.
5. Polymorphism	▪ Each of the different methods of the same name should be tested separately. ▪ Maintainability of code may suffer.
6. Dynamic binding	▪ Conventional code coverage has to be modified to be applicable for dynamic binding ▪ Possibility of unanticipated runtime defects higher.
7. Inter-object communication via messages	▪ Message sequencing. ▪ Sequence diagrams.
8. Object reuse and parallel development of objects	▪ Needs more frequent integration tests and regression tests. ▪ Integration testing and unit testing are not as clearly separated as in the case of a procedure-oriented language. ▪ Errors in interfaces between objects likely to be more common in object-oriented systems and hence needs thorough interface testing.

MULTIPLE CHOICE QUESTIONS

1. UML stands for:
   - **a.** Unified modeling language
   - **b.** Universal modeling language
   - **c.** Uniform modeling language
   - **d.** None of the above.
2. A general purpose mechanism for organizing elements into groups in UML is called
   - **a.** Package
   - **b.** Class
   - **c.** Deployment
   - **d.** None of the above.
3. Which one of the following is an example of call and return architecture?
   - **a.** Object-oriented architecture
   - **b.** Layered architecture
   - **c.** Both (a) and (b)
   - **d.** Agent architecture
4. A predicate expression associated with an event is known as
   - **a.** Transition
   - **b.** Class
   - **c.** Guard
   - **d.** None of the above.
5. Statecharts refers to
   - **a.** State diagrams + depth + orthogonality + broadcast communication
   - **b.** State diagrams only
   - **c.** State diagrams + depth
   - **d.** None of the above.

6. Which of the following is OOA/OOD method?
   - **a.** OMT
   - **b.** ROOM
   - **c.** Fusion
   - **d.** All of the above.
7. The formation of a subclass that has no locally defined features other than the minimum requirement of a class definition is called
   - **a.** Concatenation
   - **b.** Binding
   - **c.** Polymorphism
   - **d.** None of the above.
8. Testing of nested loops was suggested by
   - **a.** Little wood
   - **b.** Beizer
   - **c.** Boehm
   - **d.** None of the above.
9. An accepted industry standard for unit testing is
   - **a.** IEEE 83b
   - **b.** IEEE 87a
   - **c.** IEEE 9126
   - **d.** None of the above.
10. A powerful technique to use any time a system presents a small input domain with a large number of permutations is known as
    - **a.** Regression testing
    - **b.** Object-oriented testing
    - **c.** Orthogonal array testing
    - **d.** None of the above.

ANSWERS

1. a.	**2.** a.	**3.** c.	**4.** c.
5. a.	**6.** d.	**7.** a.	**8.** b.
9. b.	**10.** c.		

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

Q. 1. What is LCOM metric?

Ans. The LCOM metric is defined as a count of the method pairs that do not have common instance variable minus the count of method pairs that do. *Please note that the larger the number of similar methods, the more cohesive is the class.*

LCOM (fopr a class) = 0 if none of the methods of a class display any instance behavior, i.e., do not use any instance variables. Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

Q. 2. How do we compute the component weightage (CW) of an object-oriented software?

Ans. We try to compute the CWs of individual design attributes like encapsulation, inheritance, coupling, and cohesion to show the complexity relationship with the design properties. That is,

Complexity = [0.16 * Encapsulation] + [2.78 * Inheritance] + [1.57 * Coupling]

Q. 3. Explain the two main variants of state models?

Ans. **(a) Moore Machine:**

- Transitions do not have output.
- An output action is associated with each state.
- States are active.

(b) Mealy Machine:

- Transitions have output.
- No output action is associated with state.
- States are passive.

Q. 4. The big bang is estimated to have occurred about 18 billion years ago.

Given: Paths/second = 1×10^3

Second/year = 3.154×10^7

If we start testing 10^3 paths/second at instant of the big bang, how many paths could we test? At what loop value of x, would we run out of time?

Ans. Given: 18 billion years = 1.8×10^9 years

and 3.154×10^7 seconds/year

We have, 1.8×10^9 years * 3.154×10^7 seconds

$= 5.6772 \times 10^{16}$ seconds (since the big bang)

If we test 10^3 paths/second at the instant of big bang, we could test

$= 5.6772 \times 10^{16} \times 10^3$ paths

$= 5.6772 \times 10^{19}$ paths

Also, we find x = ?

Now, $2^x = 5.6772 \times 10^{19}$

Taking log on both sides, we get:

$\log(2^x) = \log(5.6772 \times 1019)$

or $x \log 2 = 19 \log(5.6772)$

$x(0.3010) = 19 \times 0.754134$

or $x = 3.321928 \times 19 \times 0.754134$

$\therefore$ $x = 35.89559$

Q. 5. What is pesticide paradox?

Ans. As defined by Beizer,

1. The purpose of a test process is to find faults.
2. An effective process will eventually find fewer faults.
3. By definition, this process becomes increasingly ineffective, i.e., its ability to find faults decreases because the initial fault sources have been eliminated.
4. Generally, applications, platforms, and development tools become more complex, creating new opportunities for errors.

This is called the pesticide paradox.

For example, a given pesticide (P1) kills 98% of the first generation of bugs B1, having survivors S1. The 2% that survive are immune to P1. You have helped the survivors by reducing their competition. The survivors regenerate and become generation B2. P1 is useless against the second generation, B2.

Moral: Process monitoring is important to identify both chronic and emerging fault causes.

REVIEW QUESTIONS

1. Object-oriented languages like JAVA do not support pointers. This makes testing easier. Explain how.
2. Consider a nested class. Each class has one method. What kind of problems will you encounter during testing of such nested classes? What about their objects?
3. Explain the following:
   a. Unit and integration testing
   b. Object-oriented testing
4. Write and explain some common inheritance related bugs.
5. How is object-oriented testing different from procedural testing? Explain with examples.
6. Describe all the methods for class testing.
7. Write a short paragraph on issues in object-oriented testing.
8. Explain briefly about object-oriented testing methods with examples. Suggest how you test object-oriented systems by use-case approach.
9. Illustrate "how do you design interclass test cases." What are the various testing methods applicable at the class level?
10. a. What are the implications of inheritance and polymorphism in object-oriented testing?
    b. How does GUI testing differ from normal testing? How is GUI testing done?
11. With the help of suitable examples, demonstrate how integration testing and system testing is done for object-oriented systems?
12. How reusability features can be exploited by object-oriented testing approach?
13. a. Discuss the salient features of GUI testing. How is it different from class testing?
    b. Explain the testing process for object-oriented programs (systems).
14. Draw a state machine model for a two-player game and also write all possible control faults from the diagram.

CHAPTER 14

THE GAME TESTING PROCESS[1]

Inside this Chapter:

14.0. "Black-Box" Testing
14.1. "White-Box" Testing
14.2. The Life Cycle of a Build
14.3. On Writing Bugs Well

Developers don't fully test their own games. They don't have time to, and even if they did, it's not a good idea. Back at the dawn of the video game era, the programmer of a game was also its artist, designer, and tester. Even though games were very small—the size of email— the programmer spent most of his time designing and programming. Little of his time was spent testing. If he did any testing, it was based on his own assumptions about how players would play his game. The following sidebar illustrates the type of problem these assumptions could create.

THE PLAYER WILL ALWAYS SURPRISE YOU

The programmer of *Astrosmash,* a space shooter released for the Intellivision® system in 1981, made an assumption when he designed the game that no player would ever score 10 million points. As a result, he didn't write a check for score overflowing. He read over his own code and—based on his own assumptions—it seemed to work fine. It was a fun game—its graphics were

[1] This chapter appeared in *Game Testing*, *Third Edition*, C. Schultz and R. D. Bryant. Copyright 2017 Mercury Learning and Information. All rights reserved.

breathtaking (for the time) and the game went on to become one of the best sellers on the Intellivision platform.

Weeks after the game was released, however, a handful of customers began to call the game's publisher, Mattel Electronics, with an odd complaint: when they scored more than 9,999,999 points, the score displayed negative numbers, letters, and symbol characters. This in spite of the promise of "unlimited scoring potential" in the game's marketing materials. The problem was exacerbated by the fact that the Intellivision console had a feature that allowed players to play the game in slow motion, making it much easier to rack up high scores. John Sohl, the programmer, learned an early lesson about video games: *the player will always surprise you*.

The sidebar story demonstrates why video game testing is best done by testers who are: (a) professional, (b) objective, and (c) separated—either physically or functionally—from the game's development team. That remove and objectivity allows testers to think independently of the developers, to function as players, and to figure out new and interesting ways to break the game. This chapter discusses how, like the gears of a watch, the game *testing* process meshes into the game *development* process.

14.0. "BLACK-BOX" TESTING

Almost all game testing is *black-box* testing, testing done from outside the application. No knowledge of, or access to, the source code is granted to the tester. Game testers typically don't find defects by reading the game code. Rather, they try to find defects using the same input devices available to the average player, be it a mouse, a keyboard, a console gamepad, a motion sensor, or a plastic guitar. Black-box testing is the most cost-effective way to test the extremely complex network of systems and modules that even the simplest video game represents.

Figure 14.1 illustrates some of the various inputs you can provide to a videogame and the outputs you can receive back. The most basic of inputs are positional, and control data in the form of button presses and cursor movements, or vector inputs from accelerometers, or even full-body cameras. Audio input can come from microphones fitted in headsets or attached to a game controller. Input from other players can come from a second controller, a local network, or the Internet. Finally, stored data such as saved games and options settings can be called up as input from memory cards or a hard drive.

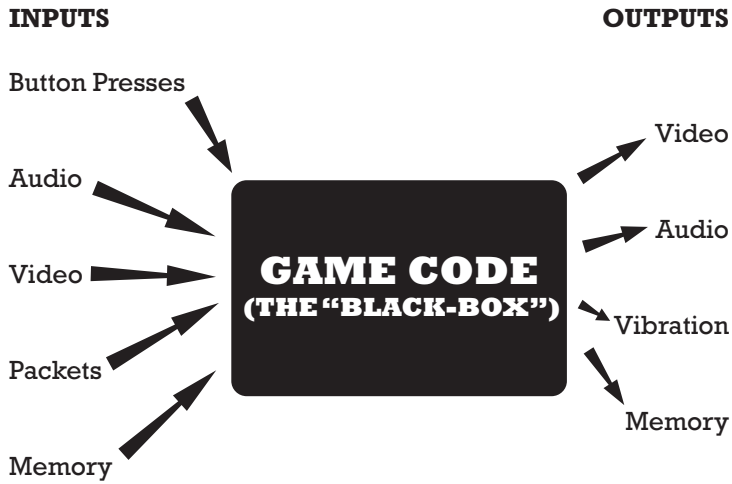


FIGURE 14.1 Black-box testing: planning inputs and examining outputs.

Once some or all of these types of input are received by the game, it reacts in interesting ways and produces such output as video, audio, vibration (via force feedback devices), and data saved to memory cards or hard drives.

The input path of a video game is not oneway, however. It is a feedback loop, where the player and the game are constantly reacting to each other. Players don't receive output from a game and stop playing. They constantly alter and adjust their input "on the fly," based on what they see, feel, and hear in the game. The game, in turn, makes similar adjustments in its outputs based on the inputs it receives from the player. Figure 14.2 illustrates this loop.

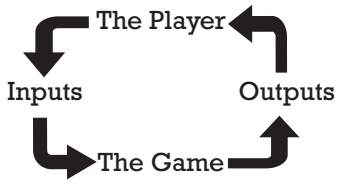


FIGURE 14.2 The player's feedback loop adjusts to the game's input, and vice versa.

If the feedback received by the player were entirely predictable all the time, the game would be no fun. Nor would it be fun if the feedback received by the player were entirely random all the time. Instead, feedback from games should be just random enough to be unpredictable. It is the unpredictability of the feedback loop that makes games fun. Because the code is designed to surprise the player and the player will always surprise the programmer, black-box testing allows testers to think and behave like players.

14.1. "WHITE-BOX" TESTING

In contrast to black-box testing, *white-box* testing gives the tester opportunities to exercise the source code directly in ways that no player ever could. It can be a daunting challenge for the white-box tester to read a piece of game code and predict every single interaction it will have with every other bit of code, and whether the programmer has accounted for every combination and order of inputs possible. Testing a game using only white-box methods is also extremely difficult because it is nearly impossible to account for the complexity of the player feedback loop. There are, however, situations in which white-box testing is more practical and necessary than black-box testing. These include the following:

- Tests performed by developers prior to submitting new code for integration with the rest of the game
- Testing code modules that will become part of a reusable library across multiple games or platforms
- Testing code methods or functions that are essential parts of a game engine or middleware product
- Testing code modules within your game that might be used by thirdparty developers or "modders" who, by design, could expand or modify the behavior of your game to their own liking
- Testing low-level routines that your game uses to support specific functions in the newest hardware devices, such as graphics cards or audio processors

In performing white-box tests, you execute specific modules and the various paths that the code can follow when you use the module in various ways. Test inputs are determined by the types and values of data that can be passed to the code. Results are checked by examining values returned by the module, global variables that are affected by the module, and local variables as they are processed within the module. To get a taste of white-box testing, consider the TeamName routine from *Castle Wolfenstein: Enemy Territory:*

```
const char *TeamName(int team)   {
      if (team==TEAM_AXIS)
            return "RED";
      else if (team==TEAM_ALLIES)
            return "BLUE";
      else if (team==TEAM_SPECTATOR)
            return "SPECTATOR";
      return "FREE";
}
```

Four white-box tests are required for this module to test the proper behavior of each line of code within the module. The first test would be to call the TeamName function with the parameter `TEAM_AXIS` and then check that the string "`RED`" is returned. Second, pass the value of `TEAM_ALLIES` and check that "`BLUE`" is returned. Third, pass `TEAM_SPECTATOR` and check that "`SPECTATOR`" is returned. Finally, pass some other value such as `TEAM_NONE`, which makes sure that "`FREE`" is returned. Together these tests not only exercise each line of code at least once, they also test the behavior of both the "true" and "false" branches of each `if` statement.

This short exercise illustrates some of the key differences between a white-box testing approach and a black-box approach:

- Black-box testing should test all of the different ways you could choose a test value from within the game, such as different menus and buttons. White-box testing requires you to pass that value to the routine in one form—its actual symbolic value within the code.
- By looking into the module, white-box testing reveals all of the possible values that can be provided to and processed by the module being tested. This information might not be obvious from the product requirements and feature descriptions that drive black-box testing.
- Black-box testing relies on a consistent configuration of the game and its operating environment in order to produce repeatable results. White-box testing relies only on the interface to the module being tested and is concerned only about external files when processing streams, file systems, or global variables.

14.2. THE LIFE CYCLE OF A BUILD

Game testers are often frustrated that, like players, they must wait (and wait) for the work product of the development team before they can spring into action. Players wait for game releases; testers wait for code releases, or *builds.* The test results from each build is how all stakeholders in the project—from QA to the project manager to the publisher—measure the game's progress toward release.

A basic game testing process consists of the following steps:

1. **Plan and design the test.** Although much of this is done early in the planning phase, planning and design should be revisited with every build. What has changed in the design spec since the last build? What additional test cases have been added? What new configurations will

the game support? What features have been cut? The scope of testing should ensure that no new issues were introduced in the process of fixing bugs prior to this release.

2. **Prepare for testing.** Code, tests, documents, and the test environment are updated by their respective owners and aligned with one another. By this time the development team should have marked the bugs fixed for this build in the defect database so the QA team can subsequently verify those fixes and close the bugs.
3. **Perform the test.** Run the test suites against the new build. If you find a defect, test "around" the bug to make certain you have all the details necessary to write as specific and concise a bug report as possible. The more research you do in this step, the easier and more useful the bug report will be.
4. **Report the results.** Log the completed test suite and report any defects you found.
5. **Repair the bug.** The test team participates in this step by being available to discuss the bug with the development team and to provide any directed testing a programmer might require to track the defect down.
6. **Return to Step 1 and re-test.** With new bugs and new test results comes a new build.

These steps not only apply to black-box testing, they also describe white-box testing, configuration testing, compatibility testing, and any other type of QA. These steps are identical no matter what their scale. If you substitute the word "game" or "project" for the word "build" in the preceding steps, you will see that they can also apply to the entire game, a phase of development (Alpha, Beta, and so on), or an individual module or feature within a build. In this manner, the software testing process can be considered fractal—the smaller system is structurally identical to the larger system, and vice versa.

As illustrated in Figure 14.3, the testing process itself is a feedback loop between the tester and developer. The tester plans and executes tests on the code, then reports the bugs to the developer, who fixes them and compiles a new build, which the tester plans and executes, and so on.

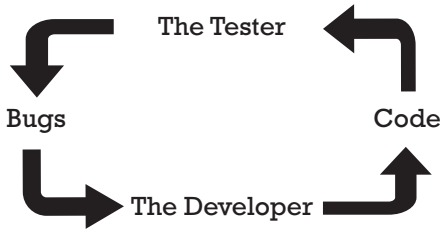


FIGURE 14.3 The testing process feedback loop.

A comfortable scale from which to examine this process is at the level of testing an individual build. Even a relatively small game project could consist of dozens of builds over its development cycle.

Test Cases and Test Suites

As discussed in the previous chapter, a single test performed to answer a single question is a test case; a collection of test cases is a test suite. The lead tester, primary tester, or any other tester tasked with test creation should draft these documents prior to the distribution of the build. Each tester will take his or her assigned test suites and perform them on the build. Any anomalies not already present in the database should be written up as new bugs.

In its simplest form, a test suite is a series of incremental steps that the tester can perform sequentially. Subsequent chapters in this book discuss in depth the skillful design of test cases and suites through such methods as combinatorial tables and test flow diagrams. For the purposes of this discussion, consider a short test suite you might execute on *Minesweeper,* a simple game available with most versions of Microsoft Windows®. A portion of this suite is shown in Figure 14.4.

Step	Pass	Fail	Comments
1. Launch Minesweeper			
2. Musical tone plays?			
3. Visible menu options are Game and Help?			
4. Right Number (time elapsed) displayed as 0?			
5. Left Number (bombs left) displayed is 10?			
6. Click Game on the menu and choose Exit.			
7. Game closes?			
8. Re-launch Minesweeper.			
9. Choose Game > Options > Custom			
10. Enter 0 in the Height box			
11. 0 accepted as input?			
12. Click OK.			
13. Error message appears?			
14. Click OK again.			
15. Game grid 9 rows high?			
16. Game grid 9 columns wide (unchanged)?			
17. Choose Game > Options > Custom			
18. Enter 999 in the Height box			
19. 999 Accepted as input?			
20. Click OK.			
21. Playing grid 24 rows high?			
22. Playing grid 9 columns wide (unchanged)?			

FIGURE 14.4 A portion of a test suite for *Minesweeper*.

This is a very small portion of a very simple test suite for a very small and simple game. The first section (steps one through seven) tests launching the game, ensuring that the default display is correct, and exiting. Each step either gives the tester one incremental instruction or asks the tester one simple question. Ideally, these questions are binary and unambiguous. The tester performs each test case and records the result.

Because the testers will inevitably observe results that the test designer hadn't planned for, the Comments field allows the tester to elaborate on a Yes/No answer, if necessary. The lead or primary tester who receives the completed test suite can then scan the Comments field and make adjustments to the test suite as needed for the next build.

Where possible, the questions in the test suite should be written in such a way that a "yes" answer indicates a "pass" condition—the software is working as designed and no defect is observed. "No" answers, in turn, should indicate that there is a problem and a defect should be reported. There are several reasons for this: it's more intuitive, because we tend to group "yes" and "pass" (both positives) together in our minds the same way we group "no" and "fail." Further, by grouping all passes in the same column, the completed test suite can be easily scanned by both the tester and test managers to determine quickly whether there were any fails. A clean test suite will have all the checks in the Pass column.

For example, consider a test case covering the display of a tool tip—a small window with instructional text incorporated into many interfaces. A fundamental test case would be to determine whether the tool tip text contains any typographical errors. The most intuitive question to ask in the test case is:

```
Does the text contain any typographical errors?
```

The problem with this question is that a pass (no typos, hence no bugs) would be recorded as a "no." It would be very easy for a hurried (or tired) tester to mistakenly mark the Fail column. It is far better to express the question so that a "yes" answer indicates a "pass" condition:

```
Is the text free of typographical errors?
```

As you can see, directed testing is very structured and methodical. After the directed testing has concluded, or concurrently with directed testing, a less structured, more intuitive form of testing, known as *ad hoc* testing, takes place.

Entry Criteria

It's advisable to require that any code release meets some criteria for being fit to test before you risk wasting your time, or your team's time, testing it. This is similar to the checklists that astronauts and pilots use to evaluate the fitness of their vehicle systems before attempting flight. Builds submitted to testing that don't meet the basic entry criteria are likely to waste the time of both testers and programmers. The countdown to testing should stop until the test "launch" criteria are met.

The following is a list of suggestions for entry criteria. Don't keep these a secret from the rest of the development team. Make the team aware of the purpose—to prevent waste—and work with them to produce a set of criteria that the whole team can commit to.

- The game code should be built without compiler errors. Any new compiler warnings that occur are analyzed and discussed with the test team.
- The code release notes should be complete and should provide the detail that testers need to plan which tests to run or to re-run for this build.
- Defect records for any bugs closed in the new release should be updated so they can be used by testers to make decisions about how much to test in the new build.
- Tests and builds should be properly version-controlled, as described in the sidebar, "Version Control: Not Just for Developers."
- When you are sufficiently close to the end of the project, you also want to receive the game on the media on which it will ship. Check that the media provided contains all of the files that would be provided to your customer.

VERSION CONTROL: NOT JUST FOR DEVELOPERS

A fundamental principle of software development is that every build of an application should be treated as a separate and discrete version. Inadvertent blending of old code with new is one of the most common (and most preventable) causes of software defects. The process of tracking builds and ensuring that all members of a development team are checking current code and assets into the current version is known as *version control*.

Test teams must practice their own form of version control. There are few things more time wasting than for a test team to report a great number

of bugs in an old build. This is not only a waste of time, but it can cause panic on the part of the programmer and the project manager.

