Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

SOFTWARE TESTING

Principles and Practices

Naresh Chauhan

OXFORD
UNIVERSITY PRESS

© Oxford University Press 2010

ISBN: 978-0-1980618-47

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

To
my parents
who have made me capable
to struggle in this world

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Preface

There is no life without struggles and no software without bugs. Just as one needs to sort out the problems in one's life, it is equally important to check and weed out the bugs in software. Bugs cripple the software in a way problems in life unsettle one. In our life, both joys and sorrows are fleeting. But a person is best tested in times of crises. One who cultivates an optimistic outlook by displaying an equipoise taking prosperity as well as adversity in his stride and steadily ventures forth on a constructive course is called a sthir pragna. We should follow the same philosophy while testing software too. We need to develop an understanding that unless these bugs appear in our software and until we weed out all of them, our software will not be robust and of superior quality. So, a software test engineer should be an optimist who welcomes the struggles in life and similarly bugs in software, and takes them head-on.

Software engineering as a discipline emerged in the late 1960s to guide software development activities in producing quality software. Quality here is not a single-dimensional entity. It has several factors including rigorous software testing. In fact, testing is the critical element of quality and consumes almost half the total development effort. However, it is unfortunate that the quality and testing process does not get its due credit. In software engineering, testing is considered to be a single phase operation performed only after the development of code wherein bugs or errors are removed. However, this is not the case. Testing is not just an intuitive method to remove the bugs, rather it is a systematic process such as software development life cycle (SDLC). The testing process starts as soon as the first phase of SDLC starts. Therefore, even after learning many things about software engineering, there are still some questions and misconceptions regarding the testing process which need to be known, such as the following:

- When should testing begin?
- How much testing is practically possible?
- What are the various techniques to design a good test case (as our knowledge is only limited to black-box and white-box techniques)?

Moreover, the role of software testing as a systematic process to produce quality software is not recognized on a full scale. Many well-proven methods are largely unused in industries today. Companies rely only on the automated testing tools rather than a proper testing methodology. What they need to realize is that Computer-Aided Software Engineering (CASE) environments or tools are there only to assist in the development effort and not meant to serve as silver bullets! Similarly, there are many myths that both students and professionals believe in, which need to be exploded. The present scenario requires *software testing* to be acknowledged as a separate discipline from software engineering. Some universities have already started this course. Therefore, there is a need for a book that explains all these issues for the benefit of students who will learn software testing and become knowledgeable test engineers as also for the benefit of test engineers who are already working in the industries and want to hone their testing skills.

About the book

This book treats software testing as a separate discipline to teach the importance of testing process both in academia as well as in the industry. The book stresses on software testing as a systematic process and explains software testing life cycle similar to SDLC and gives insight into the practical importance of software testing. It also describes all the methods/techniques for test case design which is a prime issue in software testing. Moreover, the book advocates the notion of effective software testing in place of exhaustive testing (which is impossible).

The book has been written in a lucid manner and is packed with practical approach of designing the test cases targeting undergraduate and postgraduate students of computer science and engineering (B.Tech., M.Tech., MCA), and test engineers. It discusses all the software testing issues and gives insight into their practical importance. Each chapter starts with the learning objectives and ends with a summary containing a quick review of important concepts discussed in the chapter. Some chapters provide solved examples in between the theory to understand the method or technique practically at the same moment. End-chapter exercises and multiple-choice questions are provided to assist instructors in classroom teaching and students in preparing better for their exams.

The key feature of the book is a fully devoted case study on Income Tax Calculator which shows how to perform verification and validation at various phases of SDLC. The case study includes ready-to-use software and designing of test cases using the techniques described in the book. This material will help both students and testers understand the test design techniques and use them practically.

Apart from the above-mentioned features, the book follows the following methodology in defining key concepts in software testing:

- Emphasis on software testing as a systematic process
- Effective testing concepts rather than exhaustive complete testing
- A testing strategy with a complete roadmap has been developed that shows which software testing technique with how much risk assessment should be adopted at which phase of SDLC
- Testing models
- Verification and validation as the major components of software testing process. These have been discussed widely expanding in separate chapters.
- Software testing life cycle along with bug classification and bug life cycle
- Complete categorization of software testing techniques such as static testing and dynamic testing expanding in different chapters
- Testing techniques with solved examples to illustrate how to design test cases using these techniques
- Extensive coverage of regression testing, software testing metrics, and test management
- Efficient test suite management to prioritize test cases suitable for a project
- The appropriate use of testing tools
- Software quality management and test maturity model (TMM)
- Testing techniques for two specialized environments: object-oriented software and Web-based software

About the CD

The CD accompanying the book contains the following:

- *Executable files* for the examples given in Chapter 5 so that a user can directly implement white-box testing on the codes without any extra effort.
- *Checklists* for verification of parameters, such as general software design document (SDD), generic code, high level design (HLD), low level design (LLD), and software requirement specification (SRS) document.
- *A program on Income Tax Calculator* along with its description in the form of a case study that illustrates all the steps of the software testing process.

Content and Coverage

The book has been divided into seven different parts. Each part further consists of various chapters.

Part I (Testing Methodology) introduces concepts such as effective software testing, testing terminology, testing as a process, and development of testing methodology.

Chapter 1 introduces the concept of effective testing versus complete testing, explains the psychology for performing effective testing, and establishes that software testing is a complete process.

Chapter 2 discusses the commonly used testing terminology such as error, bug, and failure, explains life cycle of a bug with its various states, phases of software testing life cycle and V testing model, and development of a testing methodology.

Chapter 3 explains how verification and validation, a part of testing strategy, are performed at various phases of SDLC.

Part II (Testing Techniques) deals with various test case design techniques based on static testing and dynamic testing and verification and validation concepts.

Chapter 4 covers test case design techniques using black-box testing including boundary value analysis, equivalence class partitioning method, state table based testing, decision table based testing, and cause-effect graphing technique.

Chapter 5 discusses test case design techniques using white-box testing, including basis path testing, loop testing, data flow testing, and mutation testing.

Chapter 6 deals with the techniques, namely inspection, walkthrough, and reviews, largely used for verification of various intermediate work products resulting at different stages of SDLC.

Chapter 7 discusses various techniques used in validation testing such as unit testing, integration testing, function testing, system testing, and acceptance testing.

Chapter 8 describes regression testing that is used to check the effect of modifications on other parts of software.

Part III (Managing the Testing Process) discusses how to manage the testing process, various persons involved in test organization hierarchy, testing metrics to monitor and control the testing process, and how to reduce the number of test cases.

Chapter 9 covers the concept of introduction of management for test process for its effectiveness. Various persons involved in test management hierarchy are discussed. The test planning

for various verification and validation activities are also discussed along with the test result specifications.

Chapter 10 provides an introductory material to understand that measurement is a necessary part of software engineering, known as software metrics.

Chapter 11 explains how software metrics assist in monitoring and controlling different testing activities.

Chapter 12 explains the fact that test cases, specially designed for system testing and regression testing, become unmanageable in a way that we cannot test all of them. The problem is how to select or reduce the test cases out of a big test suite. This chapter discusses many such techniques to resolve the problem.

Part IV (Quality Management) covers software quality issues with some standards along with testing process maturity models.

Chapter 13 discusses various terminologies, issues, and standards related to software quality management to produce high quality software.

Chapter 14 discusses various test process maturity models, namely test improvement model (TIM), test organization model (TOM), test process improvement (TPI), and test maturity model (TMM).

Part V (Test Automation) discusses the need of testing and provides an introduction to testing tools.

Chapter 15 explains the need for automation, categories of testing tools, and how to select a testing tool.

Part VI (Testing for Specialized Environment) introduces the testing environment and the issues related to two specialized environments, namely object-oriented software and Web-based software.

Chapters 16 and 17 discuss the issues, challenges, and techniques related to object-oriented and Web-based software, respectively.

Part VII (Tracking the Bug) explains the process and techniques of debugging.

Chapter 18 covers the debugging process and discusses various methods to debug a software product.

The book concludes with a case study of Income Tax Calculator illustrating the testing of software using verification and validation techniques. In addition, the book contains useful appendices that provide various ready-to-use checklists which can be used at the time of verification of an item at various stages of SDLC.

Do send your valuable suggestions, comments, and constructive criticism for further improvement of the book.

Naresh Chauhan

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Contents

Preface *v*

PART 1: Testing Methodology

1. Introduction to Software Testing **3**

1.1 Introduction *3*
1.2 Evolution of Software Testing *5*
1.3 Software Testing–Myths and Facts *8*
1.4 Goals of Software Testing *10*
1.5 Psychology for Software Testing *13*
1.6 Software Testing Definitions *14*
1.7 Model for Software Testing *15*
1.8 Effective Software Testing vs. Exhaustive Software Testing *16*
1.9 Effective Testing is Hard *21*
1.10 Software Testing as a Process *22*
1.11 Schools of Software Testing *23*
1.12 Software Failure Case Studies *25*

2. Software Testing Terminology and Methdology **32**

2.1 Software Testing Terminology *33*
2.2 Software Testing Life Cycle (STLC) *46*
2.3 Software Testing Methodology *51*

3. Verification and Validation **65**

3.1 Verification and Validation (V&V) Activities *66*
3.2 Verification *69*
3.3 Verification of Requirements *70*
3.4 Verification of High-level Design *74*
3.5 Verification of Low-level Design *76*
3.6 How to Verify Code? *77*
3.7 Validation *79*

PART 2: Testing Techniques

4. Dynamic Testing: Black-Box Testing Techniques **89**

4.1 Boundary Value Analysis (BVA) *90*
4.2 Equivalence Class Testing *107*
4.3 State Table-Based Testing *114*
4.4 Decision Table-Based Testing *119*
4.5 Cause-Effect Graphing Based Testing *125*
4.6 Error Guessing *129*

5. Dynamic Testing: White-Box Testing Techniques **135**

5.1 Need of White-Box Testing *135*
5.2 Logic Coverage Criteria *136*
5.3 Basis Path Testing *138*
5.4 Graph Matrices *156*
5.5 Loop Testing *161*
5.6 Data Flow Testing *164*
5.7 Mutation Testing *174*

6. Static Testing **188**

6.1 Inspections *190*
6.2 Structured Walkthroughs *205*
6.3 Technical Reviews *206*

7. Validation Activites **212**

7.1 Unit Validation Testing *213*
7.2 Integration Testing *218*
7.3 Function Testing *231*
7.4 System Testing *233*
7.5 Acceptance Testing *244*

8. Regression Testing 255

8.1 Progressive vs. Regressive Testing *255*
8.2 Regression Testing Produces Quality Software *256*
8.3 Regression Testability *257*
8.4 Objectives of Regression Testing *258*
8.5 When is Regression Testing Done? *258*
8.6 Regression Testing Types *259*
8.7 Defining Regression Test Problem *259*
8.8 Regression Testing Techniques *260*

PART 3: Managing the Testing Process

9. Test Management 273

9.1 Test Organization *274*
9.2 Structure of Testing Group *275*
9.3 Test Planning *276*
9.4 Detailed Test Design and Test Specifications *292*

10. Software Metrics 304

10.1 Need of Software Measurement *305*
10.2 Definition of Software Metrics *306*
10.3 Classification of Software Metrics *306*
10.4 Entities to be Measured *307*
10.5 Size Metrics *308*

11. Testing Metrics for Monitoring and Controlling the Testing Process 317

11.1 Measurement Objectives for Testing *318*
11.2 Attributes and Corresponding Metrics in Software Testing *319*
11.3 Attributes *320*
11.4 Estimation Models for Estimating Testing Efforts *327*
11.5 Architectural Design Metric Used for Testing *331*
11.6 Information Flow Metrics Used for Testing *332*
11.7 Cyclomatic Complexity Measures for Testing *333*
11.8 Function Point Metrics for Testing *333*
11.9 Test Point Analysis (TPA) *335*
11.10 Some Testing Metrics *341*

12. Efficient Test Suit Management 352

12.1 Why Does a Test Suite Grow? *352*
12.2 Minimizing the Test Suite and its Benefits *353*
12.3 Defining Test Suite Minimization Problem *354*
12.4 Test Suite Prioritization *354*
12.5 Types of Test Case Prioritization *355*
12.6 Prioritization Techniques *356*
12.7 Measuring the Effectiveness of a Prioritized Test Suite *365*

PART 4: Quality Management

13. Software Quality Management 373

13.1 Software Quality *374*
13.2 Broadening the Concept of Quality *374*
13.3 Quality Cost *375*
13.4 Benefits of Investment on Quality *376*
13.5 Quality Control and Quality Assurance *377*
13.6 Quality Management (QM) *378*
13.7 QM and Project Management *379*
13.8 Quality Factors *379*
13.9 Methods of Quality Management *380*
13.10 Software Quality Metrics *387*
13.11 SQA Models *390*

14. Testing Process Maturity Models 404

14.1 Need for Test Process Maturity *405*
14.2 Measurement and Improvement of a Test Process *406*
14.3 Test Process Maturity Models *406*

PART 5: Test Automation

15. Automation and Testing Tools 429

15.1 Need for Automation *430*
15.2 Categorization of Testing Tools *431*
15.3 Selection of Testing Tools *434*

15.4 Costs Incurred in Testing Tools *435*
15.5 Guidelines for Automated Testing *436*
15.6 Overview of Some Commercial Testing Tools *437*

PART 6: Testing for Specialized Environment

16. Testing Object-Oriented Software 445
16.1 OOT Basics *446*
16.2 Object-oriented Testing *450*

17. Testing Web-based Systems 474
17.1 Web-based System *474*
17.2 Web Technology Evolution *475*
17.3 Traditional Software and Web-based Software *476*
17.4 Challenges in Testing for Web-based Software *477*
17.5 Quality Aspects *478*
17.6 Web Engineering (Webe) *480*
17.7 Testing of Web-based Systems *484*

PART 7: Tracking the Bug

18. Debugging 503
18.1 Debugging: an Art or Technique? *503*
18.2 Debugging Process *504*
18.3 Debugging Is Difficult *505*
18.4 Debugging Techniques *506*
18.5 Correcting the Bugs *509*
18.6 Debuggers *510*

Income Tax Calculator: A Case Study

Step 1 *Introduction to Case Study* *513*
Step 2 *Income Tax Calculator SRS ver 1.0* *515*
Step 3 *Verification on Income Tax Calculator SRS ver 1.0* *517*
Step 4 *Income Tax Calculator SRS ver 2.0* *520*
Step 5 *Verification on Income Tax Calculator SRS ver 2.0* *525*
Step 6 *Income Tax Calculator SRS ver 3.0* *531*
Step 7 *Black-Box Testing on Units/Modules of Income Tax Calculator SRS ver 3.0* *538*
Step 8 *White-Box Testing on Units/Modules of Income Tax Calculator* *552*

Appendices

Appendix A *Answers to Multiple Choice Questions* *587*
Appendix B *Software Requirement Specification (SRS) Verification Checklist* *589*
Appendix C *High Level Design (HLD) Verification Checklist* *592*
Appendix D *Low Level Design (LLD) Verification Checklist* *594*
Appendix E *General Software Design Document (SDD) Verification Checklist* *595*
Appendix F *Generic Code Verification Checklist* *596*

References *600*
Index *606*

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Part 1 Testing Methodology

CHAPTERS

Chapter 1:
Introduction to Software Testing

Chapter 2:
Software Testing Terminology and Methodology

Chapter 3:
Verification and Validation

Software testing has always been considered a single phase performed after coding. But time has proved that our failures in software projects are mainly due to the fact that we have not realized the role of software testing as a process. Thus, its role is not limited to only a single phase in the software development life cycle (SDLC), but it starts as soon as the requirements in a project have been gathered.

Complete software testing has also been perceived for a long time. Again, it has been proved that exhaustive testing is not possible and we should shift our attention to effective testing. Thus, effective and early testing concepts build our testing methodology. Testing methodology shows the path for successful testing. This is the reason that parallel to SDLC, software testing life cycle (STLC) has also been established now.

The testing methodology is related to many issues. All these issues have been addressed in this part. The goals of software testing, the mindset required to perform testing, clear-cut definitions of testing terminology, phases of STLC, development of testing methodology, verification and validation, etc. have been discussed in this part.

This part will make ground for the following concepts:

- Effective testing, not exhaustive testing.
- Software testing is an established process.
- Testing should be done with the intention of finding more and more bugs, not hiding them.
- Difference between error, fault, and failure.
- Bug classification.
- Development of software testing methodology.
- Testing life cycle models.
- Difference between verification and validation.
- How to perform verification and validation at various stages of SDLC.

Chapter

1

Introduction to Software Testing

1.1 Introduction

OBJECTIVES

After reading this chapter, you should be able to understand:

- How software testing has evolved over the years
- Myths and facts of software testing
- Software testing is a separate discipline
- Testing is a complete process
- Goals of software testing
- Testing is based on a negative/destructive view
- Model for testing process
- Complete testing is not possible
- Various schools of software testing

Software has pervaded our society, from modern households to spacecrafts. It has become an essential component of any electronic device or system. This is why software development has turned out to be an exciting career for computer engineers in the last 10–15 years. However, software development faces many challenges. Software is becoming complex, but the demand for quality in software products has increased. This rise in customer awareness for quality increases the workload and responsibility of the software development team. That is why software testing has gained so much popularity in the last decade. Job trends have shifted from development to software testing. Today, software quality assurance and software testing courses are offered by many institutions. Organizations have separate testing groups with proper hierarchy. Software development is driven with testing outputs. If the testing team claims the presence of bugs in the software, then the development team cannot release the product.

However, there still is a gap between academia and the demand of industries. The practical demand is that passing graduates must be aware of testing terminologies, standards, and techniques. But students are not aware in most cases, as our universities and colleges do not offer separate software quality and testing courses. They study only software engineering. It can be said that software engineering is a mature discipline today in industry as well as in academia. On the other hand, software testing is mature in industry but not in academia. Thus, this gap must be bridged with separate courses on software quality and testing so that students do not face problems when they go for testing in industries. Today, the ideas and techniques of software testing have become essential knowledge for software developers, testers, and students as well. This book is a step forward to bridge this gap.

We cannot say that the industry is working smoothly, as far as software testing is concerned. While many industries have adopted effective software testing techniques and the development is driven by testing efforts, there are still some loopholes. Industries are dependent on automation of test execution. Therefore, testers also rely on efficient tools. But there may be an instance where automation will not help, which is why they also need to design test cases and execute them manually. Are the testers prepared for this case? This requires testing teams to have a knowledge of testing tactics and procedures of how to design test cases. This book discusses various techniques and demonstrates how to design test cases.

How do industries measure their testing process? Since software testing is a complete process today, it must be measured to check whether the process is suitable for projects. CMM (Capability Maturity Model) has measured the development process on a scale of 1–5 and companies are running for the highest scale. On the same pattern, there should be a measurement program for testing processes. Fortunately, the measurement technique for testing processes has also been developed. But how many managers, developers, testers, and of course students, know that we have a Testing Maturity Model (TMM) for measuring the maturity status of a testing process? This book gives an overview of various test process maturity models and emphasizes the need for these.

Summarizing the above discussion, it is evident that industry and academia should go parallel. Industries constantly aspire for high standards. Our university courses will have no value if their syllabi are not revised vis-à-vis industry requirements. Therefore, software testing should be included as a separate course in our curricula. On the other side, organizations cannot run with the development team looking after every stage, right from requirement gathering to implementation. Testing is an important segment of software development and it has to be thoroughly done. Therefore, there should be a separate testing group with divided responsibilities among the members.

In this chapter, we will trace the evolution of software testing. Once considered as a debugging process, it has now evolved into a complete process. Now we have software testing goals in place to have a clear picture as to why we want to study testing and execute test cases. There has been a misconception right from the evolution of software testing that it can be performed completely. But with time, we have grown out of this view and started focusing on effective testing rather than exhaustive testing. The psychology of a tester plays an important role in software testing. It matters whether one wants to show the absence of errors or their presence in the software. All these issues along with the model of testing, testing process, development of schools of testing, etc. have been discussed. This chapter presents an overview of effective software testing and its related concepts.

1.2 Evolution of Software Testing

In the early days of software development, software testing was considered only a debugging process for removing errors after the development of software. By 1970, the term 'software engineering' was in common use. But software testing was just a beginning at that time. In 1978, G. J. Myers realized the need to discuss the techniques of software testing in a separate subject. He wrote the book *The Art of Software Testing* [2] which is a classic work on software testing. He emphasized that there is a requirement that undergraduate students must learn software testing techniques so that they pass out with the basic knowledge of software testing and do not face problems in the industry. Moreover, Myers discussed the psychology of testing and emphasized that testing should be done with a mindset of finding errors and not to demonstrate that errors are not present.

By 1980, software professionals and organizations started emphasizing on quality. Organizations realized the importance of having quality assurance teams to take care of all testing activities for the project right from the beginning. In the 1990s, testing tools finally came into their own. There was a flood of various tools, which are absolutely vital to adequate testing of software systems. However, they do not solve all problems and cannot replace a testing process.

Gelperin and Hetzel [79] have characterized the growth of software testing with time. Based on this, we can divide the evolution of software testing into the following phases [80] (see Fig. 1.1).

Figure 1.1 Evolution phases of software testing

Debugging-oriented Phase (Before 1957)

This phase is the early period of testing. At that time, testing basics were unknown. Programs were written and then tested by the programmers until they were sure that all the bugs were removed. The term used for testing was *checkout*, focused on getting the system to run. Debugging was a more general term at that time and it was not distinguishable from software testing. Till 1956, there was no clear distinction between software development, testing, and debugging.

Demonstration-oriented Phase (1957–78)

The term 'debugging' continued in this phase. However, in 1957, Charles Baker pointed out that the purpose of checkout is not only to run the software but also to demonstrate the correctness according to the mentioned requirements. Thus, the scope of checkout of a program increased from program runs to program correctness. Moreover, the purpose of checkout was to show the absence of errors. There was no stress on the test case design. In this phase, there was a misconception that the software could be tested exhaustively.

Destruction-oriented Phase (1979–82)

This phase can be described as the revolutionary turning point in the history of software testing. Myers changed the view of testing from 'testing is to show the absence of errors' to 'testing is to find more and more errors.' He separated debugging from testing and stressed on the valuable test cases if they explore more bugs. This phase has given importance to effective testing in comparison to exhaustive testing. The importance of early testing was also realized in this phase.

Evaluation-oriented Phase (1983–87)

With the concept of early testing, it was realized that if the bugs were identified at an early stage of development, it was cheaper to debug them as compared to the bugs found in implementation or post-implementation phases. This phase stresses on the quality of software products such that it can be evaluated at every stage of development. In fact, the early testing concept was established in the form of verification and validation activities which help in producing better quality software. In 1983, guidelines by the National Bureau of Standards were released to choose a set of verification and validation techniques and evaluate the software at each step of software development.

Prevention-oriented Phase (1988–95)

The evaluation model stressed on the concept of bug-prevention as compared to the earlier concept of bug-detection. With the idea of early detection of bugs in earlier phases, we can prevent the bugs in implementation or further phases. Beyond this, bugs can also be prevented in other projects with the experience gained in similar software projects. The prevention model includes test planning, test analysis, and test design activities playing a major role, while the evaluation model mainly relies on analysis and reviewing techniques other than testing.

Process-oriented Phase (1996 onwards)

In this phase, testing was established as a complete process rather than a single phase (performed after coding) in the software development life cycle (SDLC). The testing process starts as soon as the requirements for a project are specified and it runs parallel to SDLC. Moreover, the model for measuring the performance of a testing process has also been developed like CMM. The model for measuring the testing process is known as Testing Maturity Model (TMM). Thus, the emphasis in this phase is also on quantification of various parameters which decide the performance of a testing process.

The evolution of software testing was also discussed by Hung Q. Nguyen and Rob Pirozzi in a white paper [81], in three phases, namely Software Testing 1.0, Software Testing 2.0, and Software Testing 3.0. These three phases discuss the evolution in the earlier phases that we described. According to this classification, the current state-of-practice is Software Testing 3.0. These phases are discussed below.

Software Testing 1.0

In this phase, software testing was just considered a single phase to be performed after coding of the software in SDLC. No test organization was there. A few testing tools were present but their use was limited due to high cost. Management was not concerned with testing, as there was no quality goal.

Software Testing 2.0

In this phase, software testing gained importance in SDLC and the concept of early testing also started. Testing was evolving in the direction of planning the test resources. Many testing tools were also available in this phase.

Software Testing 3.0

In this phase, software testing is being evolved in the form of a process which is based on strategic effort. It means that there should be a process which gives us a roadmap of the overall testing process. Moreover, it should be driven by quality goals so that all controlling and monitoring activities can be performed by the managers. Thus, the management is actively involved in this phase.

1.3 Software Testing—Myths and Facts

Before getting into the details of software testing, let us discuss some myths surrounding it. These myths are there, as this field is in its growing phase.

Myth *Testing is a single phase in SDLC .*

Truth It is a myth, at least in the academia, that software testing is just a phase in SDLC and we perform testing only when the running code of the module is ready. But in reality, testing starts as soon as we get the requirement specifications for the software. And the testing work continues throughout the SDLC, even post-implementation of the software.

Myth *Testing is easy.*

Truth This myth is more in the minds of students who have just passed out or are going to pass out of college and want to start a career in testing. So the general perception is that, software testing is an easy job, wherein test cases are executed with testing tools only. But in reality, tools are there to automate the tasks and not to carry out all testing activities. Testers' job is not easy, as they have to plan and develop the test cases manually and it requires a thorough understanding of the project being developed with its overall design. Overall, testers have to shoulder a lot of responsibility which sometimes make their job even harder than that of a developer.

Myth *Software development is worth more than testing.*

Truth This myth prevails in the minds of every team member and even in freshers who are seeking jobs. As a fresher, we dream of a job as a developer. We get into the organization as a developer and feel superior to other team members. At the managerial level also, we feel happy about the achievements of the developers but not of the testers who work towards the quality of the product being developed. Thus, we have this myth right from the beginning of our career, and testing is considered a secondary job. But testing has now

become an established path for job-seekers. Testing is a complete process like development, so the testing team enjoys equal status and importance as the development team.

Myth *Complete testing is possible.*

Truth This myth also exists at various levels of the development team. Almost every person who has not experienced the process of designing and executing the test cases manually feels that complete testing is possible. Complete testing at the surface level assumes that if we are giving all the inputs to the software, then it must be tested for all of them. But in reality, it is not possible to provide all the possible inputs to test the software, as the input domain of even a small program is too large to test. Moreover, there are many things which cannot be tested completely, as it may take years to do so. This will be demonstrated soon in this chapter. This is the reason why the term 'complete testing' has been replaced with 'effective testing.' Effective testing is to select and run some select test cases such that severe bugs are uncovered first.

Myth *Testing starts after program development.*

Truth Most of the team members, who are not aware of testing as a process, still feel that testing cannot commence before coding. But this is not true. As mentioned earlier, the work of a tester begins as soon as we get the specifications. The tester performs testing at the end of every phase of SDLC in the form of *verification* (discussed later) and plans for the *validation testing* (discussed later). He writes detailed test cases, executes the test cases, reports the test results, etc. Testing after coding is just a part of all the testing activities.

Myth *The purpose of testing is to check the functionality of the software.*

Truth Today, all the testing activities are driven by quality goals. Ultimately, the goal of testing is also to ensure quality of the software. But quality does not imply checking only the functionalities of all the modules. There are various things related to quality of the software, for which test cases must be executed.

Myth *Anyone can be a tester.*

Truth This is the extension of the myth that 'testing is easy.' Most of us think that testing is an intuitive process and it can be performed easily without any training. And therefore, anyone can be a tester. As an established process, software testing as a career also needs training for various purposes, such as to understand (i) various phases of software testing life cycle, (ii) recent techniques to design test cases, (iii) various tools and how to work on them, etc. This is the reason that various testing courses for certified testers are being run.

After having discussed the myths, we will now identify the requirements for software testing. Owing to the importance of software testing, let us first identify the concerns related to it. The next section discusses the goals of software testing.

1.4 Goals of Software Testing

To understand the new concepts of software testing and to define it thoroughly, let us first discuss the goals that we want to achieve from testing. The goals of software testing may be classified into three major categories, as shown in Fig. 1.2.

Figure 1.2 Software testing goals

Short-term or immediate goals These goals are the immediate results after performing testing. These goals may be set in the individual phases of SDLC. Some of them are discussed below.

Bug discovery The immediate goal of testing is to find errors at any stage of software development. More the bugs discovered at an early stage, better will be the success rate of software testing.

Bug prevention It is the consequent action of bug discovery. From the behaviour and interpretation of bugs discovered, everyone in the software development team gets to learn how to code safely such that the bugs discovered should not be repeated in later stages or future projects. Though errors cannot be prevented to zero, they can be minimized. In this sense, bug prevention is a superior goal of testing.

Long-term goals These goals affect the product quality in the long run, when one cycle of the SDLC is over. Some of them are discussed here.

Quality Since software is also a product, its quality is primary from the users' point of view. Thorough testing ensures superior quality. Therefore, the first goal of understanding and performing the testing process is to enhance the quality of the software product. Though quality depends on various factors, such as correctness, integrity, efficiency, etc., reliability is the major factor to achieve quality. The software should be passed through a rigorous reliability analysis to attain high quality standards. Reliability is a matter of confidence that the software will not fail, and this level of confidence increases with rigorous testing. The confidence in reliability, in turn, increases the quality, as shown in Fig. 1.3.

Figure 1.3 Testing produces reliability and quality

Customer satisfaction From the users' perspective, the prime concern of testing is customer satisfaction only. If we want the customer to be satisfied with the software product, then testing should be complete and thorough. Testing should be complete in the sense that it must satisfy the user for all the specified requirements mentioned in the user manual, as well as for the unspecified requirements which are otherwise understood. A complete testing process achieves reliability, reliability enhances the quality, and quality in turn, increases the customer satisfaction, as shown in Fig. 1.4.

Figure 1.4 Quality leads to customer satisfaction

Risk management Risk is the probability that undesirable events will occur in a system. These undesirable events will prevent the organization from successfully implementing its business initiatives. Thus, risk is basically concerned with the business perspective of an organization.

Risks must be controlled to manage them with ease. Software testing may act as a control, which can help in eliminating or minimizing risks (see Fig. 1.5).

Thus, managers depend on software testing to assist them in controlling their business goals. The purpose of software testing as a control is to provide information to management so that they can better react to risk situations [4]. For example, testing may indicate that the software being developed cannot be delivered on time, or there is a probability that high priority bugs will not be resolved by the specified time. With this advance information, decisions can be made to minimize risk situation.

Hence, it is the testers' responsibility to evaluate business risks (such as cost, time, resources, and critical features of the system being developed) and make the same a basis for testing choices. Testers should also categorize the levels of risks after their assessment (like high-risk, moderate-risk, low-risk) and this analysis becomes the basis for testing activities. Thus, risk management becomes the long-term goal for software testing.

Figure 1.5 Testing controlled by risk factors

Post-implementation goals These goals are important after the product is released. Some of them are discussed here.

Reduced maintenance cost The maintenance cost of any software product is not its physical cost, as the software does not wear out. The only maintenance cost in a software product is its failure due to errors. Post-release errors are costlier to fix, as they are difficult to detect. Thus, if testing has been done rigorously and effectively, then the chances of failure are minimized and in turn, the maintenance cost is reduced.

Improved software testing process A testing process for one project may not be successful and there may be scope for improvement. Therefore, the bug history and post-implementation results can be analysed to find out snags in the present testing process, which can be rectified in future projects. Thus, the long-term post-implementation goal is to improve the testing process for future projects.

1.5 Psychology for Software Testing

Software testing is directly related to human psychology. Though software testing has not been defined till now, but most frequently, it is defined as,

Testing is the process of demonstrating that there are no errors.

The purpose of testing is to show that the software performs its intended functions correctly. This definition is correct, but partially. If testing is performed keeping this goal in mind, then we cannot achieve the desired goals (described above in the previous section), as we will not be able to test the software as a whole. Myers first identified this approach of testing the software. This approach is based on the human psychology that human beings tend to work according to the goals fixed in their minds. If we have a preconceived assumption that the software is error-free, then consequently, we will design the test cases to show that all the modules run smoothly. But it may hide some bugs. On the other hand, if our goal is to demonstrate that a program has errors, then we will design test cases having a higher probability to uncover bugs.

Thus, if the process of testing is reversed, such that we always presume the presence of bugs in the software, then this psychology of being always suspicious of bugs widens the domain of testing. It means, now we don't think of testing only those features or specifications which have been mentioned in documents like SRS (software requirement specification), but we also think in terms of finding bugs in the domain or features which are understood but not specified. You can argue that, being suspicious about bugs in the software is a negative approach. But, this negative approach is for the benefit of constructive and effective testing. Thus, software testing may be defined as,

Testing is the process of executing a program with the intent of finding errors.

This definition has implications on the psychology of developers. It is very common that they feel embarrassed or guilty when someone finds errors in their software. However, we should not forget that humans are prone to error. We should not feel guilty for our errors. This psychology factor brings the concept that we should concentrate on discovering and preventing the errors and not feel guilt about them. Therefore, testing cannot be a joyous event unless you cast out your guilt.

According to this psychology of testing, a successful test is that which finds errors. This can be understood with the analogy of medical diagnostics of a patient. If the laboratory tests do not locate the problem, then it cannot be regarded as a successful test. On the other hand, if the laboratory test determines the disease, then the doctor can start an appropriate treatment. Thus, in

the destructive approach of software testing, the definitions of successful and unsuccessful testing should also be modified.

1.6 Software Testing Definitions

Many practitioners and researchers have defined software testing in their own way. Some are given below.

Testing is the process of executing a program with the intent of finding errors.

Myers [2]

A successful test is one that uncovers an as-yet-undiscovered error.

Myers [2]

Testing can show the presence of bugs but never their absence.

W. Dijkstra [125]

Program testing is a rapidly maturing area within software engineering that is receiving increasing notice both by computer science theoreticians and practitioners. Its general aim is to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances.

E. Miller[84]

Testing is a support function that helps developers look good by finding their mistakes before anyone else does.

James Bach [83]

Software testing is an empirical investigation conducted to provide stakeholders with information about the quality of the product or service under test, with respect to the context in which it is intended to operate.

Cem Kaner [85]

The underlying motivation of program testing is to affirm software quality with methods that can be economically and effectively applied to both large-scale and small-scale systems.

Miller [126]

Testing is a concurrent lifecycle process of engineering, using and maintaining testware (i.e. testing artifacts) in order to measure and improve the quality of the software being tested.

Craig [117]

Since quality is the prime goal of testing and it is necessary to meet the defined quality standards, software testing should be defined keeping in view the quality assurance terms. Here, it should not be misunderstood that the testing team is responsible for quality assurance. But the testing team must

be well aware of the quality goals of the software so that they work towards achieving them.

Moreover, testers these days are aware of the definition that testing is to find more and more bugs. But the problem is that there are too many bugs to fix. Therefore, the recent emphasis is on categorizing the more important bugs first. Thus, software testing can be defined as,

Software testing is a process that detects important bugs with the objective of having better quality software.

1.7 Model for Software Testing

Testing is not an intuitive activity, rather it should be learnt as a process. Therefore, testing should be performed in a planned way. For the planned execution of a testing process, we need to consider every element and every aspect related to software testing. Thus, in the testing model, we consider the related elements and team members involved (see Fig. 1.6).