Proper version control for the test team includes the following steps:

1. Collect all prior physical (*e.g.,* disk-based) builds from the test team before distributing the new build. The prior versions should be staked together and archived until the project is complete. (When testing digital downloads, uninstall and delete or archive prior digital builds.)
2. Archive all paperwork. This includes not only any build notes you received from the development team, but also any completed test suites, screen shots, saved games, notes, video files, and any other material generated during the course of testing a build. It is sometimes important to retrace steps along the paper trail, whether to assist in isolating a new defect or determining in what version an old bug was re-introduced.
3. Verify the build number with the developer prior to distributing it.
4. In cases where builds are transmitted electronically, verify the byte count, file dates, and directory structure before building it. It is vital in situations where builds are sent via FTP, email, Dropbox (*www.dropbox.com*) or other digital means that the test team makes certain to test a version identical to the version the developers uploaded. Confirm the integrity of the transmitted build before distributing it to the testers.
5. Renumber all test suites and any other build-specific paperwork or electronic forms with the current version number.
6. Distribute the new build for smoke testing.

Configuration Preparation

Before the test team can work with the new build, some housekeeping is in order. The test equipment must be readied for a new round of testing. The test lead must communicate the appropriate hardware configuration to each tester for this build. Configurations typically change little over the course of game testing. To test a single-player-only console game, you need the game console, a controller, and a memory card or hard drive. That hardware configuration typically will not change for the life of the project. If, however, the new build is the first in which network play is enabled, or a new input device or PC video card has been supported, you will perhaps need to augment the hardware configuration to perform tests on that new code.

Perhaps the most important step in this preparation is to eliminate any trace of the prior build from the hardware. "Wiping" the old build of a disk-based game on a Nintendo Wii™ is simple, because the only recordable media for that system is an SD card or its small internal flash memory drive. All you have to do is remove and archive the saved game you created with the hold build. More careful test leads will ask their testers to go the extra step of reformatting the media, which completely erases it, to ensure that not a trace of the old build's data will carry forward during the testing of the new build.

TIP

Save your saves! Always archive your old player-created data, including game saves, options files, and custom characters, levels, or scenarios.

Not surprisingly, configuration preparation can be much more complicated for PC games. The cleanest possible testing configuration for a PC game is:

- A fresh installation of the latest version of the operating system, including any patches or security updates.
- The latest drivers for all components of the computer. This not only includes the obvious video card and sound card drivers, but also chipset drivers, motherboard drivers, Ethernet card drivers, WiFi® firmware, and so on.
- The latest version of any "helper apps" or middleware the game requires in order to run. These can range from Microsoft's DirectX® multimedia libraries to third-party multiplayer matchmaking software.

The only other software installed on the computer should be the new build.

CHASING FALSE DEFECTS

We once walked into a QA lab that was testing a (then) very cutting-edge 3D PC game. Testing of the game had fallen behind, and we had been sent from the publisher to investigate. We arrived just as the testers were breaking for lunch, and were appalled to see the testers exit the game they were testing and fire up email, instant messenger clients, Web browsers, and file sharing programs—a host of applications that were installed on their test

computers. Some even jumped into a game of *Unreal Tournament*. We asked the assistant test manager why he thought it was a good idea for the testers to run these extraneous programs on their testing hardware. "It simulates real-world conditions," he shrugged, annoyed by our question.

As you perhaps have already guessed, this lab's failure to wipe their test computers clean before each build led to a great deal of wasted time chasing false defects—symptoms testers thought were defects in the game, but which were in fact problems brought about by, for example, email or file sharing programs running in the background, taxing the system's resources and network bandwidth. This wasted tester time also meant a good amount of wasted programmer time, as the development team tried to figure out what in the game code might be causing such (false) defects.

The problem was solved by reformatting each test PC, freshly installing the operating system and latest drivers, and then using a drive image backup program to create a system restore file. From that point forward, testers merely had to reformat their hard drive and copy the system restore file over from a CD-ROM.

Testing takes place in the lab and labs should be clean. So should test hardware. It's difficult to be too fastidious or paranoid when preparing test configurations. When you get a new build, reformat your PC rather than merely uninstall the new build.

TIP *Delete your old builds! Reformat your test hardware—whether it's a PC, a tablet or a smartphone. If it's a browser game, delete the cache.*

Browser games should be purged from each browser's cache and the browser should be restarted before you open the new game build. In the case of Flash® games, you can right-click on the old build and select "Global Settings…" This will launch a separate browser process and will connect you to the Flash Settings Manager. Choosing the "Website Storage Settings panel" will launch a Flash applet. Click the "Delete all sites" button and close all of your browser processes. Now you can open the new build of your Flash game.

iOS™ games should be deleted both from the device and the iTunes® client on the computer the device is synched to. When prompted by iTunes, choose to delete the app entirely (this is the "Move to Recycle Bin" or "Move to Trash" button). Now, synch your device and make certain the old build has been removed both from iTunes and your device. Empty the Recycle Bin (or the Trash), relaunch iTunes, copy the new build, and synch your device again.

Android™ games, like iOS games, should be deleted entirely from the device and your computer. Always synch your device to double-check that you have scrubbed the old build off before you install the new build.

Whatever protocol is established, *config prep* is crucial prior to the distribution of a new build.

Smoke Testing

The next step after accepting a new build and preparing to test it is to certify that the build is worthwhile to submit to formal testing. This process is sometimes called *smoke testing,* because it's used to determine whether a build "smokes" (malfunctions) when run. At a minimum, it should consisted of a "load & launch," that is, the lead or primary tester should launch the game, enter each module from the main menu, and spend a minute or two playing each module. If the game launches with no obvious performance problems and each module implemented so far loads with no obvious problems, it is safe to certify the build, log it, and duplicate it for distribution to the test team.

Now that the build is distributed, it's time to test for new bugs, right? Not just yet. Before testing can take a step forward, it must first take a step backward and verify that the bugs the development team claims to have fixed in this build are indeed fixed. This process is known as *regression testing.*

Regression Testing

Fix verification can be at once very satisfying and very frustrating. It gives the test team a good sense of accomplishment to see the defects they report disappear one by one. It can be very frustrating, however, when a fix of one defect creates another defect elsewhere in the game, as can often happen.

The test suite for regression testing is the list of bugs the development team claims to have fixed. This list, sometimes called a *knockdown list,* is ideally communicated through the bug database. When the programmer or artist fixes the defect, all they have to do is change the value of the Developer Status field to "Fixed." This allows the project manager to track the progress on a minute-to-minute basis. It also allows the lead tester to sort the regression set (by bug author or by level, for example). At a minimum, the knockdown list can take the form of a list of bug numbers sent from the development team to the lead tester.

TIP

Don't accept a build into test unless it is accompanied by a knockdown list. It is a waste of the test team's time to regress every open bug in the database every time a new build enters test.

Each tester will take the bugs they've been assigned and perform the steps in the bug write-up to verify that the defect is indeed fixed. The fixes for many defects are easily verified (typos, missing features, and so on). Some defects, such as hard-to-reproduce crashes, could *seem* fixed, but the lead tester might want to err on the side of caution before he closes the bug. By flagging the defect as *verify fix*, the bug can remain in the regression set (i.e., stay on the knockdown list) for the next build (or two), but out of the set of open bugs that the development team is still working on. Once the bug has been verified as fixed in two or three builds, the lead tester can then close the bug with more confidence.

At the end of regression testing, the lead tester and project manager can get a very good sense of how the project is progressing. A high fix rate (number of bugs closed divided by the number of bugs claimed to have been fixed) means the developers are working efficiently. A low fix rate could be cause for concern. Are the programmers arbitrarily marking bugs as fixed if they think they've implemented new code that might address the defect, rather than troubleshooting the defect itself? Are the testers not writing clear bugs? Is there a version control problem? Are the test systems configured properly? While the lead tester and project manager mull over these questions, it's time for you to move on to the next step in the testing process: performing structured tests and reporting the results.

Testing "Around" a Bug

The old saying in carpentry is "measure twice, cut once." Good game testers thoroughly investigate a defect before they write it up, anticipating any questions the development team might have.

Before you begin to write a defect report, ask yourself some questions:

1. Is this the only location or level where the bug occurs?
2. Does the bug occur while using other characters or units?
3. Does the bug occur in other game modes (for example, multiplayer as well as single player, skirmish as well as campaign)?
4. Can I eliminate any steps along the path to reproducing the bug?
5. Does the bug occur across all platforms (for example, does it occur on both the Xbox One and PlayStation 4 builds)?
6. Is the bug machine-specific (for example, does it occur only on PCs with a certain hardware configuration)?

These are the types of questions you will be asked by the lead tester, project manager, or developer. Try to develop the habit of second-guessing such questions by performing some quick additional testing before you write the bug. Test to see whether the defect occurs in other areas. Test to determine whether the bug happens when you choose a different character. Test to check which other game modes contain the issue. This practice is known as testing "around" the bug.

Once you are satisfied that you have anticipated any questions that the development team might ask, and you have all your facts ready, you are finally ready to write the bug report.

14.3. ON WRITING BUGS WELL

Good bug writing is one of the most important skills a tester must learn. A defect can be fixed only if it is communicated clearly and effectively. One of the oldest jokes in software development goes something like this:

Q: How many programmers does it take to screw in a light bulb?

A: None—it's not dark where they're sitting.

Good bug report writing gets the development team to "see the light" of the bug. The developers are by no means the only people who will read your bug, however. Your audience could include:

- The lead tester or primary tester, who might wish to review the bug before she gives it an "open" status in the bug database.
- The project manager, who will read the bug and assign it to the appropriate member of the development team.
- Marketing and other business executives, who might be asked to weigh in on the possible commercial impact of fixing (or not fixing) the bug.
- Third parties, such as middleware developers, who could be asked to review a bug that is possibly related to a project they supply to the project team
- Customer service representatives, who might be asked to devise workarounds for the bug
- Other testers, who will reproduce the steps if they are asked to verify a fix during regression testing

Because you never know exactly who will be reading your bug report, you must always write in as clear, objective, and dispassionate a manner as

possible. You can't assume that everyone reading your bug report will be as familiar with the game as you are. Testers spend more time in the game—exploring every hidden path, closely examining each asset—than almost anyone else on the entire project team. A well-written bug will give a reader who is not familiar with the game a good sense of the type and severity of the defect it describes.

Just the Facts, Ma'am

The truth is that defects stress out development teams, especially during "crunch time." Each new bug added to the database means more work still has to be done. An average-sized project can have hundreds or thousands of defects reported before it is completed. Developers can feel overwhelmed and might, in turn, get hostile if they feel their time is being wasted by frivolous or arbitrary bugs. That's why good bug writing is fact based and unbiased.

```
The guard's hat would look better if it was blue.
```

This is neither a defect nor a fact; it's an unsolicited and arbitrary opinion about design. There are forums for such opinions—discussions with the lead tester, team meetings, play testing feedback—but the bug database isn't one of them.

A common complaint in many games is that the artificial intelligence, or AI, is somehow lacking. (AI is a catch-all term that means any opponents or NPCs controlled by the game code.)

```
The AI is weak.
```

This could indeed be a fact, but it is written in such a vague and general way that it is likely to be considered an opinion. A much better way to convey the same information is to isolate and describe a specific example of AI behavior and write up that specific defect. By boiling issues down to specific facts, you can turn them into defects that have a good chance of being addressed.

TIP *Before you begin to write a bug report, you have to be certain that you have all your facts.*

Brief Description

Larger databases could contain two description fields: Brief Description (or Summary) and Full Description (or Steps). The Brief Description field is used as a quick reference to identify the bug. This should not be a cute nickname, but a one-sentence description that allows team members to identify and discuss the defect without having to read the longer, full description each time. Think of the brief description as the headline of the defect report.

```
Crash to desktop.
```

This is not a complete sentence, nor is it specific enough to be a brief description. It could apply to one of dozens of defects in the database. The brief description must be brief enough to be read easily and quickly, but long enough to describe the bug.

```
The saving system is broken.
```

This is a complete sentence, but it is not specific enough. What did the tester experience? Did the game not save? Did a saved game not load? Does saving cause a crash?

```
Crash to desktop when choosing "Options" from Main Menu.
```

This is a complete sentence, and it is specific enough so that anyone reading it will have some idea of the location and severity of the defect.

```
Game crashed after I killed all the guards and
doubled back through the level to get all the
pick-ups and killed the first re-spawned guard.
```

This is a run-on sentence that contains far too much detail. A good way to boil it down might be

```
Game crashes after killing respawned guards.
```

The one-sentence program descriptions used by cable television guides and download stores can provide excellent examples of brief description writing—they boil an entire one-hour cop show or two-hour movie into one sentence.

TIP *Write the full description first, and then write the brief description. Spending some time polishing the full description will help you understand the most important details to include in the brief description.*

Full Description

If the brief description is the headline of a bug report, the Full Description field provides the gory details. Rather than a prose discussion of the defect, the full description should be written as a series of brief instructions so that anyone can follow the steps and reproduce the bug. Like a cooking recipe—or computer code, for that matter—the steps should be written in second person imperative, as though you were telling someone what to do. The last step is a sentence (or two) describing the bad result.

```
1. Launch the game.
2. Watch the animated logos. Do not press ESC to
   skip through them.
--> Notice the bad strobing effect at the end of
    the Developer logo.
```

The fewer steps, the better; and the fewer words, the better. Remember Brad Pitt's warning to Matt Damon in *Ocean's Eleven:* don't use seven steps when four will do. Time is a precious resource when developing a game. The less time it takes a programmer to read, reproduce, and understand the bug, the more time he has to fix it.

```
1. Launch game.
2. Choose multiplayer.
3. Choose skirmish.
4. Choose "Sorrowful Shoals" map.
5. Choose two players.
6. Start game.
```

These are very clear steps, but for the sake of brevity they can be boiled down to

```
1. Start a two player skirmish game on "Sorrowful Shoals."
```

Sometimes, however, you need several steps. The following bug describes a problem with a power-up called "mugging," which steals any other power-up from any other unit.

```
1. Create a game against one human player. Choose
   Serpent tribe.
2. Send a swordsman into a Thieves Guild to get the
   Mugging power-up.
3. Have your opponent create any unit and give
   that unit any power-up.
```

```
4. Have your Swordsman meet the other player's
   unit somewhere neutral on the map.
5. Activate the Mugging power-up.
6. Attack your opponent's unit.
--> Crash to desktop as Swordsman strikes.
```

This might seem like many steps, but it is the quickest way to reproduce the bug. Every step is important to isolate the behavior of the mugging code. Even small details, like meeting in a neutral place, are important, because meeting in occupied territory might bring allied units from one side or another into the fight, and the test might then be impossible to perform.

TIP *Good bug writing is* precise *yet* concise.

Great Expectations

Oftentimes, the defect itself will not be obvious from the steps in the full description. Because the steps produce a result that deviates from player expectation, but does not produce a crash or other severe or obvious symptom, it is sometimes necessary to add two additional lines to your full description: Expected Result and Actual Result.

FIGURE 14.5 *Fallout 4*: One would expect player-placed structures to appear grounded on the terrain rather than floating above it.

Expected Result describes the behavior that a normal player would reasonably expect from the game if the steps in the bug were followed. This expectation is based on the tester's knowledge of the design specification, the target audience, and precedents set (or broken) in other games, especially games in the same genre.

Actual Result describes the defective behavior. Here's an example.

```
1. Create a multiplayer game.
2. Click Game Settings.
3. Using your mouse, click any map on the map list.
   Remember the map you clicked on.
4. Press up or down directional keys on your keyboard.
5. Notice the highlight changes. Highlight any other
   map.
6. Click Back.
7. Click Start Game.
Expected Result: Game loads map you chose with the keyboard.
Actual Result: Game loads map you chose with the mouse.
```

Although the game loaded a map, it wasn't the map the tester chose with the keyboard (the last input device he used). That's a bug, albeit a subtle one. Years of precedent creates the expectation in the player's mind that the computer will execute a command based on the last input the player gave. Because the map-choosing interface failed to conform to player expectation and precedent, it could be confusing or annoying, so it should be written up as a bug.

Use the Expected/Actual Result steps sparingly. Much of the time, defects are obvious (see Figure 14.5) Here's an example of "stating the obvious" in a crash bug.

```
4. Choose "Next" to continue.
Expected Result: You continue.
Actual Result: Game locks up. You must reboot the console.
```

It is understood by all members of the project team that the game shouldn't crash. Don't waste time and space stating that with an unnecessary statement of Expected and Actual Results.

You should use these statements sparingly in your bug reports, but you should use them when necessary. They can often make a difference when a developer wants to close a bug in the database by declaring it "by design," "working as intended," or "NAB" (Not a Bug).

INTERVIEW

More players are playing games than ever before. As any human population grows—and the pool of game players has grown exponentially over the last decade—that population becomes more diverse. Players are different from each other, have different levels of experience with games, and play games for a range of different reasons. Some players want a competitive experience, some an immersive experience, some want a gentle distraction.

The pool of game testers in any organization is always less diverse than the player base of the game they are testing. Game testers are professionals, they have skills in manipulating software interfaces, they are generally (but not necessarily) experienced game players. It's likely that if your job is creating games, that you've played video games—a lot of them. But not every player is like you.

Brent Samul, QA Lead for developer Mobile Deluxe, put it this way: "The biggest difference when testing for mobile is your audience. With mobile you have such a broad spectrum of users. Having played games for so long myself, it can sometimes be really easy to overlook things that someone who doesn't have so much experience in games would get stuck on or confused about."

It's a big job. "With mobile, we have the ability to constantly update and add or remove features from our games. There are always multiple things to test for with all the different configurations of smartphones and tablets that people have today," Mr. Samul says.

Although testers should write bugs against the design specification, the authors of that specification are not omniscient. As the games on every platform become more and more complex, it's the testers' job to advocate for the players—all players—in their bug writing. (Permission Brent Samul)

Habits to Avoid

For the sake of clarity, effective communication, and harmony among members of the project team try to avoid two common bug writing pitfalls: humor and jargon.

Although humor is often welcome in high-stress situations, it is not welcome in the bug database. Ever. There are too many chances for misinterpretation and confusion. During crunch time, tempers are short, skins are thin, and nerves are frayed. The defect database could already be a point of contention. Don't make the problem worse with attempts at humor (even if

you think your joke is hilarious). Finally, as the late William Safire warned, you should "avoid clichés like the plague."

It perhaps seems counterintuitive to want to avoid jargon in such a specialized form of technical writing as a bug report, but it is wise to do so. Although some jargon is unavoidable, and each project team quickly develops it own nomenclature specific to the project they're working on, testers should avoid using (or misusing) too many obscure technical terms or acronyms. Remember that your audience could range from programmers to financial or marketing executives, so use plain language as much as possible.

Although testing build after build might seem repetitive, each new build provides exciting new challenges with its own successes (fixed bugs and passed tests) and shortfalls (new bugs and failed tests). The purpose of going about the testing of each build in a structured manner is to reduce waste and to get the most out of the game team. Each time around, you get new build data that is used to re-plan test execution strategies and update or improve your test suites. From there, you prepare the test environment and perform a smoke test to ensure the build is functioning well enough to deploy to the entire test team. Once the test team is set loose, your top priority is typically regression testing to verify recent bug fixes. After that, you perform many other types of testing in order to find new bugs and to check that old ones have not re-emerged. New defects should be reported in a clear, concise, and professional manner after an appropriate amount of investigation. Once you complete this journey, you are rewarded with the opportunity to do it all over again.

EXERCISES

1. Briefly describe the difference between the Expected Result and the Actual Result in a bug write-up.
2. What's the purpose of regression testing?
3. Briefly describe the steps in preparing a test configuration.
4. What is a "knockdown list"? Why is it important?
5. True or False: Black-box testing refers to examining the actual game code.
6. True or False: The Brief Description field of a defect report should include as much information as possible.

7. True or False: White-box testing describes the testing of gameplay.
8. True or False: Version control should be applied only to the developers' code.
9. True or False: A "Verify Fix" status on a bug means it will remain on the knockdown list for at least one more test cycle.
10. True or False: A tester should write as many steps as possible when reporting a bug to ensure the bug can be reliably reproduced.
11. On a table next to a bed is a touch-tone landline telephone. Write step-by-step instructions for using that phone to dial the following local number: 555-1234. Assume the person reading the instructions has never seen or used a telephone before.

CHAPTER 15

BASIC TEST PLAN TEMPLATE[1]

Game Name

1. Copyright Information

Table of Contents

SECTION I: QA TEAM (and areas of responsibility)

1. QA Lead
   a. Office phone
   b. Home phone
   c. Mobile phone
   d. Email / IM / VOIP addresses
2. Internal Testers
3. External Test Resources

SECTION II: TESTING PROCEDURES

1. General Approach
   a. Basic Responsibilities of Test Team
      i. Bugs
         1. Detect them as soon as possible after they enter the build
         2. Research them

[1] This chapter appeared in *Game Testing*, *Third Edition*, C. Schultz and R. D. Bryant. Copyright 2017 Mercury Learning and Information. All rights reserved.

3. Communicate them to the dev team
4. Help get them resolved
5. Track them

ii. Maintain the Daily Build

iii. Levels of Communication. There's no point in testing unless the results of the tests are communicated in some fashion. There are a range of possible outputs from QA. In increasing levels of formality, they are:

1. Conversation
2. ICQ/IM/Chat
3. Email to individual
4. Email to group
5. Daily Top Bugs list
6. Stats/Info Dump area on DevSite
7. Formal Entry into Bug Tracking System

2. Daily Activities

a. The Build

i. Generate a daily build.

ii. Run the daily regression tests, as described in "Daily Tests" which follows.

iii. If everything is okay, post the build so everyone can get it.

iv. If there's a problem, send an email message to the entire dev team that the new build cannot be copied, and contact whichever developers can fix the problem.

v. Decide whether a new build needs to be run that day.

b. Daily Tests

i. Run though a predetermined set of single-player levels, performing a specified set of activities.

1. Level #1

a. Activity #1

b. Activity #2

c. Etc.

d. The final activity is usually to run an automated script that reports the results of the various tests and posts them in the QA portion of the internal Web site.

2. Level #2

3. Etc.

ii. Run though a predetermined set of multiplayer levels, performing a specified set of activities.

1. Level #1

a. Activity #1

b. Activity #2

c. Etc.

d. The final activity is usually for each tester involved in the multiplayer game to run an automated script that reports the results of the various tests and posts them in the QA portion of the internal Web site.

2. Level #2

3. Etc.

iii. Email showstopper crashes or critical errors to the entire team.

iv. Post showstopper crashes or critical errors to the daily top bugs list (if one is being maintained).

3. Daily Reports

a. Automated reports from the preceding daily tests are posted in the QA portion of the internal Web site.

4. Weekly Activities

a. Weekly tests

i. Run though every level in the game (not just the preset ones used in the daily test), performing a specified set of activities and generating a predetermined set of tracking statistics. The same machine should be used each week.

1. Level #1

a. Activity #1
b. Activity #2
c. Etc.

2. Level #2
3. Etc.

ii. Weekly review of bugs in the Bug Tracking System

1. Verify that bugs marked "fixed" by the development team really are fixed.
2. Check the appropriateness of bug rankings relative to where the project is in the development.
3. Acquire a "feel" for the current state of the game, which can be communicated in discussions to the producer and department heads.
4. Generate a weekly report of closed-out bugs.

b. Weekly Reports

i. Tracking statistics, as generated in the weekly tests.

5. Ad Hoc Testing

a. Perform specialized tests as requested by the producer, tech lead, or other development team members
b. Determine the appropriate level of communication to report the results of those tests.

6. Integration of Reports from External Test Groups

a. If at all possible, ensure that all test groups are using the same bug tracking system.
b. Determine which group is responsible for maintaining the master list.
c. Determine how frequently to reconcile bug lists against each other.
d. Ensure that only one consolidated set of bugs is reported to the development team.

7. Focus Testing (if applicable)
   a. Recruitment methods
   b. Testing location
   c. Who observes them?
   d. Who communicates with them?
   e. How is their feedback recorded?
8. Compatibility Testing
   a. Selection of external vendor
   b. Evaluation of results
   c. Method of integrating filtered results into bug tracking system

SECTION III: HOW TESTING REQUIREMENTS ARE GENERATED

1. Some requirements are generated by this plan.
2. Requirements can also be generated during project meetings, or other formal meetings held to review current priorities (such as the set of predetermined levels used in the daily tests).
3. Requirements can also result from changes in a bug's status within the bug tracking system. For example, when a bug is marked "fixed" by a developer, a requirement is generated for someone to verify that it has been truly killed and can be closed out. Other status changes include "Need More Info" and "Can't Duplicate," each of which creates a requirement for QA to investigate the bug further.
   a. Some requirements are generated when a developer wants QA to check a certain portion of the game (see "Ad Hoc Testing").

SECTION IV: BUG TRACKING SOFTWARE

1. Package name
2. How many seats will be needed for the project?
3. Access instructions (Everyone on the team should have access to the bug list)
4. "How to report a bug" instructions for using the system

SECTION V: BUG CLASSIFICATIONS

1. "A" bugs and their definition
2. "B" bugs and their definition
3. "C" bugs and their definition

SECTION VI: BUG TRACKING

1. Who classifies the bug?
2. Who assigns the bug?
3. What happens when the bug is fixed?
4. What happens when the fix is verified?

SECTION VII: SCHEDULING AND LOADING

1. Rotation Plan. How testers will be brought on and off the project, so that some testers stay on it throughout its life cycle while "fresh eyes" are periodically brought in.
2. Loading Plan. Resource plan that shows how many testers will be needed at various points in the life of the project.

SECTION VIII: EQUIPMENT BUDGET AND COSTS

1. QA Team Personnel with Hardware and Software Toolset
    a. Team Member #1
        i. Hardware
            1. Testing PC
                a. Specs
            2. Console Debug Kit
                a. Add-ons (TV, controllers, etc.)

3. Record/capture hardware or software (see "Capture/Playback Testing")

ii. Software Tools Needed

1. Bug tracking software
2. Other

b. Team Member #2

c. Etc.

2. Equipment Acquisition Schedule and Costs (summary of who needs what, when they will need it, and how much it will cost)

CHAPTER 16

GAME TESTING BY THE NUMBERS[1]

Inside this Chapter:

16.0. Testing progress
16.1. Testing effectiveness
16.2. Tester performance

Product metrics, such as the number of defects found per line of code, tell you how fit the game code is for release. Test metrics can tell you about the effectiveness and efficiency of your testing activities and results. A few pieces of basic test data can be combined in ways that reveal important information that you can use to keep testing on track, while getting the most out of your tests and testers.

16.0. TESTING PROGRESS

Collecting data is important to understanding where the test team is and where they are headed in terms of meeting the needs and expectations of the overall game project. Data and charts can be collected by the test lead or the individual testers. Take responsibility for knowing how well you're doing. For example, in order to estimate the duration of the test execution for any portion of the game project, estimate the total number of tests to be performed. This number is combined with data on how many tests can be completed per staff-day of effort, how much of a tester's calendar time is actually spent on testing activities, and how many tests you can expect to be redone.

[1] This chapter appeared in *Game Testing*, *Third Edition*, C. Schultz and R. D. Bryant. Copyright 2017 Mercury Learning and Information. All rights reserved.

Figure 16.1 provides a set of data for a test team starting to run tests against a new code release. The project manager worked with the test lead to use an estimate of 12 tests per day as the basis for projecting how long it would take to complete the testing for this release.

Date	Daily Execution		Total Execution	
	Planned	Actual	Planned	Actual
22-Dec	12	13	12	13
23-Dec	12	11	24	24
28-Dec	12	11	36	35
29-Dec	12	12	48	47
30-Dec	12	8	60	55
4-Dec	12	11	72	66
5-Jan	12	10	84	76
6-Jan	12	11	96	87
7-Jan	12	11	108	98
8-Jan	12	16	120	114
10-Jan	12	10	132	124
11-Jan	12	3	144	127
12-Jan	12	7	156	134

FIGURE 16.1 Planned and actual test execution progress data.

Thirteen days into the testing, the progress lagged what had been projected, as shown in Figure 16.2. It looks like progress started to slip on the fifth day, but the team was optimistic that they could catch up. By the tenth day they seemed to have managed to steer back toward the goal, but during the last three days the team lost ground again, despite the reassignment of some people on to and off of the team.

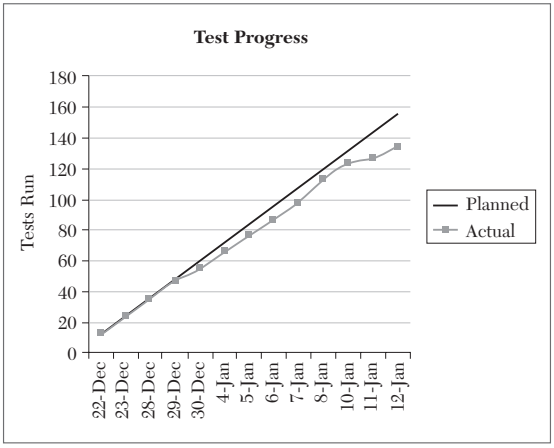


FIGURE 16.2 Planned and actual test execution progress graph.

To understand what is happening here, data was collected for each day that a tester was available to do testing, and the number of tests he or she completed each day. This information can be put into a chart, as shown in Figure 16.3. The totals show that an individual tester completes an average of about four tests a day.

DATE	TESTER					TESTER DAYS	COMPLETED TESTS
	B	C	D	K	Z		
22-Dec	*				*	2	13
23-Dec	*				*	2	11
28-Dec	*				*	2	11
29-Dec	*				*	2	12
30-Dec	*				*	2	8
4-Jan	*		*		*	3	11
5-Jan	*		*		*	3	10
6-Jan	*		*		*	3	11
7-Jan	*		*		*	3	11
8-Jan		*	*	*	*	4	16
10-Jan	*	*	*			3	10
11-Jan	*	*	*			3	3
12-Jan			*			1	7
					TOTALS	33	134
					TESTS/TESTER DAY		4.06

FIGURE 16.3 Test completion rate per tester per day.

Once you have the test effort data for each person and each day, you must compare the test effort people have contributed to the number of work days they were assigned to participate in system testing. Ideally, this ratio would come out to 1.00. The numbers you actually collect will give you a measurement of something you felt was true, but couldn't prove before: most testers are unable to spend 100% of their time on testing. This being the case, don't plan on testers spending 100% of their time on a single task! Measurements will show you how much to expect from system testers, based on various levels of participation. Some testers will be dedicated to testing as their only assignment. Others perhaps perform a dual role, such as developer/tester or QA engineer/tester. Collect effort data for your team members that fall into each category, as shown in Figure 16.4.

FULL - TIME TESTERS

WEEK	1	2	3	4	5	6	7	TOTAL
TESTER DAYS	15.5	21.5	35.5	31.5	36.5	22	23.5	186
ASSIGNED DAYS	44	50	51	53	50	41	41	330
							FULL - TIME TESTER AVAILABILITY	56%

PART - TIME TESTERS

WEEK	1	2	3	4	5	6	7	TOTAL
TESTER DAYS	0	0	0	18.5	18.5	6	15	58
ASSIGNED DAYS	0	0	0	49	54	53	46	202
							PART - TIME TESTER AVAILABILITY	29%

CUMULATIVE

TESTER DAYS	244
ASSIGNED DAYS	532
AVAILABILITY	46%

FIGURE 16.4 Tester participation rate calculations.

These data lead to a number of important points. One is that, given tester "overhead" tasks such as training, meetings, preparing for demos, and so on, a full-time tester might be able to contribute only about 75% of his or her time at best, and 50%–60% on average, over the course of a long project. If you are counting on people with other responsibilities—for example, artists, developers, or QA—to help with testing, then expect only half as much participation as the full-time testers. Using the numbers in Figure 16.4, that would be about 30% of their total available time. You will need to make these measures for your own particular project.

Also, by combining the individual productivity numbers to find a team productivity number, you can see that this team performs only half as many tests as they could if they had 100% of their time to perform testing. This number can be combined with your effort estimate to give an accurate count of calendar work days remaining before testing will be finished. Using the number of 125 tests remaining, and a staff size of 11 testers, you would approximate 11 staff-days of testing remaining. Now that you know what the team's productivity is, however, you divide 11 by 46%, resulting in 24 calendar work days remaining, or nearly five "normal" work weeks. If you had committed to the original, optimistic number of 11 days, there would

be much gnashing of teeth when the tests weren't actually completed until three weeks after they had been promised!

You need this kind of information to answer questions such as "How many people do you need to get testing done by Friday?" or "If I can get you two more testers, when can you be done?"

TIP

Burn into your mind that it's easier to stay on track by getting a little extra done day to day than by trying to make up a large amount in a panic situation; remember Rule #1: Don't Panic.

Going back to Figure 16.1, you can see that on 8-Jan the team was only six tests behind the goal. Completing one extra test on each of the previous six work days would have had the team on goal. If you can keep short-term commitments to stay on track, you will be able to keep the long-term commitment to deliver completed testing on schedule.

16.1. TESTING EFFECTIVENESS

Measure your Test Effectiveness (TE) by adding up defects and dividing by the number of tests completed. This measurement tells you not only how "good" the current release is compared to previous ones, it can also be used to predict how many defects will be found by the remaining tests for that release. For example, with 30 tests remaining and a TE of 0.06, testers should find approximately two more defects. This could be a sign to developers to delay a new code release until the two expected defects are identified, classified, and removed. An example table of TE measurements is shown in Figure 16.5.

Code Release	Defects		Tests Run		Defects/Test	
	New	Total	Release	Total	Release	Total
DEV1	34	34	570	570	0.060	0.060
DEV2	47	81	1230	1800	0.038	0.045
DEV3	39	120	890	2690	0.044	0.045
DEMO1	18	138	490	3180	0.037	0.043
ALPHA1	6	144	220	3400	0.027	0.042

FIGURE 16.5 Test Effectiveness measurements.

You should measure TE for each release as well as for the overall project. Figure 16.6 shows a graphical view of this TE data.

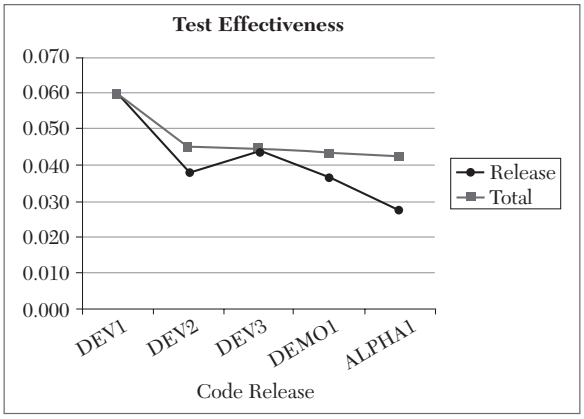


FIGURE 16.6 Test Effectiveness graph.

Notice how the cumulative TE reduced with each release and settled at .042. You can take this measurement one step further by using test completion and defect detection data for each tester in order to calculate individual TEs. Figure 16.7 shows a snapshot of tester TEs for the overall project. You can also calculate each tester's TE per release.

TESTER	B	C	D	K	Z	TOTAL
TESTS RUN	151	71	79	100	169	570
DEFECTS FOUND	9	7	6	3	9	34
DEFECTS /TEST	0.060	0.099	0.076	0.030	0.053	0.060

FIGURE 16.7 TE measured for individual testers.

Note that for this project, the effectiveness of each tester ranges from 0.030 to 0.099, with an average of 0.060. The effectiveness is perhaps as much a function of the particular tests each tester was asked to perform as it is a measure of the skill of each tester. Like the overall TE measurement, however, this number can be used to predict how many additional defects a particular tester could find when performing a known number of tests. For example, if tester C has 40 more tests to perform, expect her to find about four more defects.

In addition to measuring how many defects you detect (quantitative), it is important to understand the severity of defects introduced with each

release (qualitative). Using a defect severity scale of 1 to 4, where 1 is the highest severity, detection of new severity 1 and 2 defects should be reduced to 0 prior to shipping the game. Severity 3 and 4 defect detection should be on a downward trend approaching 0. Figure 16.8 provides examples of severity data.

Release	Defects by Severity				
	1	2	3	4	All
Dev1	7	13	13	1	34
Dev2	4	11	30	2	47
Dev3	2	3	34	0	39
Demo1	1	2	12	3	18
Alpha1	0	0	6	0	6

FIGURE 16.8 Defect severity trend data.

Figure 16.9 graphs the trend of the severity data listed in Figure 16.8. Take a moment to examine the graph. What do you see?

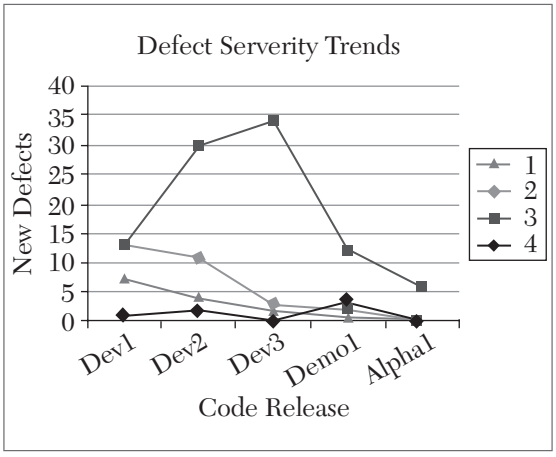


FIGURE 16.9 Defect severity trend graph.

Notice that the severity 3 defects dominate. They are also the only category to significantly increase after Dev1 testing, except for some extra 4s popping up in the Demo1 release. When you set a goal that does not allow any severity 2 defects to be in the shipping game, there will be a tendency to push any borderline severity 2 issues into the severity 3 category. Another explanation could be that the developers focus their efforts on the 1s and 2s so they leave the 3s alone early in the project, with the intention of dealing

with them later. This approach bears itself out in Figures 16.8 and 16.9, where the severity 3 defects are brought way down for the Demo1 release and continue to drop in the Alpha1 release. Once you see "what" is happening, try to understand "why" it is happening that way.

16.2. TESTER PERFORMANCE

You can implement some other measurements to encourage testers to find defects and to give them a sense of pride in their skills. One of them is the "Star Chart." This chart is posted in the testing area and shows the accomplishments of each tester according to how many defects they find of each severity. Tester names are listed down one side of the chart and each defect is indicated by a stick-on star. The star's color indicates the defect's severity. For example, you can use blue for 1, red for 2, yellow for 3, and silver for 4. Points can also be assigned to each severity (for example, A = 10, B = 5, C = 3, D = 1), and a "Testing Star" can be declared at the end of the project based on who has the most points.

NOTE *In our experience, this chart has led to a sense of friendly competition among testers, increased their determination to find defects, promoted tester ownership of defects, and has caused testers to pay more attention to the severity assigned to the defects they find. This approach turns testing into a game for the testers to play while they're testing games.*

Figure 16.10 shows what a Star Chart looks like prior to applying the testers' stars.

STAR CHART FOR XYZZY

TESTERS	STARS (Sev. 1 = BLUE, 2 = RED, 3 = YELLOW, 4 = SILVER)
B	
C	
D	
K	
Z	

FIGURE 16.10 Empty Star Chart.

If you're worried about testers getting into battles over defects and not finishing their assigned tests quickly enough, you can create a composite measure of each tester's contribution to test execution and defects found. Add the total number of test defects found and calculate a percentage for each tester, based on how many they found divided by the project total. Then do the same for tests run. You can add these two numbers for each tester. Whoever has the highest total is the "Best Tester" for the project. This might or might not turn out to be the same person who becomes the Testing Star. Here's how this works for testers B, C, D, K, and Z for the Dev1 release:

- Tester B executed 151 of the team's 570 Dev1 tests. This comes out to 26.5%. B has also found 9 of the 34 Dev1 defects, which is also 26.5%. B's composite rating is 53.
- Tester C ran 71 of the 570 tests, which is 12.5%. C found 7 out of the 34 total defects in Dev1, which is 20.5%. C's rating is 33.
- Tester D ran 79 tests, which is approximately 14% of the total. D also found 6 defects, which is about 17.5% of the total. D earns a rating of 31.5.
- Tester K ran 100 tests and found 3 defects. These represent 17.5% of the test total and about 9% of the defect total. K has a 26.5 rating.
- Tester Z ran 169 tests, which is about 29.5% of the 570 total. Z found 9 defects, which is 26.5% of that total. Z's total rating is 56.
- Tester Z has earned the title of "Best Tester."

TIP

When you have people on your team who keep winning these awards, take them to lunch and find out what they are doing so you can win some too!

Be careful to use this system for good and not for evil. Running more tests or claiming credit for new defects should not come at the expense of other people or the good of the overall project. You could add factors to give more weight to higher-severity defects in order to discourage testers from spending all their time chasing and reporting low-severity defects that won't contribute as much to the game as a few very important high-severity defects.

Use this system to encourage and exhibit positive test behaviors. Remind your team (and yourself!) that some time spent automating tests could have generous payback in terms of test execution. Likewise, spending a little time

up front to effectively design your tests, before you run off to start banging on the game controller, will probably lead you to more defects. You will learn more about these strategies and techniques in the remaining chapters of this book.

This chapter introduced you to a number of metrics you can collect to track and improve testing results. Each metric from this chapter is listed below, together with the raw data you need to collect for each, mentioned in parentheses:

- Test Progress Chart (# of tests completed by team each day, # of tests required each day)
- Test Completed/Days of Effort (# of tests completed,# days of test effort for each tester)
- Test Participation (# of days of effort for each tester, # of days each tester assigned to test)
- Test Effectiveness (# of defects, # of tests for each release and/or tester)
- Defect Severity Profile (# of defects of each severity for each release)
- Star Chart (# of defects of each severity for each tester)
- Testing Star (# of defects of each severity for each tester, point value of each severity)
- Best Tester (# of tests per tester, # of total tests, # of defects per tester, # of total defects)

Testers or test leads can use these metrics to aid in planning, predicting, and performing game testing activities. Then you will be *testing by the numbers.*

EXERCISES

1. How does the data in Figure 16.3 explain what is happening on the graph in Figure 16.2?
2. How many testers do you need to add to the project represented in Figures 16.1 and 16.2 in order to bring the test execution back on plan in the next 10 working days? The testers will begin work on the very next day that is plotted on the graph.