Figure 1.6 Software testing model

The software is basically a part of a system for which it is being developed. Systems consist of hardware and software to make the product run. The developer develops the software in the prescribed system environment considering the testability of the software. Testability is a major issue for the developer while developing the software, as a badly written software may be difficult to test. Testers are supposed to get on with their tasks as soon as the requirements are specified. Testers work on the basis of a bug model which classifies the bugs based on the criticality or the SDLC phase in which the testing is to be performed. Based on the software type and the bug model, testers decide a testing methodology which guides how the testing will be performed. With suitable testing techniques decided in the testing methodology, testing is performed on the software with a particular goal. If the testing results are

in line with the desired goals, then the testing is successful; otherwise, the software or the bug model or the testing methodology has to be modified so that the desired results are achieved. The following describe the testing model.

Software and Software Model

Software is built after analysing the system in the environment. It is a complex entity which deals with environment, logic, programmer psychology, etc. But a complex software makes it very difficult to test. Since in this model of testing, our aim is to concentrate on the testing process, therefore the software under consideration should not be so complex such that it would not be tested. In fact, this is the point of consideration for developers who design the software. They should design and code the software such that it is testable at every point. Thus, the software to be tested may be modeled such that it is testable, avoiding unnecessary complexities.

Bug Model

Bug model provides a perception of the kind of bugs expected. Considering the nature of all types of bugs, a bug model can be prepared that may help in deciding a testing strategy. However, every type of bug cannot be predicted. Therefore, if we get incorrect results, the bug model needs to be modified.

Testing methodology and Testing

Based on the inputs from the software model and the bug model, testers can develop a testing methodology that incorporates both testing strategy and testing tactics. Testing strategy is the roadmap that gives us well-defined steps for the overall testing process. It prepares the planned steps based on the risk factors and the testing phase. Once the planned steps of the testing process are prepared, software testing techniques and testing tools can be applied within these steps. Thus, testing is performed on this methodology. However, if we don't get the required results, the testing plans must be checked and modified accordingly.

All the components described above will be discussed in detail in subsequent chapters.

1.8 Effective Software Testing vs. Exhaustive Software Testing

Exhaustive or complete software testing means that every statement in the program and every possible path combination with every possible combination of data must be executed. But soon, we will realize that exhaustive testing is out of scope. That is why the questions arise: (i) When are we done with testing? or (ii) How do we know that we have tested enough? There may be

many answers for these questions with respect to time, cost, customer, quality, etc. This section will explore that exhaustive or complete testing is not possible. Therefore, we should concentrate on effective testing which emphasizes efficient techniques to test the software so that important features will be tested within the constrained resources.

The testing process should be understood as a domain of possible tests (see Fig. 1.7). There are subsets of these possible tests. But the domain of possible tests becomes infinite, as we cannot test every possible combination.

Figure 1.7 Testing domain

This combination of possible tests is infinite in the sense that the processing resources and time are not sufficient for performing these tests. Computer speed and time constraints limit the possibility of performing all the tests. Complete testing requires the organization to invest a long time which is not cost-effective. Therefore, testing must be performed on selected subsets that can be performed within the constrained resources. This selected group of subsets, but not the whole domain of testing, makes effective software testing. Effective testing can be enhanced if subsets are selected based on the factors which are required in a particular environment.

Now, let us see in detail why complete testing is not possible.

The Domain of Possible Inputs to the Software is too Large to Test

If we consider the input data as the only part of the domain of testing, even then, we are not able to test the complete input data combination. The domain of input data has four sub-parts: (a) valid inputs, (b) invalid inputs, (c) edited inputs, and (d) race condition inputs (See Fig. 1.8)

Valid inputs It seems that we can test every valid input on the software. But look at a very simple example of adding two-digit two numbers. Their range is from –99 to 99 (total 199). So the total number of test case combinations will be $199 \times 199 = 39601$. Further, if we increase the range from two digits to four-digits, then the number of test cases will be 399,960,001. Most addition programs accept 8 or 10 digit numbers or more. How can we test all these combinations of valid inputs?

Invalid inputs Testing the software with valid inputs is only one part of the input sub-domain. There is another part, invalid inputs, which must be

tested for testing the software effectively. The important thing in this case is the behaviour of the program as to how it responds when a user feeds invalid inputs. The set of invalid inputs is also too large to test. If we consider again the example of adding two numbers, then the following possibilities may occur from invalid inputs:

(i) Numbers out of range
(ii) Combination of alphabets and digits
(iii) Combination of all alphabets
(iv) Combination of control characters
(v) Combination of any other key on the keyboard

Figure 1.8 Input domain for testing

Edited inputs If we can edit inputs at the time of providing inputs to the program, then many unexpected input events may occur. For example, you can add many spaces in the input, which are not visible to the user. It can be a reason for non-functioning of the program. In another example, it may be possible that a user is pressing a number key, then Backspace key continuously and finally after sometime, he presses another number key and Enter. Its input buffer overflows and the system crashes.

The behaviour of users cannot be judged. They can behave in a number of ways, causing defect in testing a program. That is why edited inputs are also not tested completely.

Race condition inputs The timing variation between two or more inputs is also one of the issues that limit the testing. For example, there are two input events,

A and B. According to the design, A precedes B in most of the cases. But, B can also come first in rare and restricted conditions. This is the race condition, whenever B precedes A. Usually the program fails due to race conditions, as the possibility of preceding B in restricted condition has not been taken care, resulting in a race condition bug. In this way, there may be many race conditions in the system, especially in multiprocessing systems and interactive systems. Race conditions are among the least tested.

There are too Many Possible Paths Through the Program to Test

A program path can be traced through the code from the start of a program to its termination. Two paths differ if the program executes different statements in each, or executes the same statements but in different order. A testing person thinks that if all the possible paths of control flow through the program are executed, then possibly the program can be said to be completely tested. However, there are two flaws in this statement.

(i) The number of unique logic paths through a program is too large. This was demonstrated by Myers[2] with an example shown in Fig. 1.9. It depicts a 10–20 statements program consisting of a DO loop that iterates up to 20 times. Within the body of the DO loop is a set of nested IF statements. The number of all the paths from point A to B is approximately 10^{14}. Thus, all these paths cannot be tested, as it may take years of time.

Figure 1.9 Sample flow graph 1

See another example for the code fragment shown in Fig. 1.10 and its corresponding flow graph in Fig. 1.11 (We will learn how to convert the program into a flow graph in Chapter 5).

```
for (int i = 0; i < n; ++i)
{
if (m >=0)
     x[i] = x[i] + 10;
else
     x[i] = x[i] - 2;
}
```

Figure 1.10 Sample code fragment

Figure 1.11 Example flow graph 2

Now calculate the number of paths in this fragment. For calculating the number of paths, we must know how many paths are possible in one iteration. Here in our example, there are two paths in one iteration. Now the total number of paths will be $2^n + 1$, where n is the number of times the loop will be carried out, and 1 is added, as the `for` loop will exit after its looping ends and it terminates. Thus, if n is 20, then the number of paths will be $2^{20} + 1$, i.e. 1048577. Therefore, all these paths cannot be tested, as it may take years.

(ii) The complete path testing, if performed somehow, does not guarantee that there will *not* be errors. For example, it does not claim that a program matches its specification. If one were asked to write an ascending order sorting program but the developer mistakenly produces a descending order program, then exhaustive path testing will be of little value. In another case, a program may be incorrect because of missing paths. In this case, exhaustive path testing would not detect the missing path.

Every Design Error Cannot be Found

Manna and Waldinger [15] have mentioned the following fact: 'We can never be sure that the specifications are correct.' How do we know that the specifications are achievable? Its consistency and completeness must be proved, and in general, that is a provably unsolvable problem [9]. Therefore, specification errors are one of the major reasons that make the design of the software faulty. If the user requirement is to have measurement units in inches and the specification says that these are in meters, then the design will also be in meters. Secondly, many user interface failures are also design errors.

The study of these limitations of testing shows that the domain of testing is infinite and testing the whole domain is just impractical. When we leave a single test case, the concept of complete testing is abandoned. But it does not mean that we should not focus on testing. Rather, we should shift our attention from exhaustive testing to effective testing. Effective testing provides the flexibility to select only the subsets of the domain of testing based on project priority such that the chances of failure in a particular environment is minimized.

1.9 Effective Testing is hard

We have seen the limitations of exhaustive software testing which makes it nearly impossible to achieve. Effective testing, though not impossible, is hard to implement. But if there is careful planning, keeping in view all the factors which can affect it, then it is implementable as well as effective. To achieve that planning, we must understand the factors which make effective testing difficult. At the same time, these factors must be resolved. These are described as follows.

Defects are hard to find The major factor in implementing effective software testing is that many defects go undetected due to many reasons, e.g. certain test conditions are never tested. Secondly, developers become so familiar with their developed system that they overlook details and leave some parts untested. So a proper planning for testing all the conditions should be done and independent testing, other than that done by developers, should be encouraged.

When are we done with testing This factor actually searches for the definition of effective software testing. Since exhaustive testing is not possible, we don't know what should be the criteria to stop the testing process. A software engineer needs more rigorous criteria for determining when sufficient testing has been performed. Moreover, effective testing has the limiting factor of cost, time, and personnel. In a nutshell, the criteria should be developed for enough

testing. For example, features can be prioritized which must be tested within the boundary of cost, time, and personnel of the project.

1.10 Software Testing as a Process

Since software development is an engineering activity for a quality product, it consists of many processes. As it was seen in testing goals, software quality is the major driving force behind testing. Software testing has also emerged as a complete process in software engineering (see Fig. 1.12). Therefore, our major concern in this text is to show that testing is not just a phase in SDLC normally performed after coding, rather software testing is a process which runs parallel to SDLC. In Fig. 1.13, you can see that software testing starts as soon as the requirements are specified. Once the SRS document is prepared, testing process starts. Some examples of test processes, such as test plan, test design, etc. are given. All the phases of testing life cycle will be discussed in detail in the next chapter.

Figure 1.12 Testing process emerged out of development process

Figure 1.13 Testing process runs parallel to software process

Software testing process must be planned, specified, designed, implemented, and quantified. Testing must be governed by the quality attributes of the software product. Thus, testing is a dual-purpose process, as it is used to detect bugs as well as to establish confidence in the quality of software.

An organization, to ensure better quality software, must adopt a testing process and consider the following points:

- Testing process should be organized such that there is enough time for important and critical features of the software.

- Testing techniques should be adopted such that these techniques detect maximum bugs.
- Quality factors should be quantified so that there is a clear understanding in running the testing process. In other words, the process should be driven by quantified quality goals. In this way, the process can be monitored and measured.
- Testing procedures and steps must be defined and documented.
- There must be scope for continuous process improvement.

All the issues related to testing process will be discussed in succeeding chapters.

1.11 Schools of Software Testing

Software testing has also been classified into some views according to some practitioners. They call these views or ideas as *schools of testing*. The idea of schools of testing was given by Bret Pettichord [82]. He has proposed the following schools:

Analytical School of Testing

In this school of testing, software is considered as a logical artifact. Therefore, software testing techniques must have a logico-mathematical form. This school requires that there must be precise and detailed specifications for testing the software. Moreover, it provides an objective measure of testing. After this, testers are able to verify whether the software conforms to its specifications. Structural testing is one example for this school of testing. Thus, the emphasis is on testing techniques which should be adopted.

This school defines software testing as a branch of computer science and mathematics.

Standard School of Testing

The core beliefs of this school of testing are:

1. Testing must be managed (for example, through traceability matrix. It will be discussed in detail in succeeding chapters). It means the testing process should be predictable, repeatable, and planned.
2. Testing must be cost-effective.
3. Low-skilled workers require direction.
4. Testing validates the product.
5. Testing measures development progress.

Thus, the emphasis is on measurement of testing activities to track the development progress.

This school defines software testing as a managed process.

The implications of this school are:

1. There must be clear boundaries between testing and other activities.
2. Plans should not be changed as it complicates progress tracking.
3. Software testing is a complete process.
4. There must be some test standards, best practices, and certification.

Quality School of Testing

The core beliefs of this school of testing are:

1. Software quality requires discipline.
2. Testing determines whether development processes are being followed.
3. Testers may need to police developers to follow the rules.
4. Testers have to protect the users from bad software.

Thus, the emphasis is to follow a good process.

This school defines software testing as a branch of software quality assurance.

The implications of this school are:

1. It prefers the term 'quality assurance' over 'testing.'
2. Testing is a stepping stone to 'process improvement.'

Context-driven School of Testing

This school is based on the concept that testing should be performed according to the context of the environment and project. Testing solutions cannot be the same for every context. For example, if there is a high-cost real-time defense project, then its testing plans must be different as compared to any daily-life low-cost project. Test plan issues will be different for both projects. Therefore, testing activities should be planned, designed, and executed keeping in view the context of environment in which testing is to be performed. The emphasis is to select a testing type that is valuable. Thus, context-driven testing can be defined as the testing driven by environment, type of project, and the intended use of software.

The implications of this school are:

1. Expect changes. Adapt testing plans based on test results.
2. Effectiveness of test strategies can only be determined with field research.

3. Testing research requires empirical and psychological study.
4. Focus on skill over practice.

Agile School of Testing

This type of school is based on testing the software which is being developed by iterative method of development and delivery. In this type of process model, the software is delivered in a short span of time; and based on the feedback, more features and capabilities are added. The focus is on satisfying the customer by delivering a working software quickly with minimum features and then improvising on it based on the feedback. The customer is closely related to the design and development of the software. Since the delivery timelines are short and new versions are built by modifying the previous one, chances of introducing bugs are high during the changes done to one version. Thus, regression testing becomes important for this software. Moreover, test automation also assumes importance to ensure the coverage of testing in a short span of time.

It can be seen that agile software development faces various challenges. This school emphasizes on all the issues related to agile testing.

1.12 Software Failure Case Studies

At the end of this chapter, let us discuss a few case studies that highlight the failures of some expensive and critical software projects. These case studies show the importance of software testing. Many big projects have failed in the past due to lack of proper software testing. In some instances, the product was replaced without question. The concerned parties had to bear huge losses in every case. It goes on to establish the fact that the project cost increases manifold if a product is launched without proper tests being performed on it. These case studies emphasize the importance of planning the tests, designing, and executing the test cases in a highly prioritized way, which is the central theme of this book.

Air Traffic Control System Failure (September 2004)

In September 2004, air traffic controllers in the Los Angeles area lost voice contact with 800 planes allowing 10 to fly too close together, after a radio system shut down. The planes were supposed to be separated by five nautical miles laterally, or 2,000 feet in altitude. But the system shut down while 800 planes were in the air, and forced delays for 400 flights and the cancellations of 600 more. The system had voice switching and control system, which gives controllers a touch-screen to connect with planes in flight and with controllers across the room or in distant cities.

The reason for failure was partly due to a 'design anomaly' in the way Microsoft Windows servers were integrated into the system. The servers were timed to shut down after 49.7 days of use in order to prevent a data overload. To avoid this automatic shutdown, technicians are required to restart the system manually every 30 days. An improperly trained employee failed to reset the system, leading it to shut down without warning.

Welfare Management System Failure (July 2004)

It was a new government system in Canada costing several hundred million dollars. It failed due to the inability to handle a simple benefits rate increase after being put into live operation. The system was not given adequate time for system and acceptance testing and never tested for its ability to handle a rate increase.

Northeast Blackout (August 2003)

It was the worst power system failure in North American history. The failure involved loss of electrical power to 50 million customers, forced shutdown of 100 power plants and economic losses estimated at $6 billion. The bug was reportedly in one utility company's vendor-supplied power monitoring and management system. The failures occurred when multiple systems trying to access the same information at once got the equivalent of busy signals. The software should have given one system precedent. The error was found and corrected after examining millions of lines of code.

Tax System Failure (March 2002)

This system was Britain's national tax system which failed in 2002 and resulted in more than 100,000 erroneous tax overcharges. It was suggested in the error report that the integration testing of multiple parts could not be done.

Mars Polar Lander Failure (December 1999)

NASA's Mars Polar Lander was to explore a unique region of the red planet; the main focus was on climate and water. The spacecraft was outfitted with a robot arm which was capable of digging into Mars in search for near-surface ice. It was supposed to gently set itself down near the border of Mars' southern polar cap. But it couldn't be successful to touch the surface of Mars. The communication was lost when it was 1800 meters away from the surface of Mars.

When the Lander's legs started opening for landing on Martian surface, there were vibrations which were identified by the software. This resulted in the vehicle's descent engines being cut off while it was still 40 meters above the surface, rather than on touchdown as planned. The software design failed to take into account that a touchdown signal could be detected before the

Lander actually touched down. The error was in design. It should have been configured to disregard touchdown signals during the deployment of the Lander's legs.

Mars Climate Orbiter Failure (September 1999)

Mars Climate Orbiter was one of a series of missions in a long-term program of Mars exploration managed by the Jet Propulsion Laboratory for NASA's Office of Space Science, Washington, DC. Mars Climate Orbiter was to serve as a communications relay for the Mars Polar Lander mission. But it disappeared as it began to orbit Mars. Its cost was about $125 million. The failure was due to an error in transfer of information between a team in Colorado and a team in California. This information was critical to the maneuvers required to place the spacecraft in the proper Mars orbit. One team used English units (e.g. inches, feet, and pounds), while the other team used metric units for a key spacecraft operation.

Stock Trading Service Failure (February 1999)

This was an online US stock trading service which failed during trading hours several times over a period of days in February 1999. The problem found was due to bugs in a software upgrade intended to speed online trade confirmations.

Intel Pentium Bug (April 1997)

Intel Pentium was also observed with a bug that is known as Dan-0411 or Flag Erratum. The bug is related to the operation where conversion of floating point numbers is done into integer numbers. All floating-point numbers are stored inside the microprocessor in an 80-bit format. Integer numbers are stored externally in two different sizes, i.e. 16 bits for short integers and 32 bits for long integers. It is often desirable to store the floating-point numbers as integer numbers. When the converted numbers won't fit the integer size range, a specific error flag is supposed to be set in a floating point status register. But the Pentium II and Pentium Pro fail to set this error flag in many cases.

The Explosion of Ariane 5 (June 1996)

Ariane 5 was a rocket launched by the European Space Agency. On 4 June 1996, it exploded at an altitude of about 3700 meters just 40 seconds after its lift-off from Kourou, French Guiana. The launcher turned off its flight path, broke up and exploded. The rocket took a decade of development time with a cost of $7 billion. The destroyed rocket and its cargo were valued at $500 million. The failure of Ariane was caused due to the complete loss of guidance and altitude information, 37 seconds after the start of main engine ignition sequence (30 seconds after lift-off).

A board of inquiry investigated the causes of the explosion and in two weeks issued a report. It was found that the cause of the failure was a software error in the inertial reference system (SRI). The internal SRI software exception was caused during the execution of a data conversion from 64-bit floating point to 16-bit signed integer value. A 64-bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16-bit signed integer. The number was larger than 32,767, the largest integer stored in a 16-bit signed integer; and thus the conversion failed. The error was due to specification and design errors in the software of the inertial reference system.

SUMMARY

This chapter emphasizes that software testing has emerged as a separate discipline. Software testing is now an established process. It is driven largely by the quality goals of the software. Thus, testing is the critical element of software quality. This chapter shows that testing cannot be performed with an optimistic view that the software does not contain errors. Rather, testing should be performed keeping in mind that the software always contains errors.

A misconception has prevailed through the evolution of software testing that complete testing is possible, but it is not true. Here, it has been demonstrated that complete testing is not possible. Thus, the term 'effective software testing' is becoming more popular as compared to 'exhaustive' or 'complete testing'. The chapter gives an overview of software testing discipline along with definitions of testing, model for testing, and different schools of testing. To realize the importance of effective software testing as a separate discipline, some case studies showing the software failures in systems have also been discussed.

Let us quickly review the important concepts described in this chapter.

- Software testing has evolved through many phases, namely (i) debugging-oriented phase, (ii) demonstration-oriented phase, (iii) destruction-oriented phase, (iv) evaluation-oriented phase, (v) prevention-oriented phase, and (vi) process-oriented phase.
- There is another classification for evolution of software testing, namely Software testing 1.0, Software testing 2.0, and Software testing 3.0.
- Software testing goals can be partitioned into following categories:
  1. Immediate goals
     - Bug discovery
     - Bug prevention
  2. Long-term goals
     - Reliability
     - Quality
     - Customer satisfaction
     - Risk management

3. Post-implementation goals
    - Reduced maintenance cost
    - Improved testing process

- Testing should be performed with a mindset of finding bugs. This suspicious strategy (destructive approach) helps in finding more and more bugs.
- Software testing is a process that detects important bugs with the objective of having better quality software.
- Exhaustive testing is not possible due to the following reasons:
    - It is not possible to test every possible input, as the input domain is too large.
    - There are too many possible paths through the program to test.
    - It is difficult to locate every design error.
- Effective software testing, instead of complete or exhaustive testing, is adopted such that critical test cases are covered first.
- There are different views on how to perform testing which have been categorized as schools of software testing, namely (i) analytical school, (ii) standard school, (iii) quality school, (iv) context school, and (v) agile school.
- Software testing is a complete process like software development.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Bug discovery is a _____ goal of software testing.
    (a) Long-term
    (b) Short-term
    (c) Post-implementation
    (d) All
2. Customer satisfaction and risk management are _____ goals of software testing.
    (a) Long-term
    (b) Short-term
    (c) Post-implementation
    (d) All
3. Reduced maintenance is a _____ goal of software testing.
    (a) Long-term
    (b) Short-term
    (c) Post-implementation
    (d) All

4. Software testing produces _____.
   (a) Reliability
   (b) Quality
   (c) Customer Satisfaction
   (d) All
5. Testing is the process of _____ errors.
   (a) Hiding
   (b) Finding
   (c) Removing
   (d) None
6. Complete testing is _____.
   (a) Possible
   (b) Impossible
   (c) None
7. The domain of possible inputs to the software is too _____ to test.
   (a) Large
   (b) Short
   (c) none
8. The set of invalid inputs is too _____ to test.
   (a) Large
   (b) Short
   (c) none
9. Race conditions are among the _____ tested.
   (a) Most
   (b) Least
   (c) None
10. Every design error _____ be found.
   (a) Can
   (b) Can definitely
   (c) Cannot
   (d) None

Review Questions

1. How does testing help in producing quality software?
2. 'Testing is the process of executing a program with the intent of finding errors.' Comment on this statement.
3. Differentiate between effective and exhaustive software testing.

4. Find out some myths related to software testing, other than those described in this chapter.
5. 'Every design error cannot be found.' Discuss this problem in reference to some project.
6. 'The domain of possible inputs to the software is too large to test.' Demonstrate using some example programs.
7. 'There are too many possible paths through the program to test.' Demonstrate using some example programs.
8. What are the factors for determining the limit of testing?
9. Explore some more software failure case studies other than those discussed in this chapter.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter

2

Software Testing Terminology and Methodology

OBJECTIVES

After reading this chapter, you should be able to understand:

- Difference between error, fault, and failure
- Life cycle of a bug
- How a bug affects the economics of software testing
- How the bugs are classified
- Testing principles
- Software Testing Life Cycle (STLC) and its models
- Difference between verification and validation
- Development of software testing methodology

Since software testing or rather effective software testing is an emerging field, many related terms are either undefined or yet to be known. We tend to use the terms 'error', 'bug', 'defect', 'fault', 'failure', etc. interchangeably. But they don't mean the same.

Bugs in general, are more important during software testing. All the bugs do not have the same level of severity. Some bugs are less important, as they are not going to affect the product badly. On the other hand, bugs with high severity level will affect the product such that the product may not be released. Therefore, bugs need to be classified according to their severity.

A bug has a complete cycle from its initiation to death. Moreover, these bugs also affect a project cost at various levels of development. If there is a bug at the specification level and it is caught, then it is more economical to debug it as compared to the case if it is found in the implementation stage. If the bug propagates further into later stages of software development life cycle (SDLC), it would become more costly to debug.

As we have discussed in the previous chapter, software testing today is recognized as a complete process, and we need to identify various stages of software testing life cycle. These stages, like SDLC stages, define various tasks to be done during the testing of software in a hierarchy. Besides this, what should be our complete methodology to test the software? How should we go for testing a complete project? What will be the techniques used during testing? Should we use testing tools during testing? This chapter defines all the terms related to software testing, some concepts related to these terms, and then discusses the development of a testing methodology.

2.1 Software Testing Terminology

2.1.1 Definitions

As mentioned earlier, terms like error, bug, failure, defect, etc. are not synonymous and hence these should not be used interchangeably. All these terms are defined below.

Failure When the software is tested, *failure* is the first term being used. It means the inability of a system or component to perform a required function according to its specification. In other words, when results or behaviour of the system under test are different as compared to specified expectations, then failure exists.

Fault/Defect/Bug Failure is the term which is used to describe the problems in a system on the output side, as shown in Fig. 2.1. *Fault* is a condition that in actual causes a system to produce failure. Fault is synonymous with the words *defect* or *bug*. Therefore, fault is the reason embedded in any phase of SDLC and results in failures. It can be said that failures are manifestation of bugs. One failure may be due to one or more bugs and one bug may cause one or more failures. Thus, when a bug is executed, then failures are generated. But this is not always true. Some bugs are hidden in the sense that these are not executed, as they do not get the required conditions in the system. So, hidden bugs may not always produce failures. They may execute only in certain rare conditions.

Figure 2.1 Testing terminology

Error Whenever a development team member makes a mistake in any phase of SDLC, errors are produced. It might be a typographical error, a misleading of a specification, a misunderstanding of what a subroutine does, and so on. Error is a very general term used for human mistakes. Thus, an error causes a bug and the bug in turn causes failures, as shown in Fig. 2.2.

Figure 2.2 Flow of faults

Example 2.1

Consider the following module in a software:

```
Module A()
{
    ….

    while(a > n+1);
    {
     …
         print("The value of x is", x);
     }

     ….
}
```

Suppose the module shown above is expected to print the value of `x` which is critical for the use of software. But when this module will be executed, the value of `x` will not be printed. It is a *failure* of the program. When we try to look for the reason of the failure, we find that in `Module A()`, the `while` loop is not being executed. A condition is preventing the body of `while` loop to be executed. This is known as *bug/defect/fault*. On close observation, we find a semicolon being misplaced after the `while` loop which is not its correct syntax and it is not allowing the loop to execute. This mistake is known as an *error*.

Test case Test case is a well-documented procedure designed to test the functionality of a feature in the system. A test case has an identity and is associated with a program behaviour. The primary purpose of designing a test case is to find errors in the system. For designing the test case, it needs to provide a set of inputs and its corresponding expected outputs. The sample for a test case is shown in Fig. 2.3.

Figure 2.3 Test case template

Test case ID is the identification number given to each test case.

Purpose defines why the case is being designed.

Preconditions for running the inputs in a system can be defined, if required, in a test case.

Inputs should not be hypothetical. Actual inputs must be provided, instead of general inputs. For example, if two integer numbers have to be provided as input, then specifically mention them as 23 and 56.

Expected outputs are the outputs which should be produced when there is no failure.

Test cases are designed based on the chosen testing techniques. They provide inputs when the system is executed. After execution, observed results are compared with expected outputs mentioned in the test case.

Testware The documents created during testing activities are known as *testware*. Taking the analogy from software and hardware as a product, testware are the documents that a test engineer produces. It may include test plans, test specifications, test case design, test reports, etc. Testware documents should also be managed and updated like a software product.

Incident When a failure occurs, it may or may not be readily apparent to the user. An *incident* is the symptom(s) associated with a failure that alerts the user about the occurrence of a failure.

Test oracle An *oracle* is the means to judge the success or failure of a test, i.e. to judge the correctness of the system for some test. The simplest oracle is comparing actual results with expected results by hand. This can be very time-consuming, so automated oracles are sought.

2.1.2 Life Cycle of a Bug

It should be clear that any member of the development team can make an error in any phase of SDLC. If an error has been produced in the requirement specification phase and not detected in the same phase, then it results in a bug in the next phase, i.e. the design phase. In the design phase, a bug has come from the previous stage, but an error can also be produced in this stage. Again, if the error in this phase is not detected and it passes on to the next stage, i.e. coding phase, then it becomes a bug. In this way, errors and bugs appear and travel through various stages of SDLC, as shown in Fig. 2.4. It means, one stage may contain errors as well as bugs and the bugs which come from the previous stage are harder to detect and debug.

In the testing phase, we analyse the incidents when the failure occurs. On the basis of symptoms derived from the incidents, a bug can be classified into certain categories. After this, the bug can be isolated in the corresponding phase of SDLC and resolved by finding its exact location.

The whole life cycle of a bug can be classified into two phases: (i) bugs-in phase and (ii) bugs-out phase.

Figure 2.4 Life cycle of a bug

Bugs-In Phase

This phase is where the errors and bugs are introduced in the software. Whenever we commit a mistake, it creates errors on a specific location of the software and consequently, when this error goes unnoticed, it causes some conditions to fail, leading to a bug in the software. This bug is carried out to the subsequent phases of SDLC, if not detected. Thus, a phase may have its own errors as well as bugs received from the previous phase. If you are not performing verification (discussed later in this chapter) on earlier phases, then there is no chance of detecting these bugs.

Bugs-Out Phase

If failures occur while testing a software product, we come to the conclusion that it is affected by bugs. However, there are situations when bugs are present, even though we don't observe any failures. That is another issue of discussion. In this phase, when we observe failures, the following activities are performed to get rid of the bugs.

Bug classification In this part, we observe the failure and classify the bugs according to its nature. A bug can be critical or catastrophic in nature or it may have no adverse effect on the output behaviour of the software. In this way, we classify all the failures. This is necessary, because there may be many bugs to be resolved. But a tester may not have sufficient time. Thus, categorization of bugs may help by handling high criticality bugs first and considering other trivial bugs on the list later, if time permits. We have taken various considerations to classify different bugs (discussed later in this chapter).

Bug isolation Bug isolation is the activity by which we locate the module in which the bug appears. Incidents observed in failures help in this activity. We observe the symptoms and back-trace the design of the software and reach the module/files and the condition inside it which has caused the bug. This is known as bug isolation.

Bug resolution Once we have isolated the bug, we back-trace the design to pinpoint the location of the error. In this way, a bug is resolved when we have found the exact location of its occurrence.

2.1.3 States of a Bug

Based on the above discussion, a bug attains the following different states in its life cycle (see Fig. 2.5).

New The state is new when the bug is reported first time by a tester.

Open The new state does not verify that the bug is genuine. When the test leader (test organization hierarchy will be discussed in Chapter 9) approves that the bug is genuine, its state becomes open.

Assign An open bug comes to the development team where the development team verifies its validity. If the bug is valid, a developer is assigned the job to fix it and the state of the bug now is 'ASSIGN'.

Deferred The developer who has been assigned to fix the bug will check its validity and priority. If the priority of the reported bug is not high or there is not sufficient time to test it or the bug does not have any adverse effect on the software, then the bug is changed to deferred state which implies the bug is expected to be fixed in next releases.

Rejected It may be possible that the developer rejects the bug after checking its validity, as it is not a genuine one.

Test After fixing the valid bug, the developer sends it back to the testing team for next round of checking. Before releasing to the testing team, the developer changes the bug's state to 'TEST'. It specifies that the bug has been fixed by the development team but not tested and is released to the testing team.

Verified/fixed The tester tests the software and verifies whether the reported bug is fixed or not. After verifying, the developer approves that the bug is fixed and changes the status to 'VERIFIED'.

Reopened If the bug is still there even after fixing it, the tester changes its status to 'REOPENED'. The bug traverses the life cycle once again. In another case, a bug which has been closed earlier may be reopened if it appears again. In this case, the status will be REOPENED instead of OPEN.

Closed Once the tester and other team members are confirmed that the bug is completely eliminated, they change its status to 'CLOSED'.

Figure 2.5 States of a bug

The states of a bug, as shown in Fig. 2.5, are in general and the terminology or sequence of hierarchy may differ in organizations.

2.1.4 Why do Bugs occur?

This is a very basic question for testers. The following points can answer this best:

To Err is Human

As discussed earlier, errors are produced when developers commit mistakes. Sample this. The phone rang while coding and the developer got distracted, he pressed a wrong key and the results of that mistake produced a bug in the system. But this is not the only reason for bugs. Consider the second point.

Bugs in Earlier Stages go Undetected and Propagate

The phases of SDLC are connected to each other and the output of one phase becomes the input to the next. Therefore, an undetected bug easily propagates into subsequent phases. These propagated bugs, in later phases, are harder to detect and if found, are costlier to debug. It is a proven fact that requirement specification and design phases contain the largest percentage of bugs.

This point also establishes the fact that testing should be performed at each level of SDLC. Errors are inevitable; however, we can prevent critical bugs from propagating into later stages by checking minute details at each level of SDLC.

How do bugs get into a software product? There may be various reasons: unclear or constantly changing requirements, software complexity, programming errors, timelines, errors in bug tracking, communication gap, documentation errors, deviation from standards, etc. Some examples are given below:

- Miscommunication in gathering requirements from the customer is a big source of bugs.
- If the requirements keep changing from time to time, it creates a lot of confusion and pressure, both on the development as well as the testing team. Often, a new feature added or an existing feature removed can be linked to other modules or components in the software.
- If the effect of changes in one module to another module is overlooked, it causes bugs.
- Rescheduling of resources, re-doing or discarding an already completed work, and changes in hardware/software requirements can affect the software. Assigning a new developer to the project midway can cause bugs. If proper coding standards have not been followed, then the new developer might not get all the relevant details of the project. Improper code documentation and ineffective knowledge transfer can limit the developer's ability to produce error-free codes. Discarding a portion of the existing code might just leave its trail behind in other parts of the software. Overlooking or not eliminating such code can cause bugs. Serious bugs can especially occur with larger projects, as it gets tougher to identify the problem area.
- Complexity in keeping a track of all the bugs can in turn cause more bugs. This gets harder when a bug has a very complex life cycle, i.e. when the number of times it has been closed, re-opened, not accepted, ignored, etc. goes on increasing.

2.1.5 Bugs Affect Economics of Software Testing

Studies have demonstrated that testing prior to coding is 50% effective in detecting errors and after coding, it is 80% effective. Moreover, it is at least 10 times as costly to correct an error after coding as before, and 100 times as

costly to correct a production error (post-release error). This is how the bugs affect the economics of testing.

There is no guarantee that all the errors will be recognized after testing and that the testing phase will remove all of them. This is not possible practically. If the bugs embedded in earlier stages go undetected, it is more difficult to detect them in later stages. So the cost to remove bugs from a system increases, as *the cost of a bug is equal to Detection Cost + Correction Cost.*

A bug found and fixed during early stages when the specification is being written, cost next to nothing. The same bug, if not found until the software is coded and tested, might cost ten times the cost in early stages. Moreover, after the release of the product, if a customer finds the same bug, the cost will be 100 times more. Figure 2.6 outlines the relative cost to correct a defect depending on the SDLC in which it is discovered. Therefore, cost increase is logarithmic.

Figure 2.6 Cost of debugging increases if bug propagates

2.1.6 Bug Classification Based on Criticality

Bugs can be classified based on the impact they have on the software under test. This classification can be used for the prioritization of bugs, as all bugs cannot be resolved in one release. Since all bugs are not of the same criticality, prioritization will put high criticality bugs on top of the list. We have divided bugs based on their criticality in the following broad categories:

Critical Bugs

This type of bugs has the worst effect such that it stops or hangs the normal functioning of the software. The person using the software becomes helpless when this type of bug appears. For example, in a sorting program, after providing the input numbers, the system hangs and needs to be reset.