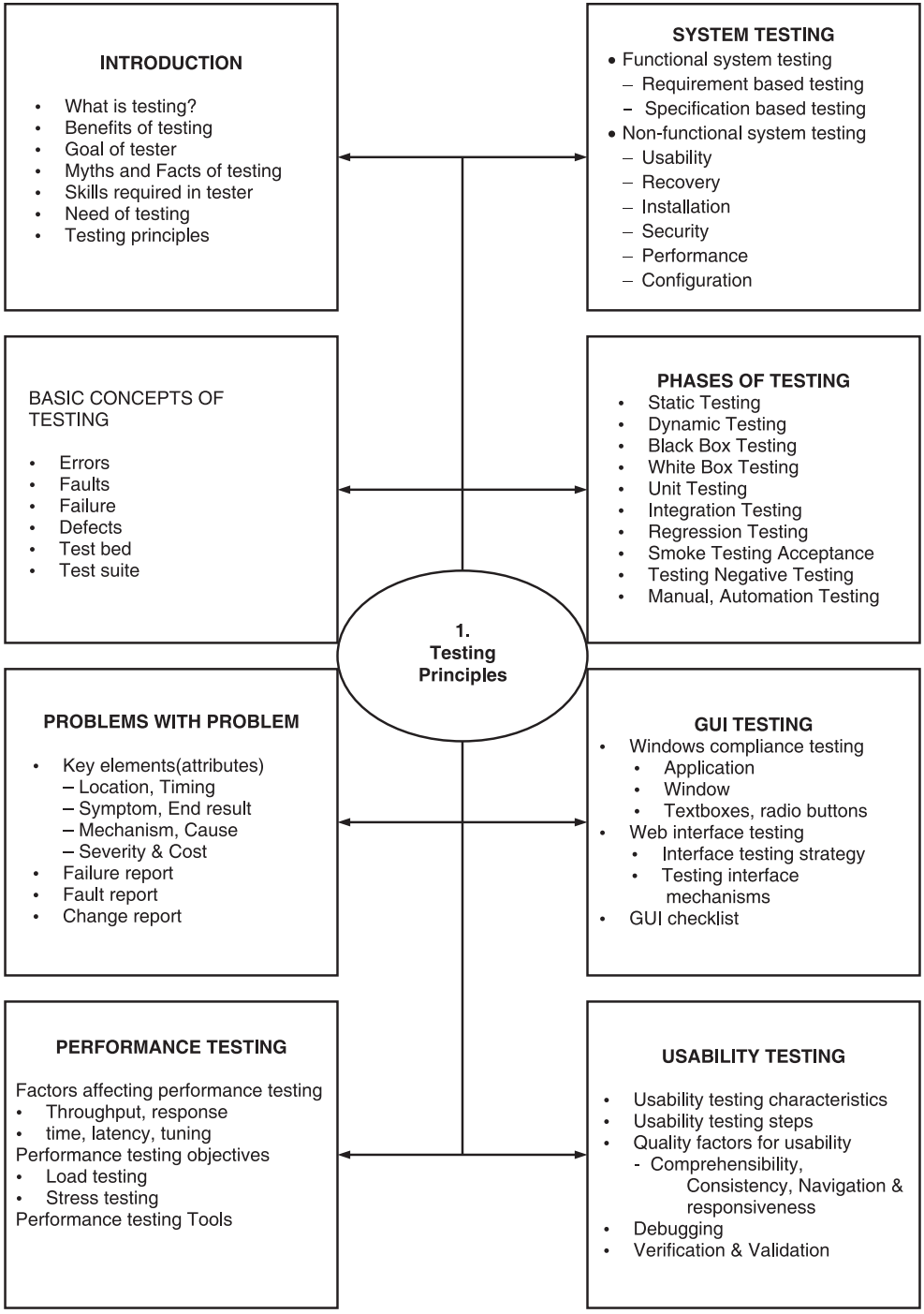
3. Tester C has the best TE as shown in Figure 16.7, but did not turn out to be the "Best Tester." Explain how this happened.
4. You are tester X working on the project represented in Figure 16.7. If you have run 130 tests, how many defects did you need to find in order to become the "Best Tester?"
5. Describe three positive and three negative aspects of measuring the participation and effectiveness of individual testers. Do not include any aspects already discussed in this chapter.

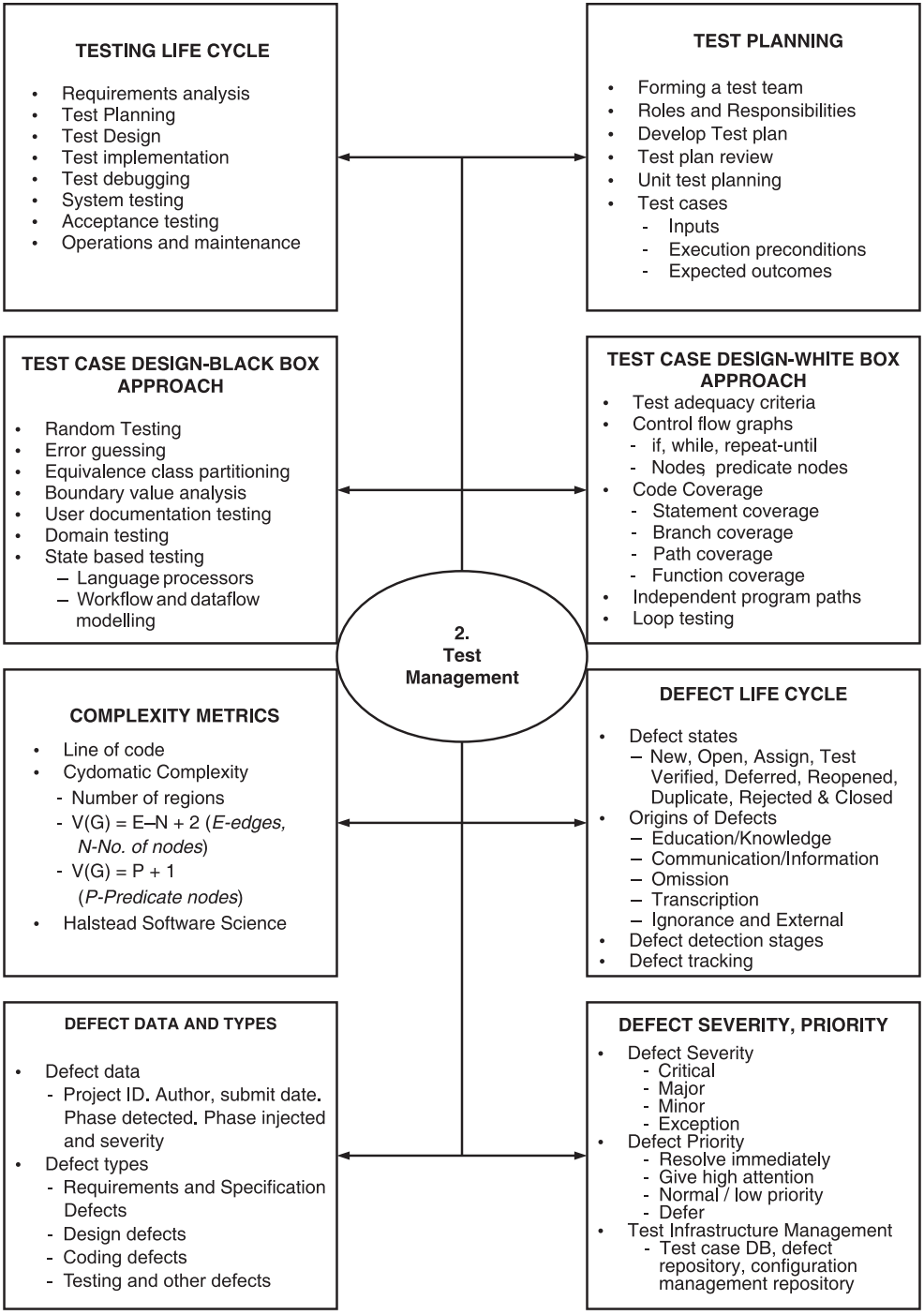
APPENDIX A

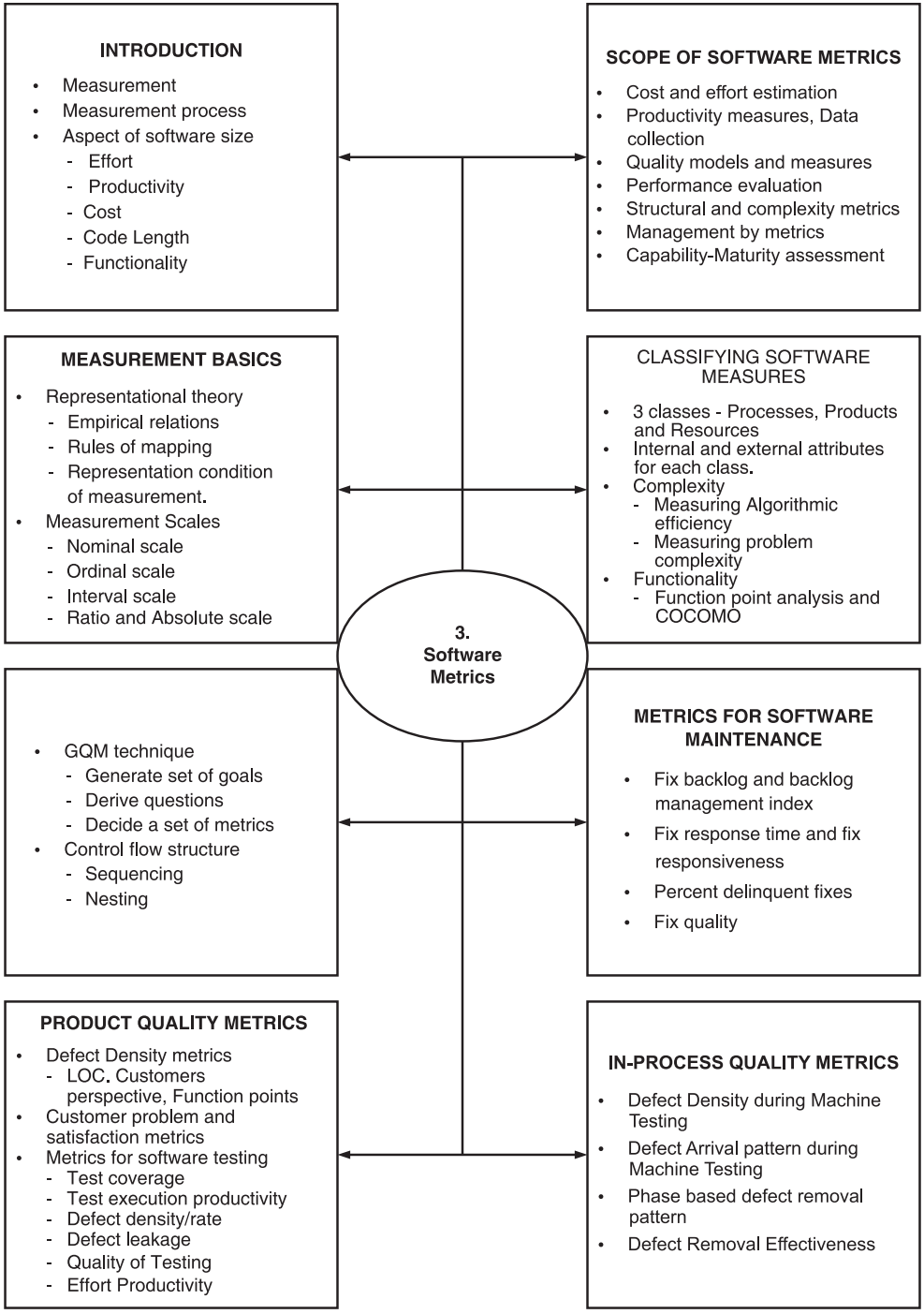
QUALITY ASSURANCE AND TESTING TOOLS

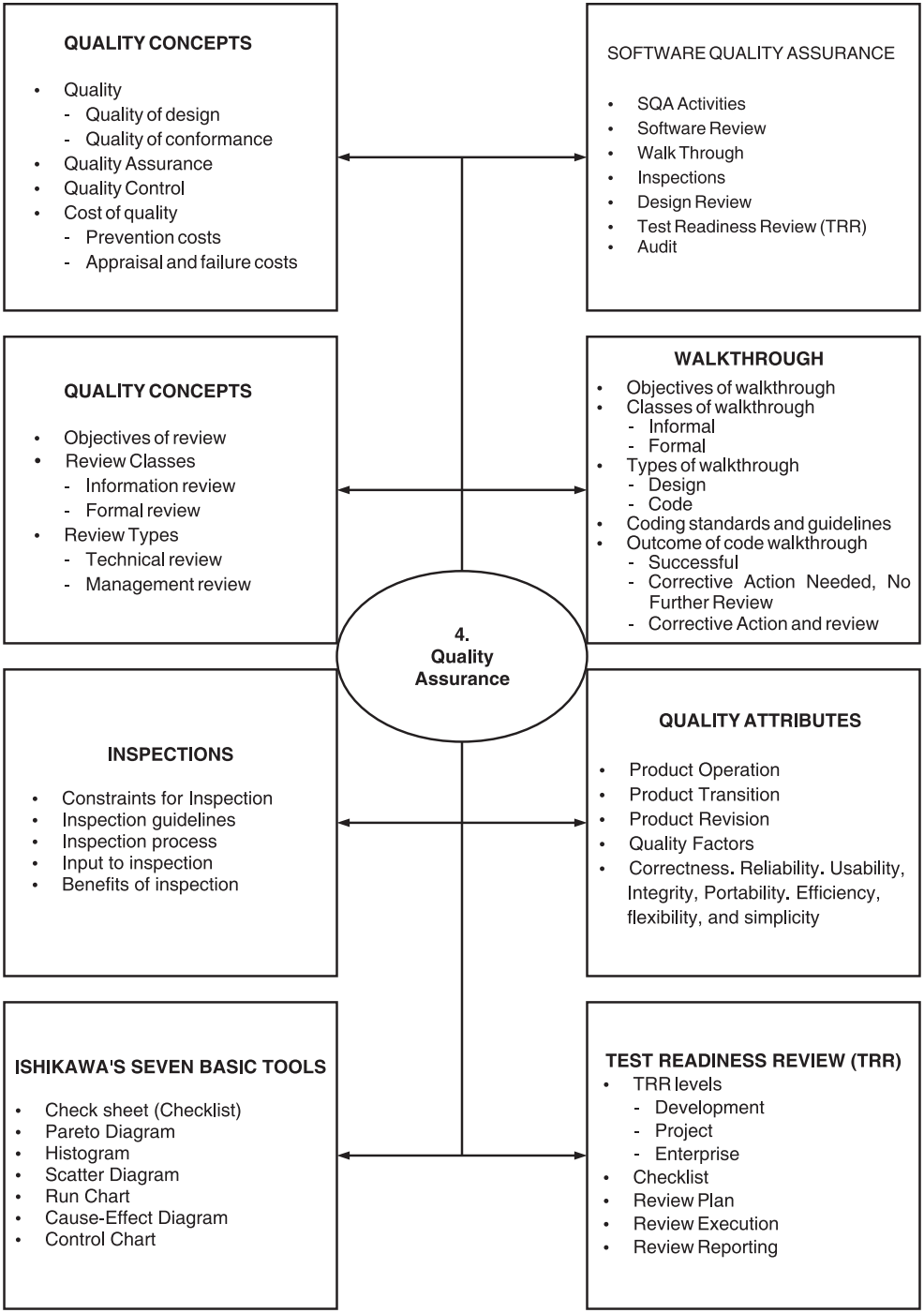
IEEE/ANSI Standard	Software Test Process	Purpose
829–1983	Software Test Documentation	This standard covers the entire testing process.
1008–1987	Software Unit Testing	This standard defines an integrated approach to systematic and documented unit testing.
1012–1986	Software Verification and Validation Plans	This standard provides uniform and minimum requirements for the format and content of software verification and validation plans.
1028–1988	Software Reviews and Audits	This standard provides direction to the reviewer or auditor on the conduct of evaluations.
730–1989	Software Quality Assurance Plans	This standard establishes a required format and a set of minimum contents for software quality assurance plans.
828–1990	Software Configuration Management Plans	This standard is similar to IEEE standard 730, but deals with the more limited subject of software configuration management. This standard identifies requirements for configuration identification, configuration control, configuration status reporting, configuration audits and reviews.
1061–1992	Software Quality Metrics Methodology	This standard provides a methodology for establishing quality requirements. It also deals with identifying, implementing, analyzing and validating the process of software quality metrics.

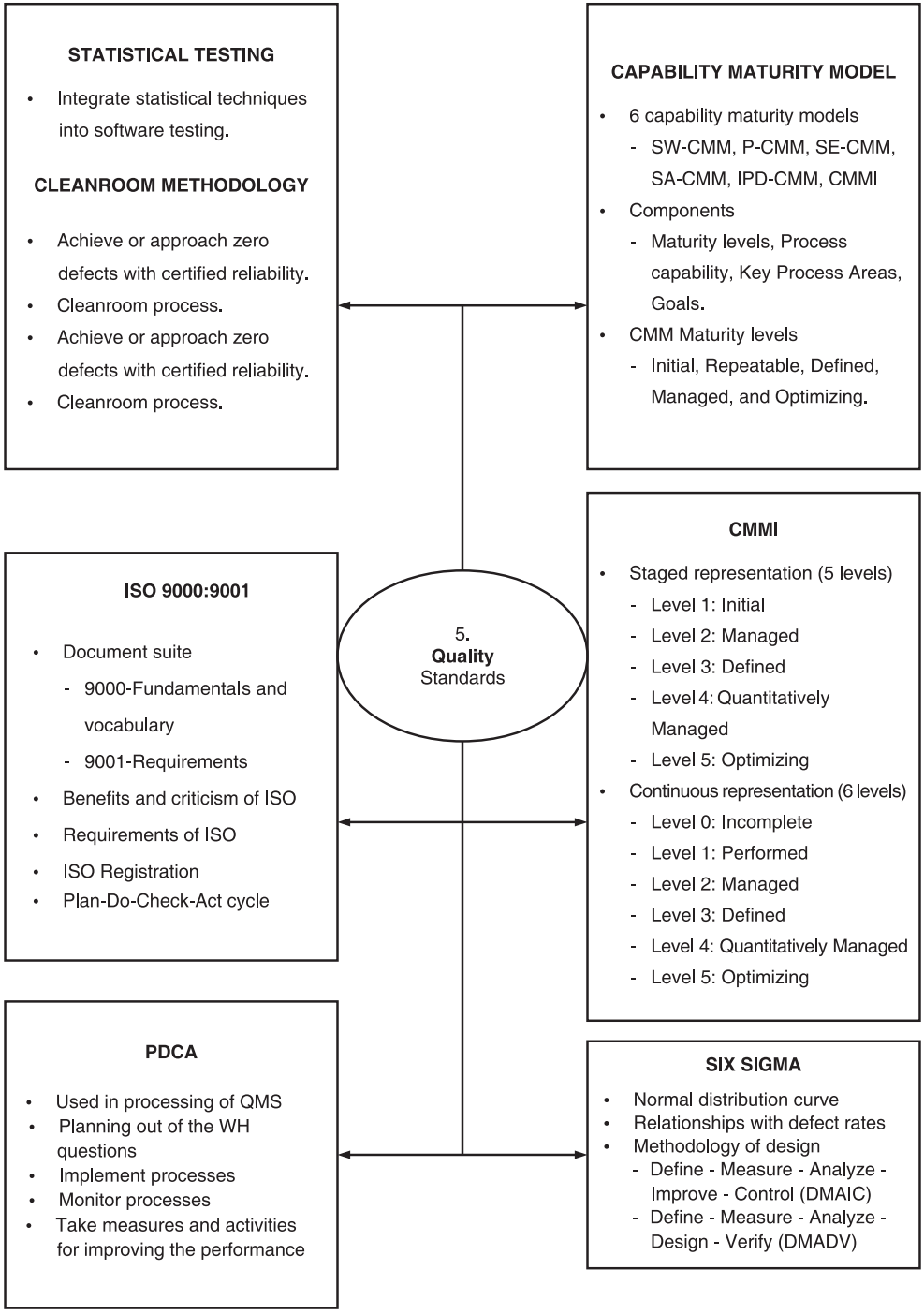
Description	Tools
Functional/Regression Testing	WinRunner Silkiest Quick Test Pro (QTP) Rational Robot Visual Test In-house Scripts
Load/Stress Testing (Performance)	LoadRunner Astra Load Test Application Centre Test (ATC) In-house Scripts Web Application Stress Tool (WAS)
Test Case Management	Test Director Test Manager In-house Test Case Management tools
Defect Tracking	TestTrack Pro Bugzilla Element Tool ClearQuest TrackRecord In-house Defect Tracking tools of clients
Unit/Integration Testing	C++ Test JUnit NUnit PhpUnit Check Cantata++

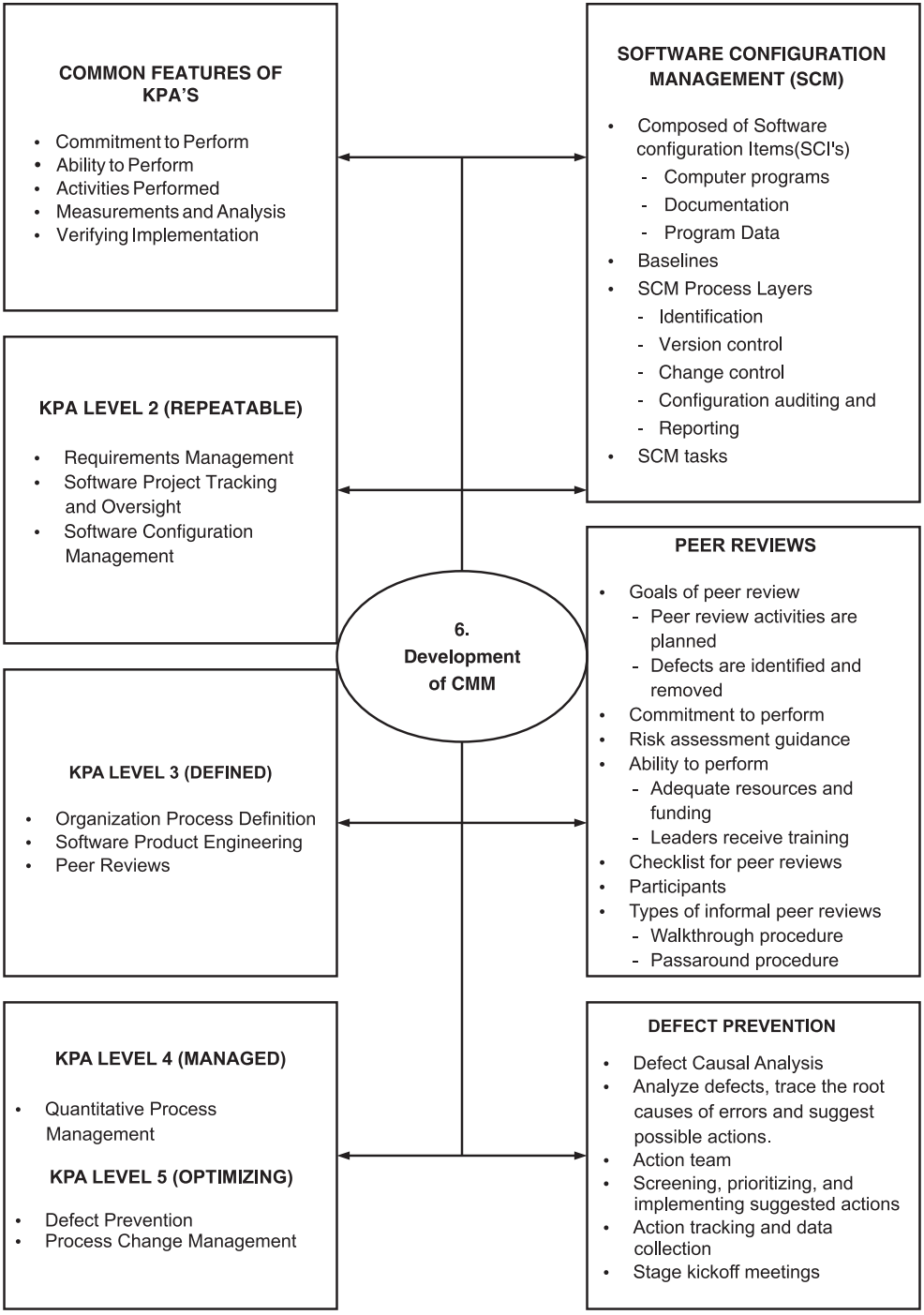

INTRODUCTION
What is testing?
Benefits of testing
Goal of tester
Myths and Facts of testing
Skills required in tester
Need of testing
Testing principles
SYSTEM TESTING
Functional system testing
Requirement based testing
Specification based testing
Non-functional system testing
Usability
Recovery
Installation
Security
Performance
Configuration
BASIC CONCEPTS OF TESTING
Errors
Faults
Failure
Defects
Test bed
Test suite
PHASES OF TESTING
Static Testing
Dynamic Testing
Black Box Testing
White Box Testing
Unit Testing
Integration Testing
Regression Testing
Smoke Testing Acceptance
Testing Negative Testing
Manual, Automation Testing
1. Testing Principles
PROBLEMS WITH PROBLEM
Key elements(attributes)
Location, Timing
Symptom, End result
Mechanism, Cause
Severity & Cost
Failure report
Fault report
Change report
GUI TESTING
Windows compliance testing
Application
Window
Textboxes, radio buttons
Web interface testing
Interface testing strategy
Testing interface mechanisms
GUI checklist
PERFORMANCE TESTING
Factors affecting performance testing
Throughput, response
time, latency, tuning
Performance testing objectives
Load testing
Stress testing
Performance testing Tools
USABILITY TESTING
Usability testing characteristics
Usability testing steps
Quality factors for usability
Comprehensibility, Consistency, Navigation & responsiveness
Debugging
Verification & Validation