Major Bug

This type of bug does not stop the functioning of the software but it causes a functionality to fail to meet its requirements as expected. For example, in a sorting program, the output is being displayed but not the correct one.

Medium Bugs

Medium bugs are less critical in nature as compared to critical and major bugs. If the outputs are not according to the standards or conventions, e.g. redundant or truncated output, then the bug is a medium bug.

Minor Bugs

These types of bugs do not affect the functionality of the software. These are just mild bugs which occur without any effect on the expected behaviour or continuity of the software. For example, typographical error or misaligned printout.

2.1.7 Bug Classification Based on SDLC

Since bugs can appear in any phase of SDLC, they can be classified based on SDLC phases which are described below [9, 48].

Requirements and Specifications Bugs

The first type of bug in SDLC is in the requirement gathering and specification phase. It has been observed that most of the bugs appear in this phase only. If these bugs go undetected, they propagate into subsequent phases. Requirement gathering and specification is a difficult phase in the sense that requirements gathered from the customer are to be converted into a *requirement specification* which will become the base for design. There may be a possibility that requirements specified are not exactly what the customers want. Moreover, specified requirements may be incomplete, ambiguous, or inconsistent. Specification problems lead to wrong missing, or superfluous features.

Design Bugs

Design bugs may be the bugs from the previous phase and in addition those errors which are introduced in the present phase. The following design errors may be there.

Control flow bugs If we look at the control flow of a program (through control flow graph–discussed in Chapter 5), then there may be many errors. For example, some paths through the flow may be missing; there may be unreachable paths, etc.

Logic bugs Any type of logical mistakes made in the design is a logical bug. For example, improper layout of cases, missing cases, improper combination of cases, misunderstanding of the semantics of the order in which a Boolean expression is evaluated.

Processing bugs Any type of computation mistakes result in processing bugs. Examples include arithmetic error, incorrect conversion from one data representation to another, ignoring overflow, improper use of logical operators, etc.

Data flow bugs Control flow cannot identify data errors. For this, we use data flow (through data flow graph–discussed in Chapter 5). There may be data flow anomaly errors like un-initialized data, initialized in wrong format, data initialized but not used, data used but not initialized, redefined without any intermediate use, etc.

Error handling bugs There may be errors about error handling in the software. There are situations in the system when exception handling mechanisms must be adopted. If the system fails, then there must be an error message or the system should handle the error in an appropriate way. If you forget to do all this, then error handling bugs appear.

Race condition bugs Race conditions (discussed in Chapter 1) also lead to bugs. Sometimes these bugs are irreproducible.

Boundary-related bugs Most of the time, the designers forget to take into consideration what will happen if any aspect of a program goes beyond its minimum and maximum values. For example, there is one integer whose range is between 1 to 100. What will happen if a user enters a value as 1 or 101? When the software fails at the boundary values, then these are known as boundary-related bugs. There may be boundaries in loop, time, memory, etc.

User interface bugs There may be some design bugs that are related to users. If the user does not feel good while using the software, then there are user interface bugs. Examples include inappropriate functionality of some features; not doing what the user expects; missing, misleading, or confusing information; wrong content in the help text; inappropriate error messages, etc.

Coding Bugs

There may be a long list of coding bugs. If you are a programmer, then you are aware of some common mistakes made. For example, undeclared data, undeclared routines, dangling code, typographical errors, documentation bugs, i.e. erroneous comments lead to bugs in maintenance.

Interface and Integration Bugs

External interface bugs include invalid timing or sequence assumptions related to external signals, misunderstanding external input and output formats, and user interface bugs. Internal interface bugs include input and output format bugs, inadequate protection against corrupted data, wrong subroutine call sequence, call parameter bugs, and misunderstood entry or exit parameter values. Integration bugs result from inconsistencies or incompatibilities between modules discussed in the form of interface bugs. There may be bugs in data transfer and data sharing between the modules.

System Bugs

There may be bugs while testing the system as a whole based on various parameters like performance, stress, compatibility, usability, etc. For example, in a real-time system, stress testing is very important, as the system must work under maximum load. If the system is put under maximum load at every factor like maximum number of users, maximum memory limit, etc. and if it fails, then there are system bugs.

Testing Bugs

One can question the presence of bugs in the testing phase because this phase is dedicated to finding bugs. But the fact is that bugs are present in testing phase also. After all, testing is also performed by testers – humans. Some testing mistakes are: failure to notice/report a problem, failure to use the most promising test case, failure to make it clear how to reproduce the problem, failure to check for unresolved problems just before the release, failure to verify fixes, failure to provide summary report.

2.1.8 Testing Principles

Now it is time to learn the testing principles that are largely based on the discussion covered in the first chapter and the present one. These principles can be seen as guidelines for a tester.

Effective testing, not exhaustive testing All possible combinations of tests become so large that it is impractical to test them all. So considering the domain of testing as infinite, exhaustive testing is not possible. Therefore, the tester's approach should be based on effective testing to adequately cover program logic and all conditions in the component level design.

Testing is not a single phase performed in SDLC Testing is not just an activity performed after the coding in SDLC. As discussed, the testing phase after coding is just a part of the whole testing process. Testing process starts as soon

as the specifications for the system are prepared and it continues till the release of the product.

Destructive approach for constructive testing Testers must have the psychology that bugs are always present in the program and they must think about the technique of how to uncover them (this is their art of creativity). This psychology of being always suspicious about bugs is a negative/destructive approach. However, it has been proved that such a destructive approach helps in performing constructive and effective testing. Thus, the criterion to have a successful testing is to discover more and more bugs, and not to show that the system does not contain any bugs.

Early testing is the best policy When is the right time to start the testing process? As discussed earlier and we will explore later that testing process is not a phase after coding, rather it starts as soon as requirement specifications are prepared. Moreover, the cost of bugs can be reduced tenfold, as bugs are harder to detect in later stages if they go undetected. Thus, the policy in testing is to start as early as possible.

Probability of existence of an error in a section of a program is proportional to the number of errors already found in that section Suppose the history of a software is that you found 50 errors in Module X, 12 in Module Y, and 3 in Module Z. The software was debugged but after a span of time, we find some errors again and the software is given to a tester for testing. Where should the tester concentrate to find the bugs? This principle says that the tester should start with Module X which has the history of maximum errors. Another way of stating it is that errors seem to come in clusters. The principle provides us the insight that if some sections are found error-prone in testing, then our next testing effort should give priority to these error-prone sections.

Testing strategy should start at the smallest module level and expand towards the whole program This principle supports the idea of incremental testing. Testing must begin at the unit or module level, gradually progressing towards integrated modules and finally the whole system. Testing cannot be performed directly on the whole system. It must start with the individual modules and slowly integrate the modules and test them. After this, the whole system should be tested.

Testing should also be performed by an independent team When programmers develop the software, they test it at their individual modules. However, these programmers are not good testers of their own software. They are basically constructors of the software, but testing needs a destructive approach.

Programmers always think positively that their code does not contain bugs. Moreover, they are biased towards the correct functioning of the specified requirements and not towards detecting bugs. Therefore, it is always recommended to have the software tested by an independent testing team. Testers associated with the same project can also help in this direction, but this is not effective. For effective testing, the software may also be sent outside the organization for testing.

Everything must be recorded in software testing As mentioned earlier, testing is not an intuitive process; rather it is a planned process. It demands that every detail be recorded and documented. We must have the record of every test case run and the bugs reported. Even the inputs provided during testing and the corresponding outputs are to be recorded. Executing the test cases in a recorded and documented way can greatly help while observing the bugs. Moreover, observations can be a lesson for other projects. So the experience with the test cases in one project can be helpful in other projects.

Invalid inputs and unexpected behaviour have a high probability of finding an error Whenever the software is tested, we test for valid inputs and for the functionality that the software is supposed to do. But thinking in a negative way, we must test the software with invalid inputs and the behaviour which is not expected in general. This is also a part of effective testing.

Testers must participate in specification and design reviews Testers' role is not only to get the software and documents and test them. If they are not participating in other reviews like specification and design, it may be possible that either some specifications are not tested or some test cases are built for no specifications.

Let us consider a program. Let S be the set of specified behaviours of the program, P be the implementation of the program, and T be the set of test cases. Now consider the following cases (see Fig. 2.7):

(i) There may be specified behaviours that are not tested (regions 2 and 5).

(ii) Test cases that correspond to unspecified behaviours (regions 3 and 7).

(iii) Program behaviours that are not tested (regions 2 and 6).

The good view of testing is to enlarge the area of region 1. Ideally, all three sets S, P, and T must overlap each other such that all specifications are implemented and all implemented specifications are tested. This is possible only when the test team members participate in all discussions regarding specifications and design.

Figure 2.7 Venn diagram for S, P, T

2.2 Software Testing Life Cycle (STLC)

Since we have recognized software testing as a process, like SDLC, there is need for a well-defined series of steps to ensure successful and effective software testing. This systematic execution of each step will result in saving time and effort. Moreover, the chances are that more number of bugs will be uncovered.

The testing process divided into a well-defined sequence of steps is termed as *software testing life cycle* (STLC). The major contribution of STLC is to involve the testers at early stages of development. This has a significant benefit in the project schedule and cost. The STLC also helps the management in measuring specific milestones.

STLC consists of the following phases (see Fig. 2.8):

Figure 2.8 Software testing life cycle

Test Planning

The goal of test planning is to take into account the important issues of testing strategy, viz. resources, schedules, responsibilities, risks, and priorities, as a roadmap. Test planning issues are in tune with the overall project planning. Broadly, following are the activities during test planning:

- Defining the test strategy.
- Estimate the number of test cases, their duration, and cost.
- Plan the resources like the manpower to test, tools required, documents required.
- Identifying areas of risks.
- Defining the test completion criteria.
- Identification of methodologies, techniques, and tools for various test cases.
- Identifying reporting procedures, bug classification, databases for testing, bug severity levels, and project metrics.

Based on the planning issues as discussed above, analysis is done for various testing activities. The major output of test planning is the *test plan document*. Test plans are developed for each level of testing. After analysing the issues, the following activities are performed:

- Develop a test case format.
- Develop test case plans according to every phase of SDLC.
- Identify test cases to be automated (if applicable).
- Prioritize the test cases according to their importance and criticality.
- Define areas of stress and performance testing.
- Plan the test cycles required for regression testing.

Test Design

One of the major activities in testing is the design of test cases. However, this activity is not an intuitional process; rather it is a well-planned process.

The test design is an important phase after test planning. It includes the following critical activities.

Determining the test objectives and their prioritization This activity decides the broad categories of things to test. The test objectives reflect the fundamental elements that need to be tested to satisfy an objective. For this purpose, you need to gather reference materials like software requirements specification and design documentation. Then on the basis of reference materials, a team

of experts compile a list of test objectives. This list should also be prioritized depending upon the scope and risk.

Preparing list of items to be tested The objectives thus obtained are now converted into lists of items that are to be tested under an objective.

Mapping items to test cases After making a list of items to be tested, there is a need to identify the test cases. A matrix can be created for this purpose, identifying which test case will be covered by which item. The existing test cases can also be used for this mapping. Thus it permits reusing the test cases. This matrix will help in:

(a) Identifying the major test scenarios.

(b) Identifying and reducing the redundant test cases.

(c) Identifying the absence of a test case for a particular objective and as a result, creating them.

Designing the test cases demands a prior analysis of the program at functional or structural level. Thus, the tester who is designing the test cases must understand the cause-and-effect connections within the system intricacies. But look at the rule quoted by Tsuneo Yamaura–*There is only one rule in designing test cases: Cover all features, but do not make too many test cases.*

Some attributes of a good test case are given below:

(a) A good test case is one that has been designed keeping in view the criticality and high-risk requirements in order to place a greater priority upon, and provide added depth for testing the most important functions [12].

(b) A good test case should be designed such that there is a high probability of finding an error.

(c) Test cases should not overlap or be redundant. Each test case should address a unique functionality, thereby not wasting time and resources.

(c) Although it is sometimes possible to combine a series of tests into one test case, a good test case should be designed with a modular approach so that there is no complexity and it can be reused and recombined to execute various functional paths. It also avoids masking of errors and duplication of test-creation efforts [7, 12].

(d) A successful test case is one that has the highest probability of detecting an as-yet-undiscovered error [2].

Selection of test case design techniques While designing test cases, there are two broad categories, namely *black-box testing* and *white-box testing*. Black-

box test case design techniques generate test cases without knowing the internal working of a system. These will be discussed later in this chapter. The techniques to design test cases are selected such that there is more coverage and the system detects more bugs.

Creating test cases and test data The next step is to create test cases based on the testing objectives identified. The test cases mention the objective under which a test case is being designed, the inputs required, and the expected outputs. While giving input specifications, test data must also be chosen and specified with care, as this may lead to incorrect execution of test cases.

Setting up the test environment and supporting tools The test created above needs some environment settings and tools, if applicable. So details like hardware configurations, testers, interfaces, operating systems, and manuals must be specified during this phase.

Creating test procedure specification This is a description of how the test case will be run. It is in the form of sequenced steps. This procedure is actually used by the tester at the time of execution of test cases.

Thus, the hierarchy for test design phase includes: developing test objectives, identifying test cases and creating their specifications, and then developing test case procedure specifications as shown in Fig. 2.9. All the details specified in the test design phase are documented in the test design specification. This document provides the details of the input specifications, output specifications, environmental needs, and other procedural requirements for the test case.

Figure 2.9 Test case design steps

Test Execution

In this phase, all test cases are executed including verification and validation. Verification test cases are started at the end of each phase of SDLC. Validation test cases are started after the completion of a module. It is the decision of the test team to opt for automation or manual execution. Test results are documented in the test incident reports, test logs, testing status, and test summary reports, as shown in Fig. 2.10. These will be discussed in detail in Chapter 9.

Figure 2.10 Documents in test execution

Responsibilities at various levels for execution of the test cases are outlined in Table 2.1.

Table 2.1 Testing level vs responsibility

Test Execution Level	Person Responsible
Unit	Developer of the module
Integration	Testers and Developers
System	Testers, Developers, End-users
Acceptance	Testers, End-users

Post-Execution/Test Review

As we know, after successful test execution, bugs will be reported to the concerned developers. This phase is to analyse bug-related issues and get feedback so that maximum number of bugs can be removed. This is the primary goal of all test activities done earlier.

As soon as the developer gets the bug report, he performs the following activities:

Understanding the bug The developer analyses the bug reported and builds an understanding of its whereabouts.

Reproducing the bug Next, he confirms the bug by reproducing the bug and the failure that exists. This is necessary to cross-check failures. However, some bugs are not reproducible which increases the problems of developers.

Analysing the nature and cause of the bug After examining the failures of the bug, the developer starts debugging its symptoms and tracks back to the actual location of the error in the design. The process of debugging has been discussed in detail in Chapter 17.

After fixing the bug, the developer reports to the testing team and the modified portion of the software is tested once again.

After this, the results from manual and automated testing can be collected. The final bug report and associated metrics are reviewed and analysed for overall testing process. The following activities can be done:

- *Reliability analysis* can be performed to establish whether the software meets the predefined reliability goals or not. If so, the product can be released and the decision on a release date can be taken. If not, then the time and resources required to reach the reliability goals are outlined.
- *Coverage analysis* can be used as an alternative criterion to stop testing.
- *Overall defect analysis* can identify risk areas and help focus our efforts on quality improvement.

2.3 Software Testing Methodology

Software testing methodology is the organization of software testing by means of which the test strategy and test tactics are achieved, as shown in Fig. 2.11. All the terms related to software testing methodology and a complete testing strategy is discussed in this section.

Figure 2.11 Testing methodology

2.3.1 Software Testing Strategy

Testing strategy is the planning of the whole testing process into a well-planned series of steps. In other words, strategy provides a roadmap that includes very specific activities that must be performed by the test team in order to achieve a specific goal.

In Chapter 1, risk reduction was described as one of the goals of testing. In fact, when a test strategy is being developed, risk reduction is addressed first. The components of a testing strategy are discussed below:

Test Factors

Test factors are risk factors or issues related to the system under development. Risk factors need to be selected and ranked according to a specific system under development. The testing process should reduce these test factors to a prescribed level.

Test Phase

This is another component on which the testing strategy is based. It refers to the phases of SDLC where testing will be performed. Testing strategy may be different for different models of SDLC, e.g. strategies will be different for waterfall and spiral models.

2.3.2 Test Strategy Matrix

A test strategy matrix identifies the concerns that will become the focus of test planning and execution. In this way, this matrix becomes an input to develop the testing strategy. The matrix is prepared using test factors and test phase (Table 2.2). The steps to prepare this matrix are discussed below.

Select and rank test factors Based on the test factors list, the most appropriate factors according to specific systems are selected and ranked from the most significant to the least. These are the rows of the matrix.

Identify system development phases Different phases according to the adopted development model are listed as columns of the matrix. These are called test phases.

Identify risks associated with the system under development In the horizontal column under each of the test phases, the test concern with the strategy used to address this concern is entered. The purpose is to identify the concerns that need to be addressed under a test phase. The risks may include any events, actions, or circumstances that may prevent the test program from being implemented or executed according to a schedule, such as late budget

approvals, delayed arrival of test equipment, or late availability of the software application. As risks are identified, they must be assessed for impact and then mitigated with strategies for overcoming them, because risks may be realized despite all precautions having been taken. The test team must carefully examine risks in order to derive effective test and mitigation strategies for a particular application. Concerns should be expressed as questions so that the test strategy becomes a high-level focus for testers when they reach the phase where it's most appropriate to address a concern.

Table 2.2 Test strategy matrix

Test Factors	Test Phase					
	Requirements	Design	Code	Unit test	Integration test	System test

Example 2.1

Creating a test strategy Let's take a project as an example. Suppose a new operating system has to be designed, which needs a test strategy. The following steps are used (Table 2.3).

Table 2.3 Example test strategy matrix

Test Factors	Test Phase					
	Requirements	Design	Code	Unit test	Integration test	System test
Portability	Is portability feature mentioned in specifications according to different hardware?					Is system testing performed on MIPS and INTEL platforms?
Service Level	Is time frame for booting mentioned?	Is time frame incorporated in design of the module?				

Select and rank test factors A critical factor to be considered for the development of an operating system is portability. This is the effort required to transfer a program from one hardware configuration to another. This factor matters the most, as the operating system has to be compatible with most hardware configurations.

Identify the test phases In this step, all test phases affected by the selected test factors are identified. All the affected test phases can be seen according to the test factors in Table 2.3.

Identify the risks associated with each test factor and its corresponding test phase All the risks are basic concerns associated with each factor in a phase, and are expressed in the form of a question. For example, 'Is testing performed successfully on INTEL, MIPS H/W platforms?'

Plan the test strategy for every risk identified After identifying the risks, it is required to plan a strategy to tackle them. It helps testers to start working on testing so that risks are mitigated.

Test strategies generally must concentrate on risks associated with cost overruns, schedule slippage, critical software errors, and other failures. Thus test-strategy design is developed with risks, constrained resources, time limits, and budget restrictions.

2.3.3 Development of Test Strategy

When the project under consideration starts and progresses, testing too starts from the first step of SDLC. Therefore, the test strategy should be such that the testing process continues till the implementation of project. Moreover, the rule for development of a test strategy is that testing 'begins from the smallest unit and progresses to enlarge'. This means the testing strategy should start at the component level and finish at the integration of the entire system. Thus, a test strategy includes testing the components being built for the system, and slowly shifts towards testing the whole system. This gives rise to two basic terms—*Verification* and *Validation*—the basis for any type of testing. It can also be said that the testing process is a combination of verification and validation.

The purpose of verification is to check the software with its specification at every development phase such that any defect can be detected at an early stage of testing and will not be allowed to propagate further. That is why verification can be applied to all stages of SDLC. So *verification* refers to the set of activities that ensures correct implementation of functions in a software. However, as we progress down to the completion of one module or system development, the scope of verification decreases. The validation process starts replacing the verification in the later stages of SDLC. *Validation* is a very general term to test the software as a whole in conformance with customer expectations. According to Boehm [5]–

Verification is 'Are we building the product right?'

Validation is 'Are we building the right product?'

You can relate verification and validation to every complex task of daily life. You divide a complex task into many sub-tasks. In this case, every sub-task is developed and accomplished towards achieving the complex task. Here, you check every sub-task to ensure that you are working in the right direction. This is *verification.* After the sub-tasks have been completed and merged, the entire task is checked to ensure the required task goals have been achieved. This is *validation.*

Verification is checking the work at intermediate stages to confirm that the project is moving in the right direction, towards the set goal.

When a module is prepared with various stages of SDLC like plan, design and code, it is verified at every stage. But there may be various modules in the project. These need to be integrated, after which the full system is built. However, if we simply integrate the modules and build the system for the customer, then we are leaving open spaces for the bugs to intrude. Therefore, after building individual modules, the following stages need to be tested: the module as a whole, integration of modules, and the system built after integration. This is *validation testing.*

2.3.4 Testing Life Cycle Model

Verification and validation (V&V) are the building blocks of a testing process. The formation of test strategy is based on these two terms only. V&V can be best understood when these are modeled in the testing process. This model is known as the Testing Life Cycle Model. Life cycle involves continuous testing of the system during the development process. At predetermined points, the results of the development process are inspected to determine the correctness of implementation. These inspections identify defects as early as possible. But, life cycle testing is dependent upon the completion of predetermined deliverables at a specified point in the development life cycle. It also shows the paths for various types of testing.

V-Testing Life Cycle Model

In V-testing concept [4], as the development team attempts to implement the software, the testing team concurrently starts checking the software. When the project starts, both the system development and the system test process begin. The team that is developing the system begins the system development process and the team that is conducting the system test begins planning the system test process, as shown in Fig. 2.12. Both teams start at the same point using the same information. If there are risks, the tester develops a process to minimize or eliminate them.

Figure 2.12 V-testing model

Figure 2.12 shown above can also be expanded. In Fig. 2.13, on the left arm of the V, the development cycle is progressing and on the right arm, the corresponding testing stages are moving. As said earlier, in the early stages of SDLC, testing comprises more verification activities and towards the later stages, the emphasis is on validation.

Figure 2.13 Expanded V-testing model

The V&V process, in a nutshell, involves (i) verification of every step of SDLC (all verification activities are explained in the next chapter) and (ii) validation of the verified system at the end.

2.3.5 Validation Activities

Validation has the following three activities which are also known as the *three levels of validation testing.*

Unit Testing

It is a major validation effort performed on the smallest module of the system. If avoided, many bugs become latent bugs and are released to the customer. Unit testing is a basic level of testing which cannot be overlooked, and confirms the behaviour of a single module according to its functional specifications.

Integration Testing

It is a validation technique which combines all unit-tested modules and performs a test on their aggregation. One may ask, when we have tested all modules in unit testing, where is the need to test them on aggregation? The answer is *interfacing.* Unit modules are not independent, and are related to each other by interface specifications between them. When we unit test a module, its interfacing with other modules remain untested. When one module is combined with another in an integrated environment, interfacing between units must be tested. If some data structures, messages, or other things are common between some modules, then the standard format of these interfaces must be checked during integration testing, otherwise these will not be able to interface with each other.

But how do we integrate the units together? Is it a random process? It is actually a systematic technique for combining modules. In fact, interfacing among modules is represented by the system design. We integrate the units according to the design and availability of units. Therefore, the tester must be aware of the system design.

System Testing

This testing level focuses on testing the entire integrated system. It incorporates many types of testing, as the full system can have various users in different environments. The purpose is to test the validity for specific users and environments. The validity of the whole system is checked against the requirement specifications.

2.3.6 Testing Tactics

The ways to perform various types of testing under a specific test strategy are discussed below.

Software Testing Techniques

In the previous sections, it has been observed that complete or exhaustive testing is not possible. Instead, our effort should be on effective testing. How-

ever, effective testing is a real challenge in the domain of testing. At this stage, testing can be defined as the design of effective test cases such that most of the testing domains will be covered detecting the maximum number of bugs. As Myers [2] said–

> *Given constraints on time, cost, computer time, etc., the key issue of testing becomes – What subset of all possible test cases has the highest probability of detecting the most errors?*

Therefore, the next objective is to find the technique which will meet both the objectives of effective test case design, i.e. coverage of testing domain and detection of maximum number of bugs. The technique used to design effective test case is called *Software Testing Technique.*

Till now, we have discussed the overall strategy for testing methodology. V&V and the levels of testing under this process describe only the organization and planning of software testing. Actual methods for designing test cases, i.e. software testing techniques, implement the test cases on the software. These techniques can be categorized into two parts: (a) static testing and (b) dynamic testing.

Static Testing

It is a technique for assessing the structural characteristics of source code, design specifications or any notational representation that conforms to well-defined syntactic rules [16]. It is called as *static* because we never execute the code in this technique. For example, the structure of code is examined by the teams but the code is not executed.

Dynamic Testing

All the methods that execute the code to test a software are known as dynamic testing techniques. In this technique, the code is run on a number of inputs provided by the user and the corresponding results are checked. This type of testing is further divided into two parts: (a) black-box testing and (b) white-box testing.

Black-box testing This technique takes care of the inputs given to a system and the output is received after processing in the system. What is being processed in the system? How does the system perform these operations? Black-box testing is not concerned with these questions. It checks the functionality of the system only. That is why the term black-box is used. It is also known as *functional testing.* It is used for system testing under validation.

White-box testing This technique complements black-box testing. Here, the system is not a black box. Every design feature and its corresponding code is checked logically with every possible path execution. So, it takes care of the structural paths instead of just outputs. It is also known as *structural testing* and is used for unit testing under verification.

Testing Tools

Testing tools provide the option to automate the selected testing technique with the help of tools. A tool is a resource for performing a test process. The combination of tools and testing techniques enables the test process to be performed. The tester should first understand the testing techniques and then go for the tools that can be used with each of the techniques.

2.3.7 Considerations in Developing Testing Methodologies

The considerations in developing a testing methodology are described below [4].

Determine Project Risks

A test strategy is developed with the help of another team familiar with the business risks associated with the software. The major issue in determining the strategy is to identify the risks associated with the project. What are the high priority risks? What are the consequences, if risks are not handled?

Determine the Type of Development Project

The environment or methodology to develop the software also affects the testing risks. The risks associated with a new development project will be different from a maintenance project or a purchased software.

Identify Test Activities According to SDLC Phase

After identifying all the risk factors in an SDLC phase, testing can be started in that phase. However, all the testing activities must be recognized at all the SDLC phases.

Build the Test Plan

A tactical test plan is required to start testing. This test plan provides:

- Environment and pre-test background
- Overall objectives
- Test team
- Schedule and budget showing detailed dates for the testing
- Resource requirements including equipment, software, and personnel
- Testing materials including system documentation, software to be tested, test inputs, test documentation, test tools
- Specified business functional requirements and structural functions to be tested
- Type of testing technique to be adopted in a particular SDLC phase or what are the specific tests to be conducted in a particular phase

Since the internal design of a software may be composed of many components, each of these components or units must also have its own test plan. The idea is that units should not be submitted without testing. So an extra effort must be put in preparing unit test plans also.

SUMMARY

Software testing fundamentals have been discussed in this chapter. There is a lot of confusion about using the testing terms. Some terms are used interchangeably, while they imply different meanings. This chapter makes all the related definitions clear. Since the basis for software testing are bugs, their complete life cycle, right from initiation to post-execution, has been described.

The testing principles have been elaborated which will guide the reader in performing every activity related to testing and help in understanding the subsequent chapters. Software testing has matured as a process, thus testing life cycle and its models have been described. Finally, a testing methodology that consists of a roadmap for performing tests has been provided. It will help the reader understand the hierarchy in the test strategy and what is to be done at every phase of testing.

Let us review the important concepts described in this chapter:

- Failure of a program is the mismatch between expected and actual results.
- Fault is a condition in the program that causes failures. This is also known as bug or defect.
- Error is the actual location in the program where a mistake has been made that produced the bug.
- Test case is a well-documented procedure designed to test the functionality of a feature in the system.
- The documents created during testing activities are known as Testware.
- An incident is the symptom(s) associated with a failure that alerts the user to the occurrence of a failure.
- An oracle is the means to judge the success or failure of a test.
- A bug propagates to all the stages of SDLC, if not detected.
- An undetected bug is harder and costlier to detect and debug at a later stage. The earlier you detect, the better.
- Bugs can be classified based on their criticality as: (a) critical, (b) major, (c) medium, and (d) minor.
- Bugs can be classified based on their SDLC phase in which they have been introduced as: (a) requirement and specification bugs, (b) design bugs, (c) coding bugs, (d) interface & integration bugs, (e) system bugs, and (f) testing bugs.
- Software testing life cycle consists of the following phases: (a) test planning, (b) test design, (c) test execution, and (d) post-implementation/test review.

- Software testing methodology is the organization of software testing by means of which the test strategy and test tactics are achieved.
- Testing strategy is planning the whole testing process into a well-planned series of steps.
- A test strategy matrix identifies the concerns that will become the focus of test planning and execution. The matrix is prepared using the test factors and test phase.
- The test strategy has two parts: Verification and Validation
- Verification is to check the software with its specification at every development phase such that any defect can be detected at an early stage of testing.
- Validation is to test the software as a whole in conformance with customer expectations.
- V-testing model emphasizes the concept of early testing. In this model, the scope of verification is more in the early phases and decreases slowly as one module is ready. After this, the scope of validation increases.
- There are three validation activities, generally known as levels of testing: Unit testing, Integration testing, and System testing.
- There are two types of testing techniques: Static and Dynamic testing.
- Static testing is to test the software and its documents without executing the software.
- Dynamic testing is to test the software by executing it.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Fault is synonymous with the word _____.
   (a) Failure
   (b) Defect
   (c) Error
   (d) All of the above
2. The inability of a system or component to perform a required function according to its specification is called as _____.
   (a) Failure
   (b) Bug
   (c) Error
   (d) None of the above
3. Testware includes _____.
   (a) test planning document
   (b) test data
   (c) test specifications
   (d) All of the above

4. Symptom(s) associated with a failure that alerts the user to the occurrence of a failure is called _____.
   (a) Bug
   (b) Error
   (c) Defect
   (d) Incident
5. Testing process starts as soon as the _____ for the system are prepared.
   (a) Design
   (b) Coding
   (c) Specifications
   (d) None of the above
6. Testing strategy should start at the _____ module level and expand towards the whole program.
   (a) Smallest
   (b) Largest
   (c) None of the above
7. Testing is a _____ process.
   (a) Intuitive
   (b) Random
   (c) Planned
   (d) None of the above
8. Planning the whole testing process into a well-planned series of steps is called _____.
   (a) Test strategy matrix
   (b) Test factor
   (c) Test phase
   (d) Test strategy
9. The test strategy matrix is prepared using the _____.
   (a) test planning and execution
   (b) test factor and test phase
   (c) test factor
   (d) test phase
10. 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 is called _____.
   (a) Verification
   (b) Validation
   (c) SDLC
   (d) None of the above

11. 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 is called _____.
    (a) Verification
    (b) Validation
    (c) SDLC
    (d) None of the above
12. In the early stages of SDLC, testing comprises more _____ activities and towards the later stages, the emphasis is on the _____ activities.
    (a) verification, validation
    (b) validation, verification
    (c) integration, coding
    (d) None
13. Technique for assessing the structural characteristics of source code, design specifications, or any notational representation that conforms to well-defined syntactic rules is called _____.
    (a) Dynamic testing
    (b) Static Testing
    (c) Black-Box Testing
    (d) None of the above
14. Every design feature and its corresponding code is checked logically with every possible path execution in _____.
    (a) Black-box testing
    (b) White-box testing
    (c) Testing tool
    (d) None of the above

Review Questions

1. Differentiate error, bug, defect, fault, failure, testware, and incident, giving examples of each.
2. How many states does a bug have?
3. Take a live project and demonstrate the fact that 'Bugs in earlier stages go undetected and propagate'.
4. Collect examples of some commercial products that prove the statement 'Bug affects the economics of Software Testing'
5. Give examples of each category of bug classified based on criticality.
6. Give examples of each category of bug classified based on SDLC.
7. How do you support destructive approach for software testing?

8. What are the benefits of early testing?
9. Demonstrate the fact that in a live project—'The probability of existence of an error in a section of a program is proportional to the number of errors already found in that section.'
10. What will happen if we test the whole system directly?
11. What is the psychology behind testing by an independent team?
12. 'Invalid inputs and unexpected behaviour have a high probability of finding an error.' Prove it in a live project.
13. What is the significance of enlarging the area of region 1 in Fig. 2.7?
14. Explain the different stages of STLC?
15. What is the use of test strategy matrix? Explain with an example other than provided in this chapter.
16. What is the difference between
    (a) Verification and Validation
    (b) Static and Dynamic testing
    (c) Black-box and White-box testing

    Explain with examples.
17. Take a project and identify the Test Activities according to SDLC phases.
18. What are the features of V-testing model? Explain in detail.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter

3

Verification and Validation

OBJECTIVES

After reading this chapter, you should be able to understand:

- V-diagram provides the basis for every type of software testing
- Various verification activities
- Various validation activities
- How to perform verification at each stage of SDLC
- Various validation test plans

In the last chapter, it has been seen that a testing strategy may be developed in the form of verification and validation, as shown in the V-diagram (see Fig. 3.1).

A V-diagram provides the following insights about software testing:

- Testing can be implemented in the same flow as for the SDLC.
- Testing can be broadly planned in two activities, namely verification and validation.
- Testing must be performed at every step of SDLC.
- V-diagram supports the concept of early testing.
- V-diagram supports parallelism in the activities of developers and testers.

Figure 3.1 V-Testing

- The more you concentrate on the V&V process, more will be the cost-effectiveness of the software.
- Testers should be involved in the development process.

Since testing is a critical element of quality, verification and validation also relate to the quality of the software. So it can be said that all testing activities can be seen in the form of verification and validation.

In this chapter, the V-diagram has been expanded to get a clear vision of verification and validation activities in SDLC. This expanded V-diagram called the V&V activities diagram gives us some more insights.

3.1 Verification and Validation (V&V) Activities

V&V activities can be understood in the form of a diagram which is described here. To understand this diagram, we first need to understand SDLC phases. After this, verification and validation activities in those SDLC phases will be described. The following SDLC phases [2] (see Fig. 3.2) are considered:

Figure 3.2 SDLC Phases

Requirements gathering The needs of the user are gathered and translated into a written set of requirements. These requirements are prepared from the user's viewpoint only and do not include any technicalities according to the developer.

Requirement specification or objectives In this phase, all the user requirements are specified in developer's terminology. The specified objectives from the full system which is going to be developed, are prepared in the form of a document known as *software requirement specification* (SRS).

Functional design or high-level design Functional design is the process of translating user requirements into a set of external interfaces. The output of the process is the functional design specification, which describes the product's behaviour as seen by an observer external to the product. The *high-level design* is prepared with SRS and software analysts convert the requirements into a usable product. In HLD, the software system architecture is prepared and broken into independent modules. Thus, an HLD document will contain the following items at a macro level:

1. Overall architecture diagrams along with technology details
2. Functionalities of the overall system with the set of external interfaces
3. List of modules
4. Brief functionality of each module
5. Interface relationship among modules including dependencies between modules, database tables identified along with key elements

Internal design or low-level design Since HLD provides the macro-level details of a system, an HLD document cannot be given to programmers for coding. So the analysts prepare a micro-level design document called *internal design* or *low-level design* (LLD). This document describes each and every module in an elaborate manner, so that the programmer can directly code the program based on this. There may be at least one separate document for each module.