TESTING LIFE CYCLE
• Requirements analysis
• Test Planning
• Test Design
• Test implementation
• Test debugging
• System testing
• Acceptance testing
• Operations and maintenance
TEST PLANNING
• Forming a test team
• Roles and Responsibilities
• Develop Test plan
• Test plan review
• Unit test planning
• Test cases
- Inputs
- Execution preconditions
- Expected outcomes
TEST CASE DESIGN-BLACK BOX APPROACH
• Random Testing
• Error guessing
• Equivalence class partitioning
• Boundary value analysis
• User documentation testing
• Domain testing
• State based testing
– Language processors
– Workflow and dataflow modelling
TEST CASE DESIGN-WHITE BOX APPROACH
• Test adequacy criteria
• Control flow graphs
- if, while, repeat-until
- Nodes predicate nodes
• Code Coverage
- Statement coverage
- Branch coverage
- Path coverage
- Function coverage
• Independent program paths
• Loop testing
2. Test Management
COMPLEXITY METRICS
• Line of code
• Cydomatic Complexity
- Number of regions
- V(G) = E–N + 2 (E-edges, N-No. of nodes)
- V(G) = P + 1 (P-Predicate nodes)
• Halstead Software Science
DEFECT LIFE CYCLE
• Defect states
– New, Open, Assign, Test Verified, Deferred, Reopened, Duplicate, Rejected & Closed
• Origins of Defects
– Education/Knowledge
– Communication/Information
– Omission
– Transcription
– Ignorance and External
• Defect detection stages
• Defect tracking
DEFECT DATA AND TYPES
• Defect data
- Project ID. Author, submit date. Phase detected. Phase injected and severity
• Defect types
- Requirements and Specification Defects
- Design defects
- Coding defects
- Testing and other defects
DEFECT SEVERITY, PRIORITY
• Defect Severity
- Critical
- Major
- Minor
- Exception
• Defect Priority
- Resolve immediately
- Give high attention
- Normal / low priority
- Defer
• Test Infrastructure Management
- Test case DB, defect repository, configuration management repository


INTRODUCTION
• Measurement
• Measurement process
• Aspect of software size
- Effort
- Productivity
- Cost
- Code Length
- Functionality
SCOPE OF SOFTWARE METRICS
• Cost and effort estimation
• Productivity measures, Data collection
• Quality models and measures
• Performance evaluation
• Structural and complexity metrics
• Management by metrics
• Capability-Maturity assessment
MEASUREMENT BASICS
• Representational theory
- Empirical relations
- Rules of mapping
- Representation condition of measurement.
• Measurement Scales
- Nominal scale
- Ordinal scale
- Interval scale
- Ratio and Absolute scale
CLASSIFYING SOFTWARE MEASURES
• 3 classes - Processes, Products and Resources
• Internal and external attributes for each class.
• Complexity
- Measuring Algorithmic efficiency
- Measuring problem complexity
• Functionality
- Function point analysis and COCOMO
3. Software Metrics
• GQM technique
- Generate set of goals
- Derive questions
- Decide a set of metrics
• Control flow structure
- Sequencing
- Nesting
METRICS FOR SOFTWARE MAINTENANCE
• Fix backlog and backlog management index
• Fix response time and fix responsiveness
• Percent delinquent fixes
• Fix quality
PRODUCT QUALITY METRICS
• Defect Density metrics
- LOC. Customers perspective, Function points
• Customer problem and satisfaction metrics
• Metrics for software testing
- Test coverage
- Test execution productivity
- Defect density/rate
- Defect leakage
- Quality of Testing
- Effort Productivity
IN-PROCESS QUALITY METRICS
• Defect Density during Machine Testing
• Defect Arrival pattern during Machine Testing
• Phase based defect removal pattern
• Defect Removal Effectiveness


QUALITY CONCEPTS
• Quality
- Quality of design
- Quality of conformance
• Quality Assurance
• Quality Control
• Cost of quality
- Prevention costs
- Appraisal and failure costs
SOFTWARE QUALITY ASSURANCE
• SQA Activities
• Software Review
• Walk Through
• Inspections
• Design Review
• Test Readiness Review (TRR)
• Audit
QUALITY CONCEPTS
• Objectives of review
• Review Classes
- Information review
- Formal review
• Review Types
- Technical review
- Management review
WALKTHROUGH
• Objectives of walkthrough
• Classes of walkthrough
- Informal
- Formal
• Types of walkthrough
- Design
- Code
• Coding standards and guidelines
• Outcome of code walkthrough
- Successful
- Corrective Action Needed, No Further Review
- Corrective Action and review
4. Quality Assurance
INSPECTIONS
• Constraints for Inspection
• Inspection guidelines
• Inspection process
• Input to inspection
• Benefits of inspection
QUALITY ATTRIBUTES
• Product Operation
• Product Transition
• Product Revision
• Quality Factors
• Correctness. Reliability. Usability, Integrity, Portability. Efficiency, flexibility, and simplicity
ISHIKAWA'S SEVEN BASIC TOOLS
• Check sheet (Checklist)
• Pareto Diagram
• Histogram
• Scatter Diagram
• Run Chart
• Cause-Effect Diagram
• Control Chart
TEST READINESS REVIEW (TRR)
• TRR levels
- Development
- Project
- Enterprise
• Checklist
• Review Plan
• Review Execution
• Review Reporting


STATISTICAL TESTING
• Integrate statistical techniques into software testing.
CLEANROOM METHODOLOGY
• Achieve or approach zero defects with certified reliability.
• Cleanroom process.
• Achieve or approach zero defects with certified reliability.
• Cleanroom process.
CAPABILITY MATURITY MODEL
• 6 capability maturity models
- SW-CMM, P-CMM, SE-CMM, SA-CMM, IPD-CMM, CMMI
• Components
- Maturity levels, Process capability, Key Process Areas, Goals.
• CMM Maturity levels
- Initial, Repeatable, Defined, Managed, and Optimizing.
ISO 9000:9001
• Document suite
- 9000-Fundamentals and vocabulary
- 9001-Requirements
• Benefits and criticism of ISO
• Requirements of ISO
• ISO Registration
• Plan-Do-Check-Act cycle
5.
Quality
Standards
CMMI
• Staged representation (5 levels)
- Level 1: Initial
- Level 2: Managed
- Level 3: Defined
- Level 4: Quantitatively Managed
- Level 5: Optimizing
• Continuous representation (6 levels)
- Level 0: Incomplete
- Level 1: Performed
- Level 2: Managed
- Level 3: Defined
- Level 4: Quantitatively Managed
- Level 5: Optimizing
PDCA
• Used in processing of QMS
• Planning out of the WH questions
• Implement processes
• Monitor processes
• Take measures and activities for improving the performance
SIX SIGMA
• Normal distribution curve
• Relationships with defect rates
• Methodology of design
- Define - Measure - Analyze - Improve - Control (DMAIC)
- Define - Measure - Analyze - Design - Verify (DMADV)


COMMON FEATURES OF KPA'S
• Commitment to Perform
• Ability to Perform
• Activities Performed
• Measurements and Analysis
• Verifying Implementation
KPA LEVEL 2 (REPEATABLE)
• Requirements Management
• Software Project Tracking and Oversight
• Software Configuration Management
KPA LEVEL 3 (DEFINED)
• Organization Process Definition
• Software Product Engineering
• Peer Reviews
KPA LEVEL 4 (MANAGED)
• Quantitative Process Management
KPA LEVEL 5 (OPTIMIZING)
• Defect Prevention
• Process Change Management
6. Development of CMM
SOFTWARE CONFIGURATION MANAGEMENT (SCM)
• Composed of Software configuration Items(SCI's)
- Computer programs
- Documentation
- Program Data
• Baselines
• SCM Process Layers
- Identification
- Version control
- Change control
- Configuration auditing and
- Reporting
• SCM tasks
PEER REVIEWS
• Goals of peer review
- Peer review activities are planned
- Defects are identified and removed
• Commitment to perform
• Risk assessment guidance
• Ability to perform
- Adequate resources and funding
- Leaders receive training
• Checklist for peer reviews
• Participants
• Types of informal peer reviews
- Walkthrough procedure
- Passaround procedure
DEFECT PREVENTION
• Defect Causal Analysis
• Analyze defects, trace the root causes of errors and suggest possible actions.
• Action team
• Screening, prioritizing, and implementing suggested actions
• Action tracking and data collection
• Stage kickoff meetings

APPENDIX B

SAMPLE PROJECT DESCRIPTION

[N.B.: Students may be encouraged to prepare descriptions of projects on these lines and then develop the following deliverables]

1. SRS Document 2. Design Document 3. Codes 4. Test Oracles

Title of the Project

Development of a practical Online Help Desk (OHD) for the facilities in the campus.

Abstract of the Project

This project is aimed at developing an Online Help Desk (OHD) for the facilities in the campus. This is an Intranet-based application that can be accessed throughout the campus. This system can be used to automate the workflow of service requests for the various facilities in the campus. This is one integrated system that covers different kinds of facilities like classrooms, labs, hostels, mess, canteen, gymnasium, computer center, faculty club, etc. Registered users (students, faculty, lab-assistants, and others) will be able to log in a request for service for any of the supported facilities. These requests will be sent to the concerned people, who are also valid users of the system, to get them resolved. There are features like email notifications/reminders, the addition of a new facility to the system, report generators, etc. in this system.

Keywords

Generic Technology Keywords: databases, network and middleware, programming.

Specific Technology Keywords: MS-SQL server, HTML, Active Server Pages.

Project Type Keywords: analysis, design, implementation, testing, user interface.

Functional Components of the Project

The following is a list of functionalities of the system. Other functionalities that you find appropriate can be added to this list. Other facilities that are appropriate to your college can be included in the system. And, in places where the description of a functionality is not adequate, you can make appropriate assumptions and proceed.

There are registered people in the system (students, faculty, lab-assistants, and others). Some of them are responsible for maintaining the facilities (for example, the lab-assistant is responsible for keeping the lab ready with all the equipment in proper condition, the student council is responsible for taking forward students' complaints/requests to the faculty/administration. etc.).

There are three kinds of users for this system:

1. Those who use the system to create a request (end-users).
2. Those who look at the created requests and assign them to the concerned people (facility-heads).
3. Those who work on the assigned requests and update the status on the system (assignees).

There is also an "Administrator" for doing the Admin-level functions such as creating user accounts, adding new facilities to the system, etc.

1. A person should be able to —
   - login to the system through the first page of the application.
   - change the password after logging into the system.
   - see the status of the requests created by him/her (the status could be one of unassigned/assigned/work in progress/closed/rejected).
   - see the list of requests (both open and closed) created by him/her in the past.
   - create a new request by specifying the facility, the severity of the request (there may be several levels of severity defined), and a brief description of the request.
   - close a request created by him/her by giving an appropriate reason.
   - see the requests that are assigned to him/her by the facility-heads and update the status of requests (after working on them).
   - view the incoming requests (if he/she is a facility-head) and assign them to registered users of the system.
   - get help about the OHD system on how to use the different features of the system.

2. As soon as a request is created, an automatic email should be sent to the person who created the request and the concerned facility-head. The email should contain the request details.
3. Similarly, when any status-change occurs for a request (such as the request getting completed etc.), an automatic email should be sent to the person who created the request and the concerned facility-head.
4. A summary report on the requests that came in and requests that were serviced should be sent to every facility-head periodically (say, once in a month).

Steps to Start-off the Project

The following steps will be helpful to start-off the project.

1. Study and be comfortable with technologies such as Active Server Pages/ HTML and SQL server. Some links to these technologies are given in the "Guidelines and References" section of this document.
2. Decide on the list of facilities that would be supported and define it formally.
3. Make a database of different kinds of users (End-users, Facility-heads, Assignees).
4. Assign a system-admin who will create mail-ids for the people in the Intranet of your lab or in the Internet. These mail-ids will be used for sending automatic notifications and reports. The system-admin will also take care of assigning the logins to the users of the OHD system.
5. Create the front-page of the OHD system giving a brief description about the system and a login box.
6. Create the help-pages of the system in the form of a Q&A. This will help you also when implementing the system.
7. Create other subsystems like automatic notification, screens for various functions (like create_new_request, view_open_requests, forward_ new_request_to_assignee, etc.).

Requirements:

Hardware requirements:

Number	Description	Alternatives (if available)
1.	PC with 2 GB hard-disk and 256 MB RAM	Not-Applicable
2.		

Software requirements:

Number	Description	Alternatives (if available)
1.	Windows 95/98/XP with MS-Office	Not-Applicable
2.	MS-SQL server	MS-Access
3.		

Manpower requirements:

2-3 students can complete this in 4-6 months if they work full-time on it.

Milestones and timelines:

Number	Milestone name	Milestone description	Timeline week no. from start of project	Remarks
1.	Requirements specification	Complete the specification of the system (with appropriate assumptions) including the facilities that would be supported, the services in each facility that would be supported, selection of facility-heads, assignees, and administrator constitutes this milestone. A document detailing the same should be written and a presentation on that be made.	1-2	Attempt should be made to add some more relevant functionalities other than those that are listed in this document. Attempt should be made to clearly formulate the work-flow of each of the services (for example, who will take care of replacing a faulty bulb in the lab, who will take care of ordering a new book/ magazine for the college library, etc.).

(*Continued*)

Number	Milestone name	Milestone description	Timeline week no. from start of project	Remarks
2.	Technology familiarization	Understanding of the technology needed to implement the project.	3-4	The presentation should be from the point of view of being able to apply it to the project, rather than from a theoretical perspective.
3.	High-level and detailed design	Listing all possible scenarios (like request creation, request assignment, status updates on a request etc.) and then coming up with flow-charts or pseudo-code to handle the scenario.	5-7	The scenarios should map to the requirement specification (i.e., for each requirement that is specified, a corresponding scenario should be there).
4.	Implementation of the front-end of the system	Implementation of the main screen giving the login, screen that follows the login giving various options, screens for facility-head, screens for the administrator functions, etc.	7-9	During this milestone period, it would be a good idea for the team (or one person from the team) to start working on a test-plan for the entire system. This test-plan can be updated as and when new scenarios come to mind.
5.	Integrating the front-end with the database	The front-end, developed in the earlier milestone will now be able to update the facilities database. Other features like mail notification, etc. should be functional at this stage. In short, the system should be ready for integration testing.	10-12	

(*Continued*)

Number	Milestone name	Milestone description	Timeline week no. from start of project	Remarks
6.	Integration testing	The system should be thoroughly tested by running all the test cases written for the system (from milestone 5).	13-14	Another 2 weeks should be sufficient to handle any issues found during the testing of the system. After that, the final demo can be arranged.
7.	Final review	Issues found during the previous milestone are fixed and the system is ready for the final review.	15-16	During the final review of the project, it should be checked that all the requirements specified during milestone number 1 are fulfilled (or appropriate reasons given for not fulfilling them).

Guidelines and References

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnasp/html/asptutorial.asp (ASP tutorial)

http://www.functionx.com/sqlserver/ (SQL-server tutorial)

APPENDIX C

GLOSSARY

Abstract class: A class that cannot be instantiated, i.e., it cannot have any instances.

Abstract test case: See high-level test case.

Acceptance: See acceptance testing.

Acceptance criteria: The exit criteria that a component or system must satisfy in order to be accepted by a user, customer, or other authorized entity. [IEEE 6.10]

Acceptance testing: It is done by the customer to check whether the product is ready for use in the real-life environment. Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers, or other authorized entity to determine whether or not to accept the system. [After IEEE 610]

Accessibility testing: Testing to determine the ease by which users with disabilities can use a component or system. [Gerrard]

Accuracy: The capability of the software product to provide the right or agreed results or effects with the needed degree of precision. [ISO 9126] See also functionality testing.

Activity: A major unit of work to be completed in achieving the objectives of a hardware/ software project.

Actor: An actor is a role played by a person, organization, or any other device which interacts with the system.

Actual outcome: See actual result.

Actual result: The behavior produced/observed when a component or system is tested.

Ad hoc review: See informal review.

Ad hoc testing: Testing carried out informally; no formal test preparation takes place, no recognized test design technique is used, there are no expectations for results and randomness guides the test execution activity.

Adaptability: The capability of the software product to be adapted for different specified environments without applying actions or means other than those provided for this purpose for the software considered. [ISO 9126] See also portability testing.

Agile testing: Testing practice for a project using agile methodologies, such as extreme programming (XP), treating development as the customer of testing and emphasizing the test-first design paradigm.

Aggregation: Process of building up of complex objects out of existing objects.

Algorithm test [TMap]: See branch testing.

Alpha testing: Simulated or actual operational testing by potential users/customers or an independent test team at the developers' site, but outside the development organization. Alpha testing is often employed as a form of internal acceptance testing.

Analyst: An individual who is trained and experienced in analyzing existing systems to prepare SRS (software requirement specifications).

Analyzability: The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified. [ISO 9126] See also maintainability testing.

Analyzer: See static analyzer.

Anomaly: Any condition that deviates from expectation based on requirements specifications, design documents, user documents, standards, etc. or from someone's perception or experience. Anomalies may be found during, but not limited to, reviewing, testing, analysis, compilation, or use of software products or applicable documentation. [IEEE 1044] See also defect, deviation, error, fault, failure, incident, or problem.

Arc testing: See branch testing.

Atomicity: A property of a transaction that ensures it is completed entirely or not at all.

Attractiveness: The capability of the software product to be attractive to the user. [ISO 9126] See also usability testing.

Audit: An independent evaluation of software products or processes to ascertain compliance to standards, guidelines, specifications, and/or procedures based on objective criteria, including documents that specify: (1) the form or content of the products to be produced, (2) the process by which the products shall be produced, and (3) how compliance to standards or guidelines shall be measured. [IEEE 1028]

Audit trail: A path by which the original input to a process (e.g., data) can be traced back through the process, taking the process output as a starting point. This facilitates defect analysis and allows a process audit to be carried out. [After TMap]

Automated testware: Testware used in automated testing, such as tool scripts.

Availability: The degree to which a component or system is operational and accessible when required for use. Often expressed as a percentage. [IEEE 610]

Back-to-back testing: Testing in which two or more variants of a component or system are executed with the same inputs, the outputs compared, and analyzed in cases of discrepancies. [IEEE 610]

Baseline: A specification or software product that has been formally reviewed or agreed upon, that thereafter serves as the basis for further development, and that can be changed only through a formal change control process. [After IEEE 610] A hardware/software work product that has been formally reviewed and agreed upon, which serves as the basis for further development.

Basic block: A sequence of one or more consecutive executable statements containing no branches.

Basis test set: A set of test cases derived from the internal structure or specification to ensure that 100% of a specified coverage criterion is achieved.

Bebugging: See error seeding. [Abbott]

Behavior: The response of a component or system to a set of input values and preconditions.

Benchmark test: (1) A standard against which measurements or comparisons can be made. (2) A test that is to be used to compare components or systems to each other or to a standard as in (1). [After IEEE 610]

Bespoke software: Software developed specifically for a set of users or customers. The opposite is off-the-shelf software.

Best practice: A superior method or innovative practice that contributes to the improved performance of an organization under given context, usually recognized as "best" by other peer organizations.

Beta testing: Operational testing by potential and/or existing users/customers at an external site not otherwise involved with the developers, to determine whether or not a component or system satisfies the user/customer needs and fits within the business processes. Beta testing is often employed as a form of external acceptance testing in order to acquire feedback from the market.

Big-bang testing: Testing all modules at once. A type of integration testing in which software elements, hardware elements, or both are combined all at once into a component or an overall system, rather than in stages. [After IEEE 610] See also integration testing.

Black-box technique: See black-box test design technique.

Black-box test design technique: Documented procedure to derive and select test cases based on an analysis of the specification, either functional or non-functional, of a component or system without reference to its internal structure.

Black-box testing: Testing, either functional or non-functional, without reference to the internal structure of the component or system.

Blocked test case: A test case that cannot be executed because the preconditions for its execution are not fulfilled.