Coding If an LLD document is prepared for every module, then it is easy to code the module. Thus in this phase, using design document for a module, its coding is done.

After understanding all the SDLC phases, we need to put together all the verification activities. As described earlier, verification activities are performed

almost at every phase, therefore all the phases described above will be verified, as shown in Fig. 3.3. Along with these verification activities performed at every step, the tester needs to prepare some test plans which will be used in validation activities performed after coding the system. These test plans are prepared at every SDLC phase (see Fig. 3.3).

Figure 3.3 Tester performs verification and prepares test plan during every SDLC phase

When the coding is over for a unit or a system, and parallel verification activities have been performed, then the system can be validated. It means validation activities can be performed now. These are executed with the help of test plans prepared by the testers at every phase of SDLC. This makes the complete V&V activities diagram (see Fig. 3.4) which is discussed in detail in the following sections.

Figure 3.4 V&V diagram

3.2 Verification

We have seen that verification is a set of activities that ensures correct implementation of specific functions in a software. What is the need of verification? Can't we test the software in the final phase of SDLC? Under the V&V process, it is mandatory that verification is performed at every step of SDLC. Verification is needed for the following:

- If verification is not performed at early stages, there are always a chance of mismatch between the required product and the delivered product. Suppose, if requirements are not verified, then it leads to situations where requirements and commitments are not clear. These things become important in case of non-functional requirements and they increase the probability of bugs.

- Verification exposes more errors.
- Early verification decreases the cost of fixing bugs.
- Early verification enhances the quality of software.

After understanding the need of verification, the goals of verification can be formalized which are discussed below [17].

Everything Must be Verified

In principle, all the SDLC phases and all the products of these processes must be verified.

Results of Verification May Not be Binary

Verification may not just be the acceptance or rejection of a product. There are many verification activities whose results cannot be reduced to a binary answer. Often, one has to accept approximations. For example, sometimes correctness of a requirement cannot be rejected or accepted outright, but can be accepted with a degree of satisfaction or rejected with a certain degree of modification.

Even Implicit Qualities Must be Verified

The qualities desired in the software are explicitly stated in the SRS. But those requirements which are implicit and not mentioned anywhere must also be verified.

3.2.1 Verification Activities

All the verification activities are performed in connection with the different phases of SDLC. The following verification activities have been identified:

- Verification of Requirements and Objectives
- Verification of High-Level Design
- Verification of Low-Level Design
- Verification of Coding (Unit Verification)

3.3 Verification of Requirements

In this type of verification, all the requirements gathered from the user's viewpoint are verified. For this purpose, an acceptance criterion is prepared. An acceptance criterion defines the goals and requirements of the proposed system and acceptance limits for each of the goals and requirements. The acceptance criteria matter the most in case of real-time systems where perfor-

mance is a critical issue in certain events. For example, if a real-time system has to process and take decision for a weapon within *x* seconds, then any performance below this would lead to rejection of the system. Therefore, it can be concluded that acceptance criteria for a requirement must be defined by the designers of the system and should not be overlooked, as they can create problems while testing the system.

The tester works in parallel by performing the following two tasks:

1. The tester reviews the acceptance criteria in terms of its completeness, clarity, and testability. Moreover, the tester understands the proposed system well in advance so that necessary resources can be planned for the project [13].
2. The tester prepares the *Acceptance Test Plan* which is referred at the time of *Acceptance Testing* (discussed later).

3.3.1 Verification of Objectives

After gathering requirements, specific objectives are prepared considering every specification. These objectives are prepared in a document called software requirement specification (SRS). In this activity also, two parallel activities are performed by the tester:

1. The tester verifies all the objectives mentioned in SRS. The purpose of this verification is to ensure that the user's needs are properly understood before proceeding with the project.
2. The tester also prepares the *System Test Plan* which is based on SRS. This plan will be referenced at the time of *System Testing* (discussed later).

In verifying the requirements and objectives, the tester must consider both functional and non-functional requirements. Functional requirements may be easy to comprehend. But non-functional requirements pose a challenge to testers in terms of understanding, quantifying, test planning, and test execution.

3.3.2 How to Verify Requirements and Objectives?

Requirement and objectives verification has a high potential of detecting bugs. Therefore, requirements must be verified. As stated above, the testers use the SRS for verification of objectives. One characteristic of a good SRS is that it can be verified. An SRS [18] can be verified, if and only if, every requirement stated herein can be verified. A requirement can be verified, if, and only if, there is some procedure to check that the software meets its requirement. It is a good idea to specify the requirements in a quantification manner. It means that ambiguous statements or language like 'good quality', 'usually', 'may

happen' should be avoided. Instead of this, quantified specifications should be provided. An example of a verifiable statement is

'Module *x* will produce output within 15 sec of its execution.'

OR

'The output should be displayed like this: **TRACK A's speed is *x***'.

It is clear now that verification starts from the requirement phase and every requirement specified in the SRS must be verified. But what are the points against which verification of requirement will be done? Following are the points against which every requirement in SRS should be verified:

Correctness

There are no tools or procedures to measure the correctness of a specification. The tester uses his or her intelligence to verify the correctness of requirements. Following are some points which can be adopted (these points can change according to situation):

(a) Testers should refer to other documentations or applicable standards and compare the specified requirement with them.

(b) Testers can interact with customers or users, if requirements are not well-understood.

(c) Testers should check the correctness in the sense of realistic requirement. If the tester feels that a requirement cannot be realized using existing hardware and software technology, it means that it is unrealistic. In that case, the requirement should either be updated or removed from SRS.

Unambiguous

A requirement should be verified such that it does not provide too many meanings or interpretations. It should not create redundancy in specifications. Each characteristic should be described using a single term, otherwise ambiguity or redundancy may cause bugs in the design phase. The following must be verified:

(a) Every requirement has only one interpretation.

(b) Each characteristic of the final product is described using a single unique term.

Consistent

No specification should contradict or conflict with another. Conflicts produce bugs in the next stages, therefore they must be checked for the following:

(a) Real-world objects conflict, for example, one specification recommends mouse for input, another recommends joystick.

(b) Logical conflict between two specified actions, e.g. one specification requires the function to perform square root, while another specification requires the same function to perform square operation.

(c) Conflicts in terminology should also be verified. For example, at one place, the term *process* is used while at another place, it has been termed as *task* or *module*.

Completeness

The requirements specified in the SRS must be verified for completeness. We must

(a) Verify that all significant requirements such as functionality, performance, design constraints, attribute, or external interfaces are complete.

(b) Check whether responses of every possible input (valid & invalid) to the software have been defined.

(c) Check whether figures and tables have been labeled and referenced completely.

Updation

Requirement specifications are not stable, they may be modified or another requirement may be added later. Therefore, if any updation is there in the SRS, then the updated specifications must be verified.

(a) If the specification is a new one, then all the above mentioned steps and their feasibility should be verified.

(b) If the specification is a change in an already mentioned specification, then we must verify that this change can be implemented in the current design.

Traceability

The traceability of requirements must also be verified such that the origin of each requirement is clear and also whether it facilitates referencing in future development or enhancement documentation. The following two types of traceability must be verified:

Backward traceability Check that each requirement references its source in previous documents.

Forward traceability Check that each requirement has a unique name or reference number in all the documents. Forward traceability assumes more meaning than this, but for the sake of clarity, here it should be understood in the sense that every requirement has been recognized in other documents.

3.4 Verification of High-Level Design

All the requirements mentioned in the SRS document are addressed in this phase and work in the direction of designing the solution. The architecture and design is documented in another document called the *software design document* (SDD).

Like the verification of requirements, the tester is responsible for two parallel activities in this phase as well:

1. The tester verifies the high-level design. Since the system has been decomposed in a number of sub-systems or components, the tester should verify the functionality of these components. Since the system is considered a black box with no low-level details considered here, the stress is also on how the system will interface with the outside world. So all the interfaces and interactions of user/customer (or any person who is interfacing with system) are specified in this phase. The tester verifies that all the components and their interfaces are in tune with requirements of the user. Every requirement in SRS should map the design.
2. The tester also prepares a *Function Test Plan* which is based on the SRS. This plan will be referenced at the time of *Function Testing* (discussed later).

The tester also prepares an *Integration Test Plan* which will be referred at the time of integration testing (discussed later).

3.4.1 How to Verify High-Level Design?

High-level design takes the second place in SDLC, wherein there is a high probability of finding bugs. Therefore, high-level design must be verified as a next step in early testing. Unless the design is specified in a formal way, design cannot be verified. So SDD is referred for design verification. IEEE [19] has provided the standard way of documenting the design in an SDD.

If a bug goes undetected in the high-level design phase, then its cost of fixing increases with every phase. Therefore, verification for high-level design must be done very carefully. This design is divided in three parts (as described by Pressman [7]).

Data Design

It creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data). The structure of data has always been an important part of software design. At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to create high-quality applications.

Architectural Design

It focuses on the representation of the structure of software components, their properties, and interactions.

Interface Design

It creates an effective communication medium between the interfaces of different software modules, interfaces between the software system and any other external entity, and interfaces between a user and the software system. Following a set of interface design principles, the design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

Verification of high-level design will consider all the three designs mentioned above and this will become the basis for effective verification.

Verification of Data Design

The points considered for verification of data design are as follows:

- Check whether the sizes of data structure have been estimated appropriately.
- Check the provisions of overflow in a data structure.
- Check the consistency of data formats with the requirements.
- Check whether data usage is consistent with its declaration.
- Check the relationships among data objects in data dictionary.
- Check the consistency of databases and data warehouses with the requirements specified in SRS.

Verification of Architectural Design

The points considered for the verification of architectural design are:

- Check that every functional requirement in the SRS has been take care of in this design.
- Check whether all exceptions handling conditions have been taken care of.
- Verify the process of transform mapping and transaction mapping, used for the transition from requirement model to architectural design.
- Since architectural design deals with the classification of a system into sub-systems or modules, check the functionality of each module according to the requirements specified.
- Check the inter-dependence and interface between the modules.

- In the modular approach of architectural design, there are two issues with modularity–*Module Coupling and Module Cohesion.* A good design will have low coupling and high cohesion. Testers should verify these factors, otherwise they will affect the reliability and maintainability of the system which are non-functional requirements of the system.

Verification of User-Interface Design

The points to be considered for the verification of user-interface design are:

- Check all the interfaces between modules according to the architecture design.
- Check all the interfaces between software and other non-human producer and consumer of information.
- Check all the interfaces between human and computer.
- Check all the above interfaces for their consistency.
- Check the response time for all the interfaces are within required ranges. It is very essential for the projects related to real-time systems where response time is very crucial.
- For a Help Facility, verify the following:
  - (i) The representation of Help in its desired manner
  - (ii) The user returns to the normal interaction from Help
- For error messages and warnings, verify the following:
  - (i) Whether the message clarifies the problem
  - (ii) Whether the message provides constructive advice for recovering from the error
- For typed command interaction, check the mapping between every menu option and their corresponding commands.

3.5 Verification of Low-level Design

In this verification, low-level design phase is considered. The abstraction level in this phase is low as compared to high-level design. In LLD, a detailed design of modules and data are prepared such that an operational software is ready. For this, SDD is preferred where all the modules and their interfaces are defined. Every operational detail of each module is prepared. The details of each module or unit is prepared in their separate SRS and SDD.

Testers also perform the following parallel activities in this phase:

1. The tester verifies the LLD. The details and logic of each module is verified such that the high-level and low-level abstractions are consistent.
2. The tester also prepares the *Unit Test Plan* which will be referred at the time of *Unit Testing* (discussed later).

3.5.1 How to Verify Low-Level Design?

This is the last pre-coding phase where internal details of each design entity are described. For verification, the SRS and SDD of individual modules are referred to. Some points to be considered are listed below:

- Verify the SRS of each module.
- Verify the SDD of each module.
- In LLD, data structures, interfaces, and algorithms are represented by design notations; verify the consistency of every item with their design notations.

Organizations can build a two-way traceability matrix between the SRS and design (both HLD and LLD) such that at the time of verification of design, each requirement mentioned in the SRS is verified. In other words, the traceability matrix provides a one-to-one mapping between the SRS and the SDD.

3.6 How to Verify Code?

Coding is the process of converting LLD specifications into a specific language. This is the last phase when we get the operational software with the source code. People have the impression that testing starts only after this phase. However, it has been observed in the last chapter that testing starts as soon as the requirement specifications are given and testers perform parallel activities during every phase of SDLC. If we start testing after coding, then there is a possibility that requirement specifications may have bugs and these might have been passed into the design and consequently into the source code. Therefore, the operational software which is ready now, is not reliable and when bugs appear after this phase, they are very expensive to fix.

Since low-level design is converted into source code using some language, there is a possibility of deviation from the LLD. Therefore, the code must also be verified. The points against which the code must be verified are:

- Check that every design specification in HLD and LLD has been coded using traceability matrix.
- Examine the code against a language specification checklist.

- Code verification can be done most efficiently by the developer, as he has prepared the code. He can verify every statement, control structure, loop, and logic such that every possible method of execution is tested. In this way, he verifies the whole module which he has developed. Some points against which the code can be verified are:
  (a) Misunderstood or incorrect arithmetic precedence
  (b) Mixed mode operations
  (c) Incorrect initialization
  (d) Precision inaccuracy
  (e) Incorrect symbolic representation of an expression
  (f) Different data types
  (g) Improper or non-existent loop termination
  (h) Failure to exit

Two kinds of techniques are used to verify the coding: (a) static testing, and (b) dynamic testing.

Static testing techniques As discussed in the previous chapter, this technique does not involve actual execution. It considers only static analysis of the code or some form of conceptual execution of the code.

Dynamic testing techniques It is complementary to the static testing technique. It executes the code on some test data. The developer is the key person in this process who can verify the code of his module by using the dynamic testing technique.

3.6.1 Unit Verification

Verification of coding cannot be done for the whole system. Moreover, the system is divided into modules. Therefore, verification of coding means the verification of code of modules by their developers. This is also known as *unit verification testing*. Listed below are the points to be considered while performing unit verification [7]:

- Interfaces are verified to ensure that information properly flows in and out of the program unit under test.
- The local data structure is verified to maintain data integrity.
- Boundary conditions are checked to verify that the module is working fine on boundaries also.
- All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.
- All error handling paths are tested.

Unit verification is largely *white-box oriented*.

3.7 Validation

As described earlier, validation is a set of activities that ensures the software under consideration has been built right and is traceable to customer requirements. Validation testing is performed after the coding is over.

What is the need for validation? When every step of SDLC has been verified, why do we want to test the product at the end? The reasons are:

- To determine whether the product satisfies the users' requirements, as stated in the requirement specification.
- To determine whether the product's actual behaviour matches the desired behaviour, as described in the functional design specification.
- It is not always certain that all the stages till coding are bug-free. The bugs that are still present in the software after the coding phase need to be uncovered.
- Validation testing provides the last chance to discover bugs, otherwise these bugs will move to the final product released to the customer.
- Validation enhances the quality of software.

3.7.1 Validation Activities

The validation activities are divided into *Validation Test Plan* and *Validation Test Execution* which are described as follows:

Validation Test Plan

It starts as soon as the first output of SDLC, i.e. the SRS, is prepared. In every phase, the tester performs two parallel activities—verification at that phase and the corresponding validation test plan. For preparing a validation test plan, testers must follow the points described below.

- Testers must understand the current SDLC phase.
- Testers must study the relevant documents in the corresponding SDLC phase.
- On the basis of the understanding of SDLC phase and related documents, testers must prepare the related test plans which are used at the time of validation testing. Under test plans, they must prepare a sequence of test cases for validation testing.

The following test plans have been recognized which the testers have already prepared with the incremental progress of SDLC phases:

Acceptance test plan This plan is prepared in the requirement phase according to the acceptance criteria prepared from the user feedback. This plan is used at the time of Acceptance Testing.

System test plan This plan is prepared to verify the objectives specified in the SRS. Here, test cases are designed keeping in view how a complete integrated system will work or behave in different conditions. The plan is used at the time of System Testing.

Function test plan This plan is prepared in the HLD phase. In this plan, test cases are designed such that all the interfaces and every type of functionality can be tested. The plan is used at the time of Function Testing.

Integration test plan This plan is prepared to validate the integration of all the modules such that all their interdependencies are checked. It also validates whether the integration is in conformance to the whole system design. This plan is used at the time of Integration Testing.

Unit test plan This plan is prepared in the LLD phase. It consists of a test plan of every module in the system separately. Unit test plan of every unit or module is designed such that every functionality related to individual unit can be tested. This plan is used at the time of Unit Testing.

Validation Test Execution

Validation test execution can be divided in the following testing activities:

Unit validation testing The testing strategy is to first focus on the smaller building blocks of the full system. One unit or module is the basic building block of the whole software that can be tested for all its interfaces and functionality. Thus, unit testing is a process of testing the individual components of a system. A unit or module must be validated before integrating it with other modules. Unit validation is the first validation activity after the coding of one module is over. The motivation for unit validation as compared to the whole system are as follows:

(a) Since the developer has his attention focused on a smaller building block of the software, i.e. unit or module, it is quite natural to test the unit first.

(b) If the whole software is tested at once, then it is very difficult to trace the bug. Thus, debugging becomes easy with unit testing.

(c) In large-scale projects, a number of modules may be there and each module may be composed of tens of thousands of lines of code. In such a case, it is not possible for a single developer to develop all the modules. There is a team of developers working on separate modules. Sometimes,

some modules are sent to other organizations for development. This requires parallelism in software development. If we did not have the concept of module, this type of parallelism would not have existed. Thus, every module can be developed and tested independently.

Integration testing It is the process of combining and testing multiple components or modules together. The individual tested modules, when combined with other modules, are not tested for their interfaces. Therefore, they may contain bugs in integrated environment. Thus, the intention here is to uncover the bugs that are present when unit tested modules are integrated.