Bohr bug: A repeatable bug; one that manifests reliably under a possibly unknown but well defined set of conditions.

Bottom-up testing: An incremental approach to integration testing where the lowest level components are tested first, and then used to facilitate the testing of higher level components. This process is repeated until the component at the top of the hierarchy is tested. See also integration testing.

Boundary value: An input value or output value which is on the edge of an equivalence partition or at the smallest incremental distance on either side of an edge, for example, the minimum or maximum value of a range.

Boundary value analysis: A black-box test design technique in which test cases are designed based on boundary values.

Boundary value coverage: The percentage of boundary values that have been exercised by a test suite.

Boundary value testing: See boundary value analysis.

Brainstorming: The process of obtaining ideas, opinions, and answers to a question in groups. The members of the group are given the opportunity to contribute.

Branch: A basic block that can be selected for execution based on a program construct in which one of two or more alternative program paths are available, e.g., case, jump, go to, ifthen-else.

Branch condition: See condition.

Branch condition combination coverage: See multiple condition coverage.

Branch condition combination testing: See multiple condition testing.

Branch condition coverage: See condition coverage.

Branch coverage: The percentage of branches that have been exercised by a test suite. 100% branch coverage implies both 100% decision coverage and 100% statement coverage.

Branch testing: A white-box test design technique in which test cases are designed to execute branches.

Budget: A statement of management plans and expected results expressed in numbers, quantitative, and monetary terms.

Bug: See defect.

Build: Builds are versions or redesigns of a project that is in development.

Business process-based testing: An approach to testing in which test cases are designed based on descriptions and/or knowledge of business processes.

Capability maturity model (CMM): A five-level staged framework that describes the key elements of an effective software process. The capability maturity model covers practices for planning, engineering, and managing software development and maintenance. [CMM]

Capability maturity model integration (CMMI): A framework that describes the key elements of an effective product development and maintenance process. The capability maturity model integration covers practices for planning, engineering,

and managing product development and maintenance. CMMI is the designated successor of the CMM. [CMMI]

Capture/playback tool: A type of test execution tool where inputs are recorded during manual testing in order to generate automated test scripts that can be executed later (i.e., replayed). These tools are often used to support automated regression testing.

Capture/replay tool: See capture/playback tool.

CASE: Acronym for Computer Aided Software Engineering.

CAST: Acronym for Computer Aided Software Testing. See also test automation.

Cause-effect analysis: See cause-effect graphing.

Cause-effect decision table: See decision table.

Cause-effect graph: A graphical representation of inputs and/or stimuli (causes) with their associated outputs (effects) which can be used to design test cases.

Cause-effect graphing: A black-box test design technique in which test cases are designed from cause-effect graphs. [BS 7925/2]

Certification: The process of confirming that a component, system, or person complies with its specified requirements, e.g., by passing an exam.

Changeability: The capability of the software product to enable specified modifications to be implemented. [ISO 9126] See also maintainability.

Checker: See reviewer.

Chow's coverage metrics: See N-switch coverage. [Chow]

Class: It provides a blueprint for an object.

Class diagram: A diagram used to show the static view of the system in terms of classes and their relationships.

Classification tree method: A black-box test design technique in which test cases, described by means of a classification tree, are designed to execute combinations of representatives of input and/or output domains. [Grochtmann]

Code analyzer: See static code analyzer.

Code coverage: An analysis method that determines which parts of the software have been executed (covered) by the test suite and which parts have not been executed, e.g., statement coverage, decision coverage, or condition coverage.

Code-based testing: See white-box testing.

Co-existence: The capability of the software product to co-exist with other independent software in a common environment sharing common resources. [ISO 9126] See portability testing.

Collaboration diagram: A diagram that focuses on object interactions irrespective of time.

Commercial off-the-shelf (COTS) software: Hardware and software can be purchased and placed in service without additional development cost for the system or component. See off-the-shelf software.

Comparator: See test comparator.

Comparison/back-to-back testing: It detects test failures by comparing the output of two or more programs complemented to the same specification.

Compatibility testing: See interoperability testing.

Complete testing: See exhaustive testing.

Completion criteria: See exit criteria.

Complexity: The degree to which a component or system has a design and/or internal structure that is difficult to understand, maintain, and verify. See also cyclomatic complexity.

Compliance: The capability of the software product to adhere to standards, conventions, or regulations in laws and similar prescriptions. [ISO 9126]

Compliance testing: The process of testing to determine the compliance of a component or system.

Component: A minimal software item that can be tested in isolation.

Component integration testing: Testing performed to expose defects in the interfaces and interaction between integrated components.

Component specification: A description of a component's function in terms of its output values for specified input values under specified conditions, and required non-functional behavior (e.g., resource-utilization).

Component testing: The testing of individual software components. [After IEEE 610]

Compound condition: Two or more single conditions joined by means of a logical operator (AND, OR, or XOR), e.g., "A>B AND C>1000."

Computer Aided Software Engineering (CASE): Are the automated tools that help a software engineer to automatically perform some tasks which if done manually, are very tedious and cumbersome.

Concrete test case: See low-level test case.

Concurrency testing: Testing to determine how the occurrence of two or more activities within the same interval of time, achieved either by interleaving the activities or by simultaneous execution, is handled by the component or system. [After IEEE 610]

Condition: A logical expression that can be evaluated as True or False, e.g., A>B. See also test condition.

Condition combination coverage: See multiple condition coverage.

Condition combination testing: See multiple condition testing.

Condition coverage: The percentage of condition outcomes that have been exercised by a test suite. 100% condition coverage requires each single condition in every decision statement to be tested as True and False.

Condition determination coverage: The percentage of all single condition outcomes that independently affect a decision outcome that have been exercised by a test case suite. 100% condition determination coverage implies 100% decision condition coverage.

Condition determination testing: A white-box test design technique in which test cases are designed to execute single condition outcomes that independently affect a decision outcome.

Condition outcome: The evaluation of a condition to True or False.

Condition testing: A white-box test design technique in which test cases are designed to execute condition outcomes.

Confidence test: See smoke test.

Configuration: The composition of a component or system as defined by the number, nature, and interconnections of its constituent parts.

Configuration auditing: The function to check on the contents of libraries of configuration items, e.g., for standards compliance. [IEEE 610]

Configuration control: An element of configuration management, consisting of the evaluation, coordination, approval or disapproval, and implementation of changes to configuration items after formal establishment of their configuration identification. [IEEE 610]

Configuration identification: An element of configuration management, consisting of selecting the configuration items for a system and recording their functional and physical characteristics in technical documentation. [IEEE 610]

Configuration item: An aggregation of hardware, software, or both, that is designated for configuration management and treated as a single entity in the configuration management process. [IEEE 610]

Configuration management: A discipline applying technical and administrative direction and surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements. [IEEE 610]

Configuration testing: See portability testing.

Confirmation testing: See retesting.

Conformance testing: See compliance testing.

Consistency: The degree of uniformity, standardization, and freedom from contradiction among the documents or parts of a component or system. [IEEE 610]

Control flow: An abstract representation of all possible sequences of events (paths) in the execution through a component or system.

Control flow graph: See control flow.

Control flow path: See path.

Conversion testing: Testing of software used to convert data from existing systems for use in replacement systems.

Correctness: The extent to which software is free from faults.

Cost-benefit analysis: The process of deciding whether to do something by evaluating the costs of doing it and the benefits of doing it.

COTS: Acronym for Commercial off-the-shelf software.

Coverage: The degree to which a software feature or characteristic is tested or analyzed. A measure of test completeness. The degree, expressed as a percentage, to which a specified coverage item has been exercised by a test suite.

Coverage analysis: Measurement of achieved coverage to a specified coverage item during test execution referring to predetermined criteria to determine whether additional testing is required and, if so, which test cases are needed.

Coverage item: An entity or property used as a basis for test coverage, e.g., equivalence partitions or code statements.

Coverage tool: A tool that provides objective measures of what structural elements, e.g., statements and branches have been exercised by the test suite.

Critical path method (CPM): A method that shows the analysis of paths in an activity graph among different milestones of the project.

Custom software: Software that is developed to meet the specific needs of a particular customer. See bespoke software.

Cyclomatic complexity: The number of independent paths through a program. Cyclomatic complexity is defined as: L – N + 2P, where – L = the number of edges/links in a graph – N = the number of nodes in a graph – P = the number of disconnected parts of the graph (e.g., a calling graph and a subroutine). [After McCabe]

Cyclomatic number: See cyclomatic complexity.

Data definition: An executable statement where a variable is assigned a value.

Data driven testing: A scripting technique that stores test input and expected results in a table or spreadsheet, so that a single control script can execute all of the tests in the table. Data driven testing is often used to support the application of test execution tools such as capture/playback tools. [Fewster and Graham] See also keyword driven testing.

Data flow: An abstract representation of the sequence and possible changes of the state of data objects, where the state of an object is a creation, usage, or destruction. [Beizer]

Data flow analysis: A form of static analysis based on the definitions and usage of variables.

Data flow coverage: The percentage of definition-use pairs that have been exercised by a test case suite.

Data flow test: A white-box test design technique in which test cases are designed to execute definitions and use pairs of variables.

Dead code: See unreachable code.

Debugger: See debugging tool.

Debugging: The process of finding, analyzing, and removing the causes of failures in software.

Debugging tool: A tool used by programmers to reproduce failures, investigate the state of programs, and find the corresponding defect. Debuggers enable programmers to execute programs step by step to halt a program at any program statement and to set and examine program variables.

Decision: A program point at which the control flow has two or more alternative routes. A node with two or more links to separate branches.

Decision condition coverage: The percentage of all condition outcomes and decision outcomes that have been exercised by a test suite. 100% decision condition coverage implies both 100% condition coverage and 100% decision coverage.

Decision condition testing: A white-box test design technique in which test cases are designed to execute condition outcomes and decision outcomes.

Decision coverage: The percentage of decision outcomes that have been exercised by a test suite. 100% decision coverage implies both 100% branch coverage and 100% statement coverage.

Decision outcome: The result of a decision (which therefore determines the branches to be taken).

Decision table: A table showing combinations of inputs and/or stimuli (causes) with their associated outputs and/or actions (effects) which can be used to design test cases. It lists various decision variables, the conditions assumed by each of the decision variables, and the actions to take in each combination of conditions.

Decision table testing: A black-box test design technique in which test cases are designed to execute the combinations of inputs and/or stimuli (causes) shown in a decision table. [Veenendaal]

Decision testing: A white-box test design technique in which test cases are designed to execute decision outcomes.

Defect: A flaw in a component or system that can cause the component or system to fail to perform its required function, e.g., an incorrect statement or data definition. A defect, if encountered during execution, may cause a failure of the component or system.

Defect bash: It is an adhoc testing, done by people performing different roles in the same time duration during the integration testing phase.

Defect density: The number of defects identified in a component or system divided by the size of the component or system (expressed in standard measurement terms, e.g., lines-of code, number of classes, or function points).

Defect detection percentage (DDP): The number of defects found by a test phase, divided by the number found by that test phase and any other means afterwards.

Defect management: The process of recognizing, investigating, taking action, and disposing of defects. It involves recording defects, classifying them, and identifying the impact. [After IEEE 1044]

Defect management tool: See incident management tool.

Defect masking: An occurrence in which one defect prevents the detection of another. [After IEEE 610]

Defect report: A document reporting on any flaw in a component or system that can cause the component or system to fail to perform its required function. [After IEEE 829]

Defect tracking tool: See incident management tool.

Definition-use pair: The association of the definition of a variable with the use of that variable. Variable uses include computational (e.g., multiplication) or to direct the execution of a path ("predicate" use).

Deliverable: Any (work) product that must be delivered to someone other than the (work) product's author.

Delphi technique: Several participants make their individual cost estimates and then share them.

Design-based testing: An approach to testing in which test cases are designed based on the architecture and/or detailed design of a component or system (e.g., tests of interfaces between components or systems).

Desk checking: Testing of software or specification by manual simulation of its execution. A manual analysis of a work product to discover errors.

Development testing: Formal or informal testing conducted during the implementation of a component or system, usually in the development environment by developers. [After IEEE 610]

Deviation: See incident.

Deviation report: See incident report.

Dirty testing: See negative testing.

Divide-and-conquer (or Decomposition): The process of dividing something large into smaller units (called modules) so as to simplify our tasks. It is appliciable to cost estimation, design, and testing.

Documentation testing: To ensure that the documentation is consistent with the product. Testing the quality of the documentation, e.g., user guide or installation guide.

Domain: The set from which valid input and/or output values can be selected.

Domain testing: The execution of test cases developed by analysis of input data relationships and constraints. Domain testing exploits the tester's domain knowledge to test the suitability of the product to what the users do on a typical day.

Driver: In bottom-up integration, we start with the leaves of the decomposition tree and test them with specially coded drivers. Less throw-away code exists in drivers than there is in stubs. A software component or test tool that replaces a component that takes care of the control and/or the calling of a component or system. [After TMap]

Duplex testing: A test bed where the test driver runs on a computer system separate from the computer system where the system-under-test (SUT) runs.

Dynamic analysis: The process of evaluating behavior, e.g., memory performance, CPU usage, of a system, or component during execution. [After IEEE 610]

Dynamic comparison: Comparison of actual and expected results, performed while the software is being executed, for example, by a test execution tool.

Dynamic testing: Testing that involves the execution of the software of a component or system.

Efficiency: The capability of the software product to provide appropriate performance, relative to the amount of resources used under stated conditions. [ISO 9126]

Efficiency testing: The process of testing to determine the efficiency of a software product.

Elementary comparison testing: A black-box test design technique in which test cases are designed to execute combinations of inputs using the concept of condition determination coverage. [TMap]

Embedded software: A software to run in specific hardware devices. It is embedded in ROM.

Emulator: A device, computer program, or system that accepts the same inputs and produces the same outputs as a given system. [IEEE 610] See also simulator.

Encapsulation: The wrapping up of data and function into a single unit.

Entry criteria: The set of generic and specific conditions for permitting a process to go forward with a defined task, e.g., test phase. The purpose of entry criteria is to prevent a task from starting which would entail more (wasted) effort compared to the effort needed to remove the failed entry criteria. [Gilb and Graham]

Entry point: The first executable statement within a component.

Equivalence class: See equivalence partition.

Equivalence partition: A portion of an input or output domain for which the behavior of a component or system is assumed to be the same, based on the specification.

Equivalence partition coverage: The percentage of equivalence partitions that have been exercised by a test suite.

Equivalence partitioning: A black-box test design technique in which test cases are designed to execute representatives from equivalence partitions. In principle, test cases are designed to cover each partition at least once.

Error: A human action that produces an incorrect result. [After IEEE 610]

Error guessing: A test design technique where the experience of the tester is used to anticipate what defects might be present in the component or system-under-test as a result of errors made, and to design tests specifically to expose them.

Error seeding: It determines whether a set of test cases is adequate by inserting known error types into the program and executing it with test cases. The process of intentionally adding known defects to those already in the component or system for the purpose of monitoring the rate of detection and removal, and estimating the number of remaining defects. [IEEE 610]

Error tolerance: The ability of a system or component to continue normal operation despite the presence of erroneous inputs. [After IEEE 610]

Evaluation: See testing.

Event: Something that causes a system or object to change state.

Exception handling: Behavior of a component or system in response to erroneous input, from either a human user or from another component or system, or to an internal failure.

Executable statement: A statement which, when compiled, is translated into object code, and which will be executed procedurally when the program is running and may perform an action on data.

Exercised: A program element is said to be exercised by a test case when the input value causes the execution of that element, such as a statement, decision, or other structural element.

Exhaustive testing: A test approach in which the test suite comprises all combinations of input values and preconditions.

Exit criteria: The set of generic and specific conditions, agreed upon with the stakeholders, for permitting a process to be officially completed. The purpose of exit criteria is to prevent a task from being considered completed when there are still outstanding parts of the task which have not been finished. Exit criteria are used by testing to report against and to plan when to stop testing. [After Gilb and Graham]

Exit point: The last executable statement within a component.

Expected outcome: See expected result.

Expected result: The behavior predicted by the specification, or another source, of the component or system under specified conditions.

Exploratory testing: Testing where the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests. [Bach]

Fail: A test is deemed to fail if its actual result does not match its expected result.

Failure: Actual deviation of the component or system from its expected delivery, service, or result. [After Fenton]

Failure mode: The physical or functional manifestation of a failure. For example, a system in failure mode may be characterized by slow operation, incorrect outputs, or complete termination of execution.

Failure mode and effect analysis (FMEA): A systematic approach to risk identification and analysis of identifying possible modes of failure and attempting to prevent their occurrence.

Failure rate: The ratio of the number of failures of a given category to a given unit of measure, e.g., failures per unit of time, failures per number of transactions, failures per number of computer runs. [IEEE 610]

Fault: See defect.

Fault density: See defect density.

Fault detection percentage (FDP): See Defect detection percentage (DDP).

Fault masking: See defect masking.

Fault spawing: The introduction of new faults when a fault is removed.

Fault tolerance: The capability of the software product to maintain a specified level of performance in cases of software faults (defects) or of infringement of its specified interface. [ISO 9126] See also reliability.

Fault tree analysis: A method used to analyze the causes of faults (defects).

Feasible path: A path for which a set of input values and preconditions exists which causes it to be executed.

Feature: An attribute of a component or system specified or implied by requirements documentation (for example, reliability, usability, or design constraints). [After IEEE 1008]

Field testing: See beta testing.

Finite state machine: A computational model consisting of a finite number of states and transitions between those states, possibly with accompanying actions. [IEEE 610]

Finite state testing: See state transition testing.

Fork: A symbol in an activity diagram to show splitting of control into multiple threads.

Formal language: A language that uses mathematics for the purpose of modelling.

Formal review: A review characterized by documented procedures and requirements, e.g., inspection. Careful planned meetings, reviewers are responsible and review reports are also generated and acted upon.

Formal testing: Testing conducted in accordance with test plans and procedures that have been reviewed and approved by a customer, user, or designated level of management.

Frozen test basis: A test basis document that can only be amended by a formal change control process. See also baseline.

Function point analysis (FPA): Method aiming to measure the size of the functionality of an information system. The measurement is independent of the technology. This measurement may be used as a basis for the measurement of productivity, the estimation of the needed resources, and project control.

Functional integration: An integration approach that combines the components or systems for the purpose of getting a basic functionality working early. See also integration testing.

Functional requirement: A requirement that specifies a function that a component or system must perform. [IEEE 610]

Functional test design technique: Documented procedure to derive and select test cases based on an analysis of the specification of the functionality of a component or system without reference to its internal structure. See also black-box test design technique.

Functional testing: Testing based on an analysis of the specification of the functionality of a component or system. See also black-box testing.

Functionality: The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions. [ISO 9126]

Functionality testing: The process of testing to determine the functionality of a software product.

Gantt chart: A diagram used to graphically represent the start and end dates of each software engineering task. These charts can be drawn using MS-Project as a CASE tool.

Generalization: It is a form of abstraction that specifies that 2 or more entities that share common attributes can be generalized into a higher level entity type called a super type or generic entity.

Generic software: Softwares that perform functions on general purpose computers.

Glass-box testing: See white-box testing.

Glue code: Code that is written to connect reused commercial off-the-shelf applications.

Gold plating: Building a list of requirements that does more than needed.

Gray-box testing: A combined (or hybrid) approach of white- and black-box techniques. Gray-box may involve 95% of white-box testing strategies and 5% of black-box and vice versa is also true.

Guard condition: A condition that determines whether a certain transition will occur in a state diagram when an event happens.

Heuristic evaluation: A static usability test technique to determine the compliance of a user interface with recognized usability principles (the so-called "heuristics").

High-level test case: A test case without concrete (implementation level) values for input data and expected results.

Horizontal traceability: The tracing of requirements for a test level through the layers of test documentation (e.g., test plan, test design specification, test case specification, and test procedure specification).

Impact analysis: The assessment of change to the layers of development documentation, test documentation, and components in order to implement a given change to specified requirements.

Incident: Any event occurring during testing that requires investigation. [After IEEE 1008]

Incident: It is the symptom associated with a failure that alerts the user to the occurrence of a failure.

Incident management: The process of recognizing, investigating, taking action, and disposing of incidents. It involves recording incidents, classifying them, and identifying the impact. [After IEEE 1044]

Incident management tool: A tool that facilitates the recording and status tracking of incidents found during testing. They often have workflow-oriented facilities to track and control the allocation, correction, and re-testing of incidents and provide reporting facilities.

Incident report: A document reporting on any event that occurs during the testing which requires investigation. [After IEEE 829]

Incremental development model: A development life cycle where a project is broken into a series of increments, each of which delivers a portion of the functionality in the overall project requirements. The requirements are prioritized and delivered in priority order in the appropriate increment. In some (but not all) versions of this life cycle model, each subproject follows a "mini-model" with its own design, coding, and testing phases.

Incremental testing: Testing where components or systems are integrated and tested one or some at a time until all of the components or systems are integrated and tested.

Independence: Separation of responsibilities, which encourages the accomplishment of objective testing. [After DO-178b]

Infeasible path: A path that cannot be exercised by any set of possible input values.

Informal review: No planned meetings, reviewers have no responsibility and they do not produce a review reports. A review not based on a formal (documented) procedure.

Informal testing: Testing conducted in accordance with test plans and procedures that have not been reviewed and approved by a customer, user, or designated level of management.

Input: A variable (whether stored within a component or outside) that is read by a component.

Input domain: The set from which valid input values can be selected. See also domain.

Input value: An instance of an input. See also input.

Inspection: A type of review that relies on visual examination of documents to detect defects, e.g., violations of development standards and non-conformance to higher level documentation. The most formal review technique and therefore always based on a documented procedure. [After IEEE 610, IEEE 1028]

Inspection leader: See moderator.

Inspector: See reviewer.

Installability: The capability of the software product to be installed in a specified environment. [ISO 9126] See also portability.

Installability testing: The process of testing the installability of a software product. See also portability testing.

Installation guide: Supplied instructions on any suitable media which guides the installer through the installation process. This may be a manual guide, step-by-step procedure, installation wizard, or any other similar process description.

Installation wizard: Supplied software on any suitable media which leads the installer through the installation process. It normally runs the installation process, provides feedback on installation results, and prompts for options.

Instrumentation: The insertion of additional code into the program in order to collect information about program behavior during execution.

Instrumenter: A software tool used to carry out instrumentation.

Intake test: A special instance of a smoke test to decide if the component or system is ready for detailed and further testing. An intake test is typically carried out at the start of the test execution phase.

Integration: The process of combining components or systems into larger assemblies.

Integration testing: Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems. See also component integration testing and system integration testing.

Integration testing in the large: See system integration testing.

Integration testing in the small: See component integration testing.

Interaction diagram: A sequence diagram or collaboration diagram used to model the dynamic aspects of software.

Interface testing: An integration test type that is concerned with testing the interfaces between components or systems.

Interoperability: The effort required to couple one system to another. The capability of the software product to interact with one or more specified components or systems. [After ISO 9126] See also functionality.

Interoperability testing: The process of testing to determine the interoperability of a software product. See also functionality testing.

Invalid testing: Testing using input values that should be rejected by the component or system. See also error tolerance.

Isolation testing: Testing of individual components in isolation from surrounding components, with surrounding components being simulated by stubs and drivers, if needed.

Item transmittal report: See release note.

Key performance indicator: See performance indicator.

Keyword driven testing: A scripting technique that uses data files to contain not only test data and expected results, but also keywords related to the application being tested. The keywords are interpreted by special supporting scripts that are called by the control script for the test. See also data driven testing.

LCSAJ: A linear code sequence and jump, consisting of the following three items (conventionally identified by line numbers in a source code listing): the start of the linear sequence of executable statements, the end of the linear sequence, and the target line to which control flow is transferred at the end of the linear sequence.

LCSAJ coverage: The percentage of LCSAJs of a component that have been exercised by a test suite. 100% LCSAJ coverage implies 100% decision coverage.

LCSAJ testing: A white-box test design technique in which test cases are designed to execute LCSAJs.

Learnability: The capability of the software product to enable the user to learn its application. [ISO 9126] See also usability.

Limited entry decision tables: Decision tables in which all conditions are binary.

Link testing: See component integration testing.

Load test: A test type concerned with measuring the behavior of a component or system with increasing load, e.g., number of parallel users and/or numbers of transactions to determine what load can be handled by the component or system.

Locale: An environment where the language, culture, laws, currency, and many other factors may be different.

Locale testing: It focuses on testing the conventions for number, punctuations, date and time, and currency formats.

Logic-coverage testing: See white-box testing. [Myers]

Logic-driven testing: See white-box testing.

Logical test case: See high-level test case.

Low-level test case: A test case with concrete (implementation level) values for input data and expected results.

Maintainability: The ease with which a software product can be modified to correct defects, modified to meet new requirements, modified to make future maintenance easier, or adapted to a changed environment. [ISO 9126]

Maintainability testing: The process of testing to determine the maintainability of a software product.

Maintenance: Modification of a software product after delivery to correct defects, to improve performance or other attributes, or to adapt the product to a modified environment. [IEEE 1219]

Maintenance testing: Testing the changes to an operational system or the impact of a changed environment to an operational system.

Management review: A systematic evaluation of software acquisition, supply, development, operation, or maintenance process, performed by or on behalf of management that monitors progress, determines the status of plans and schedules, confirms requirements and their system allocation, or evaluates the effectiveness of management approaches to achieve fitness for purpose. [After IEEE 610, IEEE 1028]

Mandel bug: A bug whose underlying causes are so complex and obscure as to make its behavior appear chaotic or even non-deterministic.

Master test plan: See project test plan.

Maturity: (1) The capability of an organization with respect to the effectiveness and efficiency of its processes and work practices. See also capability maturity model and test maturity model. (2) The capability of the software product to avoid failure as a result of defects in the software. [ISO 9126] See also reliability.

Measure: The number or category assigned to an attribute of an entity by making a measurement. [ISO 14598]

Measurement: The process of assigning a number or category to an entity to describe an attribute of that entity. [ISO 14598]

Measurement scale: A scale that constrains the type of data analysis that can be performed on it. [ISO 14598]

Memory leak: A situation in which a program requests memory but does not release it when it is no longer needed. A defect in a program's dynamic store allocation logic that causes it to fail to reclaim memory after it has finished using it, eventually causing the program to fail due to lack of memory.

Message: Message is a programming language mechanism by which one unit transfers control to another unit.

Messages: It shows how objects communicate. Each message represents one object making function call of another.

Metric: A measurement scale and the method used for measurement. [ISO 14598]

Migration testing: See conversion testing.

Milestone: A point in time in a project at which defined (intermediate) deliverables and results should be ready.

Mistake: See error.

Moderator: The leader and main person responsible for an inspection or other review process.

Modified condition decision coverage: See condition determination coverage.

Modified condition decision testing: See condition determination coverage testing.

Modified multiple condition coverage: See condition determination coverage.

Modified multiple condition testing: See condition determination coverage testing.

Module: Modules are parts, components, units, or areas that comprise a given project. They are often thought of as units of software code. See also component.

Module testing: See component testing.

Monitor: A software tool or hardware device that runs concurrently with the component or system under test and supervises, records, and/or analyzes the behavior of the component or system. [After IEEE 610]

Monkey testing: Randomly test the product after all planned test cases are done.

Multiple condition: See compound condition.

Multiple condition coverage: The percentage of combinations of all single condition outcomes within one statement that have been exercised by a test suite. 100% multiple condition coverage implies 100% condition determination coverage.

Multiple condition testing: A white-box test design technique in which test cases are designed to execute combinations of single condition outcomes (within one statement).

Multiplicity: Information placed at each end of an association indicating how many instances of one class can be related to instances of the other class.

Mutation analysis: A method to determine test suite thoroughness by measuring the extent to which a test suite can discriminate the program from slight variants (mutants) of the program.

N-switch coverage: The percentage of sequences of N+1 transitions that have been exercised by a test suite. [Chow]

N-switch testing: A form of state transition testing in which test cases are designed to execute all valid sequences of N+1 transitions. [Chow] See also state transition testing.

Negative functional testing: Testing the software with invalid inputs.

Negative testing: Tests aimed at showing that a component or system does not work. Negative testing is related to the testers' attitude rather than a specific test approach or test design technique. [After Beizer]

Nonconformity: Non fulfillment of a specified requirement. [ISO 9000]

Nonfunctional requirement: A requirement that does not relate to functionality, but to attributes of such as reliability, efficiency, usability, maintainability, and portability.

Nonfunctional testing: Testing the attributes of a component or system that do not relate to functionality, e.g., reliability, efficiency, usability, maintainability, and portability.

Nonfunctional test design techniques: Methods used to design or select tests for nonfunctional testing.

Non-reentrant code: It is a piece of program that modifies itself.

Object: It encapsulates information and behavior. It represents a real-world thing, e.g., ATM screen, card reader, etc.

Off point: A value outside of a domain.

Off-the-shelf software: A software product that is developed for the general market, i.e., for a large number of customers, and that is delivered to many customers in identical format.

Operability: The capability of the software product to enable the user to operate and control it. [ISO 9126] See also usability.

Operational environment: Hardware and software products installed at users' or customers' sites where the component or system-under-test will be used. The software may include operating systems, database management systems, and other applications.

Operational profile testing: Statistical testing using a model of system operations (short duration tasks) and their probability of typical use. [Musa]

Operational testing: Testing conducted to evaluate a component or system in its operational environment. [IEEE 610]

Oracle: See test oracle.

Outcome: See result.

Output: A variable (whether stored within a component or outside) that is written by a component.

Output domain: The set from which valid output values can be selected. See also domain.

Output value: An instance of an output. See also output.

Pair programming: A software development approach whereby lines of code (production and/ or test) of a component are written by two programmers sitting at a single computer. This implicitly means ongoing real-time code reviews are performed.

Pair testing: Two testers work together to find defects. Typically, they share one computer and trade control of it while testing.

Pareto Principle: A rule that states that 80% of the benefit can be obtained with 20% of the work. For example, 80% of CPU time is spent executing 20% of the statements.

Partition testing: See equivalence partitioning. [Beizer]

Pass: A test is deemed to pass if its actual result matches its expected result.

Pass/fail criteria: Decision rules used to determine whether a test item (function) or feature has passed or failed a test. [IEEE 829]

Path: A sequence of events, e.g., executable statements, of a component or system from an entry point to an exit point.

Path coverage: The percentage of paths that have been exercised by a test suite. 100% path coverage implies 100% LCSAJ coverage.

Path sensitizing: Choosing a set of input values to force the execution of a given path.

Path testing: A white-box test design technique in which test cases are designed to execute paths.

Peer review: See technical review.

Performance: The degree to which a system or component accomplishes its designated functions within given constraints regarding processing time and throughput rate. [After IEEE 610] See efficiency.

Performance indicator: A high level metric of effectiveness and/or efficiency used to guide and control progressive development, e.g., defect detection percentage (DDP) for testing. [CMMI]

Performance testing: The process of testing to determine the performance of a software product. See efficiency testing.

Performance testing tool: A tool to support performance testing and that usually has two main facilities: load generation and test transaction measurement. Load generation can simulate either multiple users or high volumes of input data. During execution, response time measurements are taken from selected transactions and these are logged. Performance testing tools normally provide reports based on test logs and graphs of load against response times.

Person-month: One person-month is the amount of work done by one person in one month if they are working full time.

Phase test plan: A test plan that typically addresses one test level.

Polymorphism (*poly* = many, *morphs* = forms): Many forms of a same function is polymorphism. The compiler resolves such issues based on either the total number of parameters or their data types.

Portability: The ease with which the software product can be transferred from one hardware or software environment to another. [ISO 9126]

Portability testing: The process of testing to determine the portability of a software product.

Positive functional testing: Testing the software with valid inputs.

Post-execution comparison: Comparison of actual and expected results, performed after the software has finished running.

Postcondition: Environmental and state conditions that must be fulfilled after the execution of a test or test procedure.

Precondition: Environmental and state conditions that must be fulfilled before the component or system can be executed with a particular test or test procedure.

Predicted outcome: See expected result.

Pretest: See intake test.

Priority: The level of (business) importance assigned to an item, e.g., defect.

Problem: See defect.

Problem management: See defect management.

Problem report: See defect report.

Process: A set of interrelated activities which transform inputs into outputs. [ISO 12207]

Process cycle test: A black-box test design technique in which test cases are designed to execute business procedures and processes. [TMap]

Production environment: It refers to the environment in which the final software will run.

Program instrumenter: See instrumenter.

Program testing: See component testing.

Project: A project is a unique set of coordinated and controlled activities with start and finish dates undertaken an objective conforming to specific requirements, including the constraints of time, cost, and resources. [ISO 9000]

Project Evaluation Review Technique (PERT): It shows different project task activities and their relationship with each other.

Project test plan: A test plan that typically addresses multiple test levels.

Proof of correctness: A formal technique to prove mathematically that a program satisfies its specifications.

Prototyping: It helps to examine the probable results of implementing software requirements.

Pseudo-random: A series which appears to be random but is in fact generated according to some prearranged sequence.

Quality: The degree to which a component, system, or process meets specified requirements and/or user/customer needs and expectations. [After IEEE 610]

Quality assurance: Part of quality management focused on providing confidence that quality requirements will be fulfilled. [ISO 9000]

Quality attribute: A feature or characteristic that affects an item's quality. [IEEE 610]

Quality characteristic: See quality attribute.

Quality management: Coordinated activities to direct and control an organization with regard to quality. Direction and control with regard to quality generally includes the establishment of the quality policy and quality objectives, quality planning, quality control, quality assurance, and quality improvement. [ISO 9000]

Random testing: A black-box test design technique where test cases are selected, possibly using a pseudo-random generation algorithm, to match an operational profile. This technique can be used for testing nonfunctional attributes such as reliability and performance.

Rapid (or throwaway prototyping): This approach is to construct a "quick and dirty" partial solution to the system prior to requirements stage.

Real-time software: Software in which we have strict time constraints.

Recorder: See scribe.

Record/playback tool: See capture/playback tool.

Recoverability: The capability of the software product to re-establish a specified level of performance and recover the data directly affected in case of failure. [ISO 9126] See also reliability.

Recoverability testing: The process of testing to determine the recoverability of a software product. See also reliability testing.

Recovery testing: See recoverability testing.

Reentrant code: It is a piece of program that does not modify itself.

Regression testing: The process of retesting a system after changes have been made to it. Testing of a previously tested program following modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software, as a result of the changes made. It is performed when the software or its environment is changed.

Release (or golden master): The build that will eventually be shipped to the customer, posted on the Web, or migrated to the live Web site.

Release note: A document identifying test items, their configuration, current status, and other delivery information delivered by development to testing, and possibly other stakeholders, at the start of a test execution phase. [After IEEE 829]

Reliability: Probability of failure free operation of software for a specified time under specified operating conditions. The ability of the software product to perform its required functions under stated conditions for a specified period of time, or for a specified number of operations. [ISO 9126]

Reliability testing: The process of testing to determine the reliability of a software product.

Replaceability: The capability of the software product to be used in place of another specified software product for the same purpose in the same environment. [ISO 9126] See also portability.

Requirement: A condition or capability needed by a user to solve a problem or achieve an objective that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document. [After IEEE 610]

Requirements-based testing: An approach to testing in which test cases are designed based on test objectives and test conditions derived from requirements, e.g., tests that exercise specific functions or probe non functional attributes such as reliability or usability.

Requirements management tool: A tool that supports the recording of requirements, requirements attributes (e.g., priority, knowledge responsible) and annotation, and facilitates traceability through layers of requirements and requirements change management. Some requirements management tools also provide facilities for static analysis, such as consistency checking and violations to pre-defined requirements rules.

Requirements phase: The period of time in the software life cycle during which the requirements for a software product are defined and documented. [IEEE 610]

Requirements tracing: It is a technique of ensuring that the product, as well as the testing of the product, addresses each of its requirements.

Resource utilization: The capability of the software product to use appropriate amounts and types of resources, for example, the amounts of main and secondary memory used by the program and the sizes of required temporary or overflow files, when the software performs its function under stated conditions. [After ISO 9126] See also efficiency.

Resource utilization testing: The process of testing to determine the resource utilization of a software product.

Result: The consequence/outcome of the execution of a test. It includes outputs to screens, changes to data, reports, and communication messages sent out. See also actual result and expected result.

Resumption criteria: The testing activities that must be repeated when testing is restarted after a suspension. [After IEEE 829]

Retesting: Testing that runs test cases that failed the last time they were run, in order to verify the success of corrective actions.

Review: An evaluation of a product or project status to ascertain discrepancies from planned results and to recommend improvements. Examples include management review, informal review, technical review, inspection, and walkthrough. [After IEEE 1028]

Reviewer: The person involved in the review who shall identify and describe anomalies in the product or project under review. Reviewers can be chosen to represent different viewpoints and roles in the review process.

Risk: A factor that could result in future negative consequences; usually expressed as impact and likelihood. The possibility of loss or injury; a problem that might occur.

Risk analysis: The process of assessing identified risks to estimate their impact and probability of occurrence (likelihood).

Risk-based testing: Testing oriented towards exploring and providing information about product risks. [After Gerrard]

Risk control: The process through which decisions are reached and protective measures are implemented for reducing risks to, or maintaining risks within, specified levels.

Risk identification: The process of identifying risks using techniques such as brainstorming, checklists, and failure history.

Risk management: Systematic application of procedures and practices to the tasks of identifying, analyzing, prioritizing, and controlling risk.

Risk mitigation: See risk control.

Robustness: The degree to which a component or system can function correctly in the presence of invalid inputs or stressful environmental conditions. [IEEE 610] See also error tolerance and fault-tolerance.

Root cause: An underlying factor that caused a nonconformance and possibly should be permanently eliminated through process improvement.

Root cause analysis: The process of determining the ultimate reason why a software engineer made the error that introduced a defect.

Safety: The capability of the software product to achieve acceptable levels of risk of harm to people, business, software, property, or the environment in a specified context of use. [ISO 9126]

Safety testing: The process of testing to determine the safety of a software product.

Sanity test: See smoke test.

Scalability: The capability of the software product to be upgraded to accommodate increased loads. [After Gerrard]

Scalability testing: Testing to determine the scalability of the software product.

Scenario testing: See use-case testing.

Scribe: The person who has to record each defect mentioned and any suggestions for improvement during a review meeting on a logging form. The scribe has to make sure that the logging form is readable and understandable.

Scripting language: A programming language in which executable test scripts are written, used by a test execution tool (e.g., a capture/replay tool).

Security: Attributes of software products that bear on its ability to prevent unauthorized access, whether accidental or deliberate, to programs and data. [ISO 9126]

Security testing: Testing to determine the security of the software product.

Serviceability testing: See maintainability testing.

Severity: The degree of impact that a defect has on the development or operation of a component or system. [After IEEE 610]

Shelfware: Software that is not used.

Simulation: A technique that uses an executable model to examine the behavior of the software. The representation of selected behavioral characteristics of one physical or abstract system by another system. [ISO 2382/1]

Simulator: A device, computer program, or system used during testing, which behaves or operates like a given system when provided with a set of controlled inputs. [After IEEE 610, DO178b] See also emulator.

Sink node: It is a statement fragment at which program execution terminates.

Slicing: It is a program decomposition technique used to trace an output variable back through the code to identify all code statements relevant to a computation in the program.

Smoke test: It is a condensed version of a regression test suite. A subset of all defined/planned test cases that cover the main functionality of a component or system, to ascertaining that the most crucial functions of a program work, but not bothering with finer details. A daily build and smoke test are among industry best practices. See also intake test.

Software feature: See feature.

Software quality: The totality of functionality and features of a software product that bear on its ability to satisfy stated or implied needs. [After ISO 9126]

Software quality characteristic: See quality attribute.

Software runaways: Large size projects failed due to lack of usage of systematic techniques and tools.

Software test incident: See incident.

Software test incident report: See incident report.

Software Usability Measurement Inventory (SUMI): A questionnaire based usability test technique to evaluate the usability, e.g., user-satisfaction, of a component or system. [Veenendaal]

Source node: A source node in a program is a statement fragment at which program execution begins or resumes.

Source statement: See statement.

Specialization: The process of taking subsets of a higher-level entity set to form lower-level entity sets.

Specification: A document that specifies, ideally in a complete, precise, and verifiable manner, the requirements, design, behavior, or other characteristics of a component or system, and, often, the procedures for determining whether these provisions have been satisfied. [After IEEE 610]

Specification-based test design technique: See black-box test design technique.

Specification-based testing: See black-box testing.

Specified input: An input for which the specification predicts a result.

Stability: The capability of the software product to avoid unexpected effects from modifications in the software. [ISO 9126] See also maintainability.

Standard software: See off-the-shelf software.

Standards testing: See compliance testing.

State diagram: A diagram that depicts the states that a component or system can assume, and shows the events or circumstances that cause and/or result from a change from one state to another. [IEEE 610]

State table: A grid showing the resulting transitions for each state combined with each possible event, showing both valid and invalid transitions.

State transition: A transition between two states of a component or system.

State transition testing: A black-box test design technique in which test cases are designed to execute valid and invalid state transitions. See also N-switch testing.

Statement: An entity in a programming language, which is typically the smallest indivisible unit of execution.

Statement coverage: The percentage of executable statements that have been exercised by a test suite.

Statement testing: A white-box test design technique in which test cases are designed to execute statements.

Static analysis: Analysis of software artifacts, e.g., requirements or code, carried out without execution of these software artifacts.

Static analyzer: A tool that carries out static analysis.

Static code analysis: Analysis of program source code carried out without execution of that software.

Static code analyzer: A tool that carries out static code analysis. The tool checks source code, for certain properties such as conformance to coding standards, quality metrics, or data flow anomalies.

Static testing: Testing of a component or system at specification or implementation level without execution of that software, e.g., reviews or static code analysis.

Statistical testing: A test design technique in which a model of the statistical distribution of the input is used to construct representative test cases. See also operational profile testing.

Status accounting: An element of configuration management, consisting of the recording and reporting of information needed to manage a configuration effectively. This information includes a listing of the approved configuration identification, the status of proposed changes to the configuration, and the implementation status of the approved changes. [IEEE 610]

Storage: See resource utilization.

Storage testing: See resource utilization testing.

Stress testing: Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements. [IEEE 610]

Structural coverage: Coverage measures based on the internal structure of the component.

Structural test design technique: See white-box test design technique.

Structural testing: See white-box testing.

Structured walkthrough: See walkthrough.

Stub: A skeletal or special-purpose implementation of a software component, used to develop or test a component that calls or is otherwise dependent on it. It replaces a called component. [After IEEE 610]

Stubs: These are the dummy modules or pieces of throw-away code that emulate a called unit. Top-down integration begins with the main program (root of the tree). Any lower-level unit that is called by the main program appears as a stub.

Subpath: A sequence of executable statements within a component.

Suitability: The capability of the software product to provide an appropriate set of functions for specified tasks and user objectives. [ISO 9126] See also functionality.

Suspension criteria: The criteria used to (temporarily) stop all or a portion of the testing activities on the test items. [After IEEE 829]

Syntax testing: A black-box test design technique in which test cases are designed based upon the definition of the input domain and/or output domain.

System: A collection of components organized to accomplish a specific function or set of functions. [IEEE 610]

System integration testing: Testing the integration of systems and packages; testing interfaces to external organizations (e.g., electronic data interchange, Internet).

System testing: The process of testing an integrated system to verify that it meets specified requirements. [Hetzel]

Technical review: A peer group discussion activity that focuses on achieving consensus on the technical approach to be taken. A technical review is also known as a peer review. [Gilb and Graham, IEEE 1028]

Technology transfer: The awareness, convincing, selling, motivating, collaboration, and special effort required to encourage industry, organizations, and projects to make good use of new technology products.

Test: A test is the act of exercising software with test cases. A set of one or more test cases. [IEEE 829]

Test approach: The implementation of the test strategy for a specific project. It typically includes the decisions made that follow based on the (test) project's goal and the risk assessment carried out, starting points regarding the test process, the test design techniques to be applied, exit criteria, and test types to be performed.