Function testing When the integrated system has been tested, all the specified functions and their external interfaces are tested on the software. Every functionality of the system specified in the functions is tested according to its external specifications. An external specification is a precise description of the software behaviour from the viewpoint of the outside world (e.g. user). Thus, function testing is to explore the bugs related to discrepancies between the actual system behaviour and its functional specifications.

The objective of function test is to measure the quality of the functional (business) components of the system. Tests verify that the system behaves correctly from the user/business perspective and functions according to the requirements, models, or any other design paradigm used to specify the application. The function test must determine if each component or business event:

1. performs in accordance to the specifications,
2. responds correctly to all the conditions that may be presented by incoming events/data,
3. moves the data correctly from one business event to the next (including data stores), and
4. is initiated in the order required to meet the business objectives of the system.

System testing It is different from function testing, as it does not test every function. System testing is actually a series of different tests whose primary purpose is to fully exercise a computer-based system [7]. System testing does not aim to test the specified function, but its intention is to test the whole system on various grounds where bugs may occur. For example, if the software fails in some conditions, how does it recover? Another example is protection from improper penetration, i.e. how secure the whole system is.

Acceptance testing As explained earlier, in the requirement phase of SDLC, acceptance criteria for the system to be developed are mentioned in one contract by the customer. When the system is ready, it can be tested against

the acceptance criteria contracted with the customer. This is called acceptance testing. Thus, acceptance testing can be defined as the process of comparing the final system with the needs of customer as agreed on the acceptance criteria.

Installation testing Once the testing team has given the green signal for producing the software, the software is placed into an operational status where it is installed. The installation testing does not test the system, but it tests the process of making the software system operational.

The installation process must identify the steps required to install the system. For example, it may be required that some files may have to be converted into another format. Or it may be possible that some operating software may be required. Thus, installation testing tests the interface to operating software, related software, and any operating procedures.

All these validation testing activities will be discussed in detail in Chapter 7.

SUMMARY

This chapter illustrates the verification and validation activities through the expanded V-diagram. As a part of the strategy of early testing, verification is an important activity for a software tester. This is why verification activity is performed at every stage of SDLC. It reduces the chances of propagation of bugs to the later stages of SDLC.

At the same time, a software tester plans for validation testing while performing verification. These test plans are used when we start validation after a module coding is over. Validation testing activities are performed till the software is installed in an operational status. In this way, both activities assume their importance as a part of early testing as well as a fully tested software product. Thus, verification and validation become the basis for any type of software testing.

Let us review the important concepts described in this chapter:

- Verification is performed at every stage of SDLC to uncover more and more bugs during the earlier stages
- Verification activities are: Verification of requirements and objectives, Verification of high-level design, Verification of low-level design, Verification of coding (Unit Verification)
- After the completion of every SDLC phase, the tester prepares a test plan corresponding to a particular validation testing. This means, the tester does two tasks in parallel: i) Verification of every stage ii) Planning for Validation
- The test plans and corresponding validation testing are shown below:

SDLC Phase	Test Plan	Validation Testing
Requirement Gathering	Acceptance Test Plan	Acceptance Testing
Requirement Specification/ Objectives	System Test Plan	System Testing
Functional Design/ High-Level Design	Function and Integration Test Plan	Function and Integration Testing
Internal Design/Low-level Design	Unit Test Plan	Unit Testing

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Which of the following is true?
   (a) Testing is performed after coding.
   (b) Testing is performed after the integration of modules.
   (c) Testing starts as soon as the SRS is prepared.
   (d) None of the above
2. V&V diagram includes
   (a) Verification only
   (b) Validation only
   (c) Both verification and validation
   (d) None of the above
3. Which test plan is made corresponding to requirement gathering?
   (a) Acceptance test plan
   (b) System test plan
   (c) Function & Integration test plan
   (d) Unit test plan
4. Which test plan is made corresponding to requirement specifications?
   (a) Acceptance test plan
   (b) System test plan
   (c) Function & Integration test plan
   (d) Unit test plan
5. Which test plan is made corresponding to HLD?
   (a) Acceptance test plan
   (b) System test plan
   (c) Function & Integration test plan
   (d) Unit test plan
6. Which test plan is made corresponding to LLD?
   (a) Acceptance test plan
   (b) System test plan
   (c) Function & Integration test plan
   (d) Unit test plan
7. In the V model of testing, the scope of verification from top to bottom
   (a) Increases
   (b) Decreases
   (c) Remains the same
   (d) None of the above

8. For the verification of requirements, you must check for
   (a) Correctness
   (b) Ambiguity
   (c) Completeness
   (d) All of the above
9. For the verification of high-level design, you must check
   (a) Data design
   (b) Architectural design
   (c) User interface design
   (d) All of the above
10. For verifying architectural design, which is true?
   (a) Check that every functional requirement in the SRS has been take care of in this design.
   (b) Check whether all exceptions handling conditions have been taken care of.
   (c) Check the inter-dependence and interface between the modules.
   (d) All of the above
11. For verifying data design, you must check
   (a) Sizes of data structure
   (b) Size of module
   (c) Overflow in a data structure
   (d) All of the above
12. What is the relation between static and dynamic testing technique?
   (a) Mutually exclusive
   (b) Complementary
   (c) Independent
   (d) None
13. What is the proper sequence of various testing?
   (a) Function testing, integration testing, system testing, acceptance testing, unit testing
   (b) Unit testing, integration testing, function testing, system testing, acceptance testing
   (c) Unit testing, integration testing, system testing, function testing, acceptance testing
   (d) None

Review Questions

1. What are the activities performed by a tester at the time of development of a project?
2. What is a V-diagram? What are its benefits?
3. What is the need for verification?
4. What is the need for validation?
5. What is the role of test plans in a V&V diagram?

6. What are the adverse effects on a project, if verification is not performed?
7. Discuss the goals of verification in a project.
8. In a project, only the SRS and SDD are prepared. Can you start testing at this stage?
9. How would you verify the ambiguous requirements in a project and in which stage of SDLC?
10. What do you mean by backward and forward traceability?
11. Develop a checklist of the common coding errors for the language in which you develop your programs.
12. Develop a table describing each phase and corresponding test plan documents related to the project on which you are working.
13. Evaluate the benefits of verification (early testing) in a project.
14. Evaluate the benefits of validation in a project.
15. Suppose you are a member of development team of a Database maintenance project. How would you verify transform mapping and transaction mapping in this project?
16. What is the difference between unit verification and unit validation?
17. What is the difference between function testing and system testing?
18. What is the difference between system testing and acceptance testing?
19. 'V&V diagram is basis for every type of testing?' Comment on this statement.
20. Take a project and perform verification and validation activities as described in this chapter.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Part 2 Testing Techniques

CHAPTERS

Chapter 4:
Dynamic Testing: Black-Box Testing Techniques

Chapter 5:
Dynamic Testing: White-Box Testing Techniques

Chapter 6:
Static Testing

Chapter 7:
Validation Activities

Chapter 8:
Regression Testing

In Part I, we have discussed the fundamentals of effective software testing and how to develop a testing methodology. After devising a testing strategy, we need various testing techniques so that we can design test cases in order to perform testing. There are two views of testing techniques: one view is to categorize based on verification and validation; another is based on static and dynamic testing. We have seen the general strategy of performing verification and validation in Part I. The techniques for verification and validation have been discussed in this part.

Static testing largely maps to verification and dynamic testing to validation. Static testing is performed without executing the code, and dynamic testing is performed with the execution of code. Dynamic testing techniques, namely black-box and white-box techniques, are very popular. We have tried to include every possible method under these categories.

Regression testing is a big problem in software testing. Whenever we make some modifications, we need to execute all the test cases designed earlier as well as some new cases to check the modifications and whether these changes have affected other parts of the software. It becomes a problem as the test suite becomes too large to test. Regression testing is a hot topic for researchers. We have included this testing in techniques so as to define it properly and seek some techniques to deal with it.

This part will make ground for the following concepts:

- Dynamic testing techniques
- Static testing techniques
- Validation testing techniques
- Regression testing and techniques

The following table lists the techniques grouped under their respective categories.

Testing Category	Techniques
Dynamic testing: Black-Box	Boundary value analysis, Equivalence class partitioning, State table-based testing, Decision table-based testing, Cause-effect graphing technique, Error guessing.
Dynamic testing: White-Box	Basis path testing, Graph matrices, Loop testing, Data flow testing, Mutation testing.
Static testing	Inspection, Walkthrough, Reviews.
Validation testing	Unit testing, Integration testing, Function testing, System testing, Acceptance testing, Installation testing.
Regression testing	Selective retest technique, Test prioritization.

Chapter

4

Dynamic Testing: Black-Box Testing Techniques

Black-box technique is one of the major techniques in dynamic testing for designing effective test cases. This technique considers only the functional requirements of the software or module. In other words, the structure or logic of the software is not considered. Therefore, this is also known as *functional testing*. The software system is considered as a black box, taking no notice of its internal structure, so it is also called as *black-box* testing technique.

It is obvious that in black-box technique, test cases are designed based on functional specifications. Input test data is given to the system, which is a black box to the tester, and results are checked against expected outputs after executing the software, as shown in Fig. 4.1.

OBJECTIVES

After reading this chapter, you should be able to understand:

- Black-box testing ignores the structural details of the software
- Test case designing using black-box techniques
- Boundary value analysis method
- Equivalence class testing method
- State table-based testing method
- Decision table-based testing method
- Cause-effect graphing method
- Error guessing

Figure 4.1 Black-box testing

Black-box testing attempts to find errors in the following categories:

- To test the modules independently
- To test the functional validity of the software so that incorrect or missing functions can be recognized
- To look for interface errors
- To test the system behaviour and check its performance
- To test the maximum load or stress on the system
- To test the software such that the user/customer accepts the system within defined acceptable limits

There are various methods to test a software product using black-box techniques. One method chooses the boundary values of the variables, another makes equivalence classes so that only one test case in that class is chosen and executed. Some methods use the state diagrams of the system to form the black-box test cases, while a few other methods use table structure to organize the test cases. It does not mean that one can pick any of these methods for testing the software. Sometimes a combination of methods is employed for rigorous testing. The objective of any test case is to have maximum coverage and capability to discover more and more errors.

4.1 Boundary Value Analysis (BVA)

An effective test case design requires test cases to be designed such that they maximize the probability of finding errors. BVA technique addresses this issue. With the experience of testing team, it has been observed that test cases designed with boundary input values have a high chance to find errors. It means that most of the failures crop up due to boundary values.

BVA is considered a technique that uncovers the bugs at the boundary of input values. Here, boundary means the maximum or minimum value taken by the input domain. For example, if *A* is an integer between 10 and 255, then boundary checking can be on 10(9,10,11) and on 255(256,255,254). Similarly, *B* is another integer variable between 10 and 100, then boundary checking can be on 10(9,10,11) and 100(99,100,101), as shown in Fig. 4.2.

Figure 4.2 Boundary value analysis

BVA offers several methods to design test cases as discussed in the following sections.

4.1.1 Boundary Value Checking (BVC)

In this method, the test cases are designed by holding one variable at its extreme value and other variables at their nominal values in the input domain.

The variable at its extreme value can be selected at:

(a) Minimum value (Min)

(b) Value just above the minimum value (Min^+)

(c) Maximum value (Max)

(d) Value just below the maximum value (Max^-)

Let us take the example of two variables, *A* and *B*. If we consider all the above combinations with nominal values, then following test cases (see Fig. 4.3) can be designed:

1. A_{nom}, B_{min}
2. A_{nom}, B_{min+}
3. A_{nom}, B_{max}
4. A_{nom}, B_{max-}
5. A_{min}, B_{nom}
6. A_{min+}, B_{nom}
7. A_{max}, B_{nom}
8. A_{max-}, B_{nom}
9. A_{nom}, B_{nom}

Figure 4.3 Boundary value checking

It can be generalized that for n variables in a module, $4n + 1$ test cases can be designed with boundary value checking method.

4.1.2 Robustness Testing Method

The idea of BVC can be extended such that boundary values are exceeded as:

- A value just greater than the Maximum value (Max^+)
- A value just less than Minimum value (Min^-)

When test cases are designed considering the above points in addition to BVC, it is called *robustness testing*.

Let us take the previous example again. Add the following test cases to the list of 9 test cases designed in BVC:

10. A_{max+}, B_{nom}
11. A_{min-}, B_{nom}
12. A_{nom}, B_{max+}
13. A_{nom}, B_{min-}

It can be generalized that for n input variables in a module, $6n + 1$ test cases can be designed with robustness testing.

4.1.3 Worst-Case Testing Method

We can again extend the concept of BVC by assuming more than one variable on the boundary. It is called *worst-case testing method.*

Again, take the previous example of two variables, A and B. We can add the following test cases to the list of 9 test cases designed in BVC as:

10. A_{min}, B_{min}
11. A_{min+}, B_{min}
12. A_{min}, B_{min+}
13. A_{min+}, B_{min+}
14. A_{max}, B_{min}
15. A_{max-}, B_{min}
16. A_{max}, B_{min+}
17. A_{max-}, B_{min+}
18. A_{min}, B_{max}
19. A_{min+}, B_{max}
20. A_{min}, B_{max-}
21. A_{min+}, B_{max-}
22. A_{max}, B_{max}
23. A_{max-}, B_{max}
24. A_{max}, B_{max-}
25. A_{max-}, B_{max-}

It can be generalized that for n input variables in a module, 5^n test cases can be designed with worst-case testing.

BVA is applicable when the module to be tested is a function of several independent variables. This method becomes important for physical quantities where boundary condition checking is crucial. For example, systems having requirements of minimum and maximum temperature, pressure or speed, etc. However, it is not useful for Boolean variables.

Example 4.1

A program reads an integer number within the range [1,100] and determines whether it is a prime number or not. Design test cases for this program using BVC, robust testing, and worst-case testing methods.

Solution

(a) **Test cases using BVC** Since there is one variable, the total number of test cases will be $4n + 1 = 5$.

In our example, the set of minimum and maximum values is shown below:

Min value = 1
Min^+ value = 2
Max value = 100
Max^- value = 99
Nominal value = 50–55

Using these values, test cases can be designed as shown below:

Test Case ID	Integer Variable	Expected Output
1	1	Not a prime number
2	2	Prime number
3	100	Not a prime number
4	99	Not a prime number
5	53	Prime number

(b) **Test cases using robust testing** Since there is one variable, the total number of test cases will be $6n + 1 = 7$. The set of boundary values is shown below:

Min value = 1
Min^- value = 0
Min^+ value = 2
Max value = 100
Max^- value = 99
Max^+ value = 101
Nominal value = 50–55

Using these values, test cases can be designed as shown below:

Test Case ID	Integer Variable	Expected Output
1	0	Invalid input
2	1	Not a prime number
3	2	Prime number
4	100	Not a prime number
5	99	Not a prime number
6	101	Invalid input
7	53	Prime number

(c) **Test cases using worst-case testing** Since there is one variable, the total number of test cases will be $5^n = 5$. Therefore, the number of test cases will be same as BVC.

Example 4.2

A program computes a^b where a lies in the range [1,10] and b within [1,5]. Design test cases for this program using BVC, robust testing, and worst-case testing methods.

Solution

(a) **Test cases using BVC** Since there are two variables, a and b, the total number of test cases will be $4n + 1 = 9$. The set of boundary values is shown below:

	a	b
Min value	1	1
Min^+ value	2	2
Max value	10	5
Max^- value	9	4
Nominal value	5	3

Using these values, test cases can be designed as shown below:

Test Case ID	a	b	Expected Output
1	1	3	1
2	2	3	8
3	10	3	1000
4	9	3	729
5	5	1	5
6	5	2	25
7	5	4	625
8	5	5	3125
9	5	3	125

(b) **Test cases using robust testing** Since there are two variables, a and b, the total number of test cases will be $6n + 1 = 13$. The set of boundary values is shown below:

	a	b
Min value	1	1
Min^- value	0	0
Min^+ value	2	2
Max value	10	5
Max^+ value	11	6
Max^- value	9	4
Nominal value	5	3

Using these values, test cases can be designed as shown below:

Test Case ID	a	b	Expected output
1	0	3	Invalid input
2	1	3	1
3	2	3	8
4	10	3	1000
5	11	3	Invalid input
6	9	3	729
7	5	0	Invalid input
8	5	1	5
9	5	2	25
10	5	4	625
11	5	5	3125
12	5	6	Invalid input
13	5	3	125

(c) **Test cases using worst-case testing** Since there are two variables, a and b, the total number of test cases will be $5^n = 25$.

The set of boundary values is shown below:

	a	b
Min value	1	1
Min^+ value	2	2
Max value	10	5
Max^- value	9	4
Nominal value	5	3

There may be more than one variable at extreme values in this case. Therefore, test cases can be designed as shown below:

Test Case ID	a	b	Expected Output
1	1	1	1
2	1	2	1
3	1	3	3
4	1	4	1
5	1	5	1
6	2	1	2
7	2	2	4
8	2	3	8
9	2	4	16
10	2	5	32
11	5	1	5
12	5	2	25
13	5	3	125
14	5	4	625
15	5	5	3125
16	9	1	9
17	9	2	81
18	9	3	729
19	9	4	6561
20	9	5	59049
21	10	1	10
22	10	2	100
23	10	3	1000
24	10	4	10000
25	10	5	100000

Example 4.3

A program reads three numbers, *A*, *B*, and *C*, within the range [1, 50] and prints the largest number. Design test cases for this program using BVC, robust testing, and worst-case testing methods.

Solution

(a) **Test cases using BVC** Since there are three variables, *A*, *B*, and *C*, the total number of test cases will be $4n + 1 = 13$. The set of boundary values is shown below:

Min value = 1
Min^+ value = 2
Max value = 50
Max^- value = 