Test automation: The use of software to perform or support test activities, e.g., test management, test design, test execution, and results checking.

Test basis: All documents from which the requirements of a component or system can be inferred. The documentation on which the test cases are based. If a document can be amended only by way of formal amendment procedure, then the test basis is called a frozen test basis. [After TMap]

Test bed: An environment containing the hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test. See also test environment.

Test case: A test that, ideally, executes a single well-defined test objective, i.e., a specific behavior of a feature under a specific condition. A set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement. [After IEEE 610]

Test case design technique: See test design technique.

Test case specification: A document specifying a set of test cases (objective, inputs, test actions, expected results, and execution preconditions) for a test item. [After IEEE 829]

Test case suite: See test suite.

Test charter: A statement of test objectives, and possibly test ideas. Test charters are amongst others used in exploratory testing. See also exploratory testing.

Test comparator: A test tool to perform automated test comparison.

Test comparison: The process of identifying differences between the actual results produced by the component or system under test and the expected results for a test. Test comparison can be performed during test execution (dynamic comparison) or after test execution.

Test completion criterion: See exit criteria.

Test condition: An item or event of a component or system that could be verified by one or more test cases, e.g., a function, transaction, quality attribute, or structural element.

Test coverage: See coverage.

Test data: Data that exists (for example, in a database) before a test is executed, and that affects or is affected by the component or system under test.

Test data preparation tool: A type of test tool that enables data to be selected from existing databases or created, generated, manipulated, and edited for use in testing.

Test deliverables: List of test materials developed by the test group during the test cycles that are to be delivered before the completion of the project.

Test design: See test design specification.

Test design specification: A document specifying the test conditions (coverage items) for a test item, the detailed test approach, and identifying the associated high-level test cases. [After IEEE 829]

Test design technique: A method used to derive or select test cases.

Test design tool: A tool that supports the test design activity by generating test inputs from a specification that may be held in a CASE tool repository, e.g., requirements management tool, or from specified test conditions held in the tool itself.

Test driver: It automates the execution of a test case. See also driver.

Test environment: An environment containing hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test. [After IEEE 610]

Test evaluation report: A document produced at the end of the test process summarizing all testing activities and results. It also contains an evaluation of the test process and lessons learned.

Test execution: The process of running a test by the component or system under test, producing actual result(s).

Test execution automation: The use of software, e.g., capture/playback tools, to control the execution of tests, the comparison of actual results to expected results, the setting up of test preconditions, and other test control and reporting functions.

Test execution phase: The period of time in a software development life cycle during which the components of a software product are executed, and the software product is evaluated to determine whether or not requirements have been satisfied. [IEEE 610]

Test execution schedule: A scheme for the execution of test procedures. The test procedures are included in the test execution schedule in their context and in the order in which they are to be executed.

Test execution technique: The method used to perform the actual test execution, either manually or automated.

Test execution tool: A type of test tool that is able to execute other software using an automated test script, e.g., capture/playback. [Fewster and Graham]

Test fail: See fail.

Test generator: See test data preparation tool.

Test harness: A tool that performs automated testing of the core components of a program or system. It is the driver of test drivers. Tests under a central control. A test environment comprised of stubs and drivers needed to conduct a test.

Test incident: See incident.

Test incident report: See incident report.

Test infrastructure: The organizational artifacts needed to perform testing, consisting of test environments, test tools, office environment, and procedures.

Test item: The individual element to be tested. There usually is one test object and many test items. See also test object.

Test item transmittal report: See release note.

Test level: A group of test activities that are organized and managed together. A test level is linked to the responsibilities in a project. Examples of test levels are component test, integration test, system test, and acceptance test. [After TMap]

Test log: A chronological record of relevant details about the execution of tests. [IEEE 829]

Test logging: The process of recording information about tests executed into a test log.

Test management: The planning, estimating, monitoring, and control of test activities, typically carried out by a test manager.

Test manager: The person responsible for testing and evaluating a test object. The individual, who directs, controls, administers plans, and regulates the evaluation of a test object.

Test maturity model (TMM): A five level staged framework for test process improvement, related to the capability maturity model (CMM) that describes the key elements of an effective test process.

Test object: The component or system to be tested. See also test item.

Test objective: A reason or purpose for designing and executing a test.

Test oracle: It is a mechanism, different from the program itself that can be used to check the correctness of the output of the program for the test cases. It is a process in which test cases are given to test oracles and the program under testing. The output of the two is then compared to determine if the program behaved correctly for the test cases. A source to determine expected results to compare with the actual result of the software under test. An oracle may be the existing system (for a benchmark), a user manual, or an individual's specialized knowledge, but should not be the code. [After Adrion]

Test outcome: See result.

Test pass: See pass.

Test performance indicator: A metric, in general high level, indicating to what extent a certain target value or criterion is met. Often related to test process improvement objectives, e.g., defect detection percentage (DDP).

Test phase: A distinct set of test activities collected into a manageable phase of a project, e.g., the execution activities of a test level. [After Gerrard]

Test plan: A management document outlining risks, priorities, and schedules for testing. A document describing the scope, approach, resources, and schedule of intended test activities. It identifies amongst others test items, the features to be tested, the testing tasks, who will do each task, degree of tester independence, the test environment, the test design techniques and test measurement techniques to be used, and the rationale for their choice, and any risks requiring contingency planning. It is a record of the test planning process. [After IEEE 829]

Test planning: The activity of establishing or updating a test plan.

Test point analysis (TPA): A formula-based test estimation method based on function point analysis. [TMap]

Test points: They allow data to be modified or inspected at various points in the system.

Test policy: A high-level document describing the principles, approach, and major objectives of the organization regarding testing.

Test procedure: See test procedure specification.

Test procedure specification: A document specifying a sequence of actions for the execution of a test. Also known as test script or manual test script. [After IEEE 829]

Test process: The fundamental test process comprises planning, specification, execution, recording, and checking for completion. [BS 7925/2]

Test process improvement (TPI): A continuous framework for test process improvement that describes the key elements of an effective test process, especially targeted at system testing and acceptance testing.

Test record: See test log.

Test recording: See test logging.

Test repeatability: An attribute of a test indicating whether the same results are produced each time the test is executed.

Test report: See test summary report.

Test requirement: A document that describes items and features that are tested under a required condition.

Test result: See result.

Test run: Execution of a test on a specific version of the test object.

Test run log: See test log.

Test script: Step-by-step instructions that describe how a test case is to be executed. A test script may contain one or more test cases. Commonly used to refer to a test procedure specification, especially an automated one.

Test session: One set of tests for a specific configuration actual code and stubs.

Test situation: See test condition.

Test specification: A set of test cases, input, and conditions that are used in the testing of a particular feature or set of features. A test specification often includes descriptions of expected results. A document that consists of a test design specification, test case specification, and/or test procedure specification.

Test stage: See test level.

Test strategy: A high-level document defining the test levels to be performed and the testing within those levels for a program (one or more projects).

Test stub: Dummy function/component to simulate a real component.

Test suite: A collection of test scripts or test cases that is used for validating bug fixes or finding new bugs within a logical or physical area of a product. For example, an acceptance test suite contains all of the test cases that are used to verify that the software has met certain predefined acceptance criteria. On the other hand, a regression suite contains all of the test cases that are used to verify that all previously fixed bugs are still fixed. A set of several test cases for a component or system under test, where the post condition of one test is often used as the precondition for the next one.

Test summary report: A document summarizing testing activities and results. It also contains an evaluation of the corresponding test items against exit criteria. [After IEEE 829]

Test target: A set of exit criteria.

Test tool: A software product that supports one or more test activities, such as planning and control, specification, building initial files and data, test execution, and test analysis. [TMap] See also CAST.

Test type: A group of test activities aimed at testing a component or system regarding one or more interrelated quality attributes. A test type is focused on a specific test objective, i.e., reliability test, usability test, regression test, etc., and may take place on one or more test levels or test phases. [After TMap]

Testable requirements: The degree to which a requirement is stated in terms that permit establishment of test designs (and subsequently test cases) and execution of tests to determine whether the requirements have been met. [After IEEE 610]

Testability: The capability of the software product to enable modified software to be tested. [ISO 9126] See also maintainability.

Testability looks: The code that is inserted into the program specifically to facilitate testing.

Testability review: A detailed check of the test basis to determine whether the test basis is at an adequate quality level to act as an input document for the test process. [After TMap]

Tester: A technically skilled professional who is involved in the testing of a component or system.

Testing: The process of executing the program with the intent of finding faults. The process consisting of all life cycle activities, both static and dynamic, concerned with planning, preparation, and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose, and to detect defects.

Testing interface: A set of public properties and methods that you can use to control a component from an external testing program.

Testware: Artifacts produced during the test process required to plan, design, and execute tests, such as documentation, scripts, inputs, expected results, set-up and clear-up procedures, files, databases, environment, and any additional software or utilities used in testing. [After Fewster and Graham]

Thread testing: A version of component integration testing where the progressive integration of components follows the implementation of subsets of the requirements, as opposed to the integration of components by levels of a hierarchy.

Time behavior: See performance.

Top-down testing: An incremental approach to integration testing where the component at the top of the component hierarchy is tested first, with lower level components being simulated by stubs. Tested components are then used to test lower level components. The process is repeated until the lowest level components have been tested.

Traceability: The ability to identify related items in documentation and software, such as requirements with associated tests. See also horizontal traceability and vertical traceability.

Transaction: A unit of work seen from a system user's point of view.

UML (unified modified language): A standard language used for developing software blueprints using nine different diagrams.

Understandability: The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use. [ISO 9126] See also usability.

Unit testing: See component testing.

Unreachable code: Code that cannot be reached and, therefore, is impossible to execute.

Usability: The capability of the software to be understood, learned, used, and attractive to the user when used under specified conditions. [ISO 9126]

Usability testing: The testing that validates the ease of use, speed, and aesthetics of the product from the user's point of view. Testing to determine the extent to which the software product is understood, easy to learn, easy to operate, and attractive to the users under specified conditions. [After ISO 9126]

Use-case testing: A black-box test design technique in which test cases are designed to execute user scenarios.

User acceptance testing: See acceptance testing.

User scenario testing: See use-case testing.

User test: A test whereby real-life users are involved to evaluate the usability of a component or system.

V-model: A framework to describe the software development life cycle activities from requirements specification to maintenance. The V-model illustrates how testing activities can be integrated into each phase of the software development life cycle.

Validation: It is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies the specified requirements. It involves executing the actual software. It is "computer based testing" process. Confirmation by examination and through provision of objective evidence that the requirements for a specific intended use or application have been fulfilled. [ISO 9000]

Variable: An element of storage in a computer that is accessible by a software program by referring to it by a name.

Verification: It is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. It is a "human testing" activity. Confirmation by examination and through the provision of objective evidence that specified requirements have been fulfilled. [ISO 9000]

Vertical traceability: The tracing of requirements through the layers of development documentation to components.

Volume testing: Testing where the system is subjected to large volumes of data. See also resource-utilization testing.

Walkthrough: A step-by-step presentation by the author of a document in order to gather information and to establish a common understanding of its content. [Freedman and Weinberg, IEEE 1028]

White-box test design technique: Documented procedure to derive and select test cases based on an analysis of the internal structure of a component or system.

White-box testing: Testing based on an analysis of the internal structure of the component or system.

Wide band delphi: An expert-based test estimation technique that aims at making an accurate estimation using the collective wisdom of the team members.

Widget: A synonym for user interface component and control.

Wizard: A user assistance device that guides users step by step through a procedure.

Work breakdown structure: A structure diagram that shows the project in terms of phases.

APPENDIX D

BIBLIOGRAPHY

My special thanks are—for the great researchers without whose help this book would not have been possible:

1. Jorgensen Paul, "Software Testing—A Practical Approach", CRC Press, 2nd Edition 2007.
2. Srinivasan Desikan and Gopalaswamy Ramesh, "Software testing—Principles and Practices", Pearson Education Asia, 2002.
3. Tamres Louise, "Introduction to Software Testing", Pearson Education Asia, 2002.
4. Mustafa K., Khan R.A., "Software Testing—Concepts and Practices", Narosa Publishing, 2007.
5. Puranik Rajnikant, "The Art of Creative Destination", Shroff Publishers, First Reprint, 2005.
6. Agarwal K.K., Singh Yogesh, "Software Engineering", New Age Publishers, 2nd Edition, 2007.
7. Khurana Rohit, "Software Engineering—Principles and Practices", Vikas Publishing House, 1998.
8. Agarwal Vineet, Gupta Prabhakar, "Software Engineering", Pragati Prakashan, Meerut.
9. Sabharwal Sangeeta, "Software Engineering—Principles, Tools and Techniques", New Age Publishers, 1st Edition, 2002.
10. Mathew Sajan, "Software Engineering", S. Chand and Company Ltd., 2000.
11. Kaner, "Lessons Learned in Software Testing", Wiley, 1999.
12. Rajani Renu, Oak Pradeep, "Software Testing", Tata McGraw Hill, First Edition, 2004.
13. Nguyen Hung Q., "Testing Applications on Web", John Wiley, 2001.
14. "Testing Object-Oriented Systems—A Workshop Workbook", by Quality Assurance Institute (India) Ltd., 1994-95.

15. Pressman Roger, "A Manager's Guide to Software Engineering", Tata McGraw Hill, 2004.
16. Perry William E., "Effective Methods for Software Testing", Wiley Second Edition, 1995.
17. Cardoso Antonio, "The Test Estimation Process", June 2000, Stickyminds.com.
18. Erik P.W.M., Veenendall Van, Dekkers Ton, "Test Point Analysis; A method for Test Estimation".
19. Mathur P. Aditya, "Foundations of Software Testing", Pearson Education Asia, First Impression, 2008.
20. Black Rex, "Test Estimation", 2002, Stque Magazine.com.
21. Chopra Rajiv, "Modern Software Engineering", First Edition, I.K International Publishers, 2014.
22. Nangia Rajan, "Software Testing", Cyber Tech. Publications, First Edition, 2008.
23. Stout A. Glenn, "Testing a Website: Best Practices", "http://www.stickyminds.com/sitewide.asp/XUS893545file/.pdf
24. Kota Krishen, "Testing Your Web Application—A Quick 10–Step Guide", Stickyminds.com.
25. Intosh Mac, Strigel W., "The Living Creature"—Testing Web Applications, 2000, http://www.stickyminds.com/docs_index/XUS11472file/.pdf.
26. Binder Robert V., "Testing Object Oriented Systems—Models, Patterns and Tools", Addison Wesley, 1999.
27. Websites: www.stickyminds.com, www.testingcenter.com.
28. Chopra Rajiv, "JAVA Programming", New Age International Publishers, First Edition, 2015.

INDEX

A

Acceptance testing, 260, 531
  for critical applications, 264
  guidelines to select test cases for, 261
  types of, 263
Automated testing, 371
  benefits of, 376
  consideration during, 372
  disadvantages of, 377
Automated tools, 376, 377

B

Basic test plan template, 583
Basis path coverage, 155
Basis path testing, 161
Beta testing, 247
Bidirectional integration, 231
Big-bang integration, 232
Black box testing, 77, 114, 200, 221, 495, 560
Boolean coverage, 156
Bottom-up integration, 231
Boundary value analysis, 78
  guidelines for, 86
  limitations of, 79
  robustness testing, 79
Branch coverage, 155
Bugs, 11
Bug writing, 573
Build acceptance test, 263
Business plan, 296
Business vertical testing, 244

C

Capability maturity model, 319, 320
  applications, 323
  overview of, 320
Capability maturity model integration, 319, 323
  need of, 325
Capacity planning, 280
Cause-effect functional testing
  guidelines for, 113
Cause-effect graphing, 109
Chasing false defects, 569

Classification of integration testing
call graph based, 233
decomposition-based, 230
Class partitioning, 86
Class testing, 86
heuristics for, 506
strong normal, 88
strong robust, 89
weak normal, 87
weak robust, 88
Clear box testing, 145
Code complexity testing, 153
Code coverage testing, 148
Code functional testing, 147
Commission problem
test cases for, 84, 96, 108
Condition coverage, 151
Control flow testing, 484
Conventional and object oriented testing, 540
Coupling between objects, 157
Currency converter application, 544
Cyclomatic complexity, 153, 154, 159
properties of, 160
Cyclomatic density, 155

D

Data flow testing, 183, 492
DD path testing, 177
Debug, 2
Debugging, 2, 379, 382
approaches, 382
process, 380
tools for, 383
Decision table, 98
advantages of, 99
applications of, 99
disadvantages of, 99
guidelines for, 108
Defect, 11
Deployment acceptance test, 264
Deployment testing, 245
Design for testability, 537
Development acceptance test, 263
Dynamic testing, 2, 374

E

Equivalence class partitioning, 86
Equivalence class testing, 86
guidelines for, 97
strong normal, 88
strong robust, 89
weak robust, 88
week normal, 87
Error, 11
Error seeding, 198
Essential complexity, 155

F

Failure, 11
False defects, 569
Fault, 11
Financial independence, 48
Flow graph testing, 161
Functional acceptance simple test, 263
Functional specification-based testing, 495
Functional testing, 77, 114, 240, 418, 522
guidelines for, 117
techniques, 242
Function coverage, 152
future of, 341

G

Glass box, 145
Good test, 9
Graph matrices technique, 181
Gray box testing, 219, 220, 221
GUI testing, 540

I

IEEE829 standard, 50
IEEE STD. 1012, 33
Implementation-based class testing, 483
  limitation of, 494
Incident, 12
Incremental testing, 10
Independent V&V contractor, 47
Installation testing, 419
Integration complexity, 155
Integration testing, 229, 243, 418, 517
  classification of, 230
    call graph based, 233
    decomposition-based, 230
    path-based, 235
Interoperability testing, 258
Intra class coverage, 493
ISO 9001 and CMM, 322
ISO standards, 317

K

Kiviat chart, 117

L

Lack of cohesion between methods, 158
Life cycle of a build, 563
Load testing, 418

M

Managerial independence, 48
Manual testing, 375
Metrics, 154
Mistake, 11
Modern testing tools, 387
Mothora, 20
Mutation testing, 198
  advantages of, 199
  disadvantages of, 199

N

National quality awards, 343
Neighborhood integration, 234
Next date function, 83
  complexities in, 83
  test cases for, 83, 91, 103
Non-functional testing, 240
  techniques, 250
Normal testing, 356
No silver bullet, 379

O

Object oriented testing, 451, 540
  levels of, 513
Open box testing, 145

P

Pairwise integration, 233
Path coverage, 150
Path coverage testing, 491
Pathological complexity, 155
Payroll problem, 112
Performance benchmarking, 279
Performance testing, 265
  challenges of, 272, 282
  factors, 262
  steps of, 271
  tools for, 282
Performance tuning, 278
Planning process overview, 295
Prioritization of test cases, 351
  guidelines, 351
  for regression testing, 360
Priority category scheme, 352
Proof of correctness, 37
Prototyping, 38

Q

QA analyst, 68

Quality
  core aspects of, 72
  core components of, 71
Quality assurance, 31, 67, 293
Quality challenge, 342
Quality control, 31
Quality factor, 69
Quality guidelines, 309
Quality management, 69, 70
Quality management tools, 333
Quality models, 309, 310
Quality plan, 296
Quality planning, 294
Quality plan objectives, 295
Quality standard (ISO 9001), 322
Quality standards, 309

R

Regression testing, 356, 419, 531, 571
  prioritization of test cases for, 360
  technique, 361
  types of, 357
Release acceptance test, 263
Reliability testing, 254
Response for class, 157
Responsibility-based class testing, 495
Return filed report, 420
Risk analysis, 353
Robustness testing, 79

S

Sandwich integration, 231
Scalability testing, 252
  guidelines for, 254
Simulation, 38
Single fault assumption theory, 78
Six sigma, 325
  application of, 340
  benefits of, 338
  criticism of, 339
  future of, 341
  levels, 336
  methodologies, 330
  origin and meaning of the term, 335
Slice based testing, 362
Smoke test, 571
Software control, 66
Software quality, 65
  role of process in, 66
Software requirement specifications, 399
Software technical reviews, 43
  methodologies, 46
  rationale for, 43
  types of, 45
Software test documentation, 50
Software testing, 1, 2, 382
  terminology, 11
Software V&V planning, 39
State machines, 460
Statement coverage, 148
Static testing, 2, 374
Stress testing, 255, 418
Structural testing, 145, 483
System design, 416
System testing, 238, 243, 521

T

Team effort, 9
Technical independence, 47
Test, 12
  log, 17
  oracle, 17
  report, 17
  script, 16
  suite, 16
  ware, 17
Test automation, 379
Test cases, 12, 420
Tester performance, 598
Test execution issues, 544

Testing
  approach, 10
  and debugging, 379
  developers for, 9
  effectiveness, 116, 595
  efficiency, 116
  effort, 114
  e-learning management systems, 397
  good test cases, 9
  levels of, 227, 418
  life cycle, 17
  limitations of, 19
  measurement of, 10
  metrics, 20
  number of, 9
  object oriented systems, 483
  principles of, 18
  process, 2
  purpose, 6
  techniques, 20
  time to stop, 18
  tools, 20, 378
  types of, 417
  using orthogonal arrays, 542
Testing "around" a bug, 572
Testing for standards, 249
Testing progress, 591
Testing tools, 378
Test process, 533
Test report, 417
Test tools, 384
Tool selection, 386
Total quality management, 297
  concepts, 300
  principles of, 298
Triangle problem
  test cases for, 81, 90, 102, 110

U

Unit testing, 147, 228, 418, 514

V

Verification and validation, 29
  differences between, 30
  independent contractor, 41, 47
  limitations, 32
  planning, 39
  positive and negative effects of, 48
  proof of correctness, 37
  in SDLC, 33
  simulations and prototypes, 38
  techniques, 33
Version control, 567

W

Weighted methods for class, 157
White box testing, 145, 200, 202, 221, 483, 562
  advantages of, 203
  dynamic, 146, 147
  limitation of, 494
  static, 146
Worst-case testing, 80
Writing bugs, 573

Z

Zero defect movement, 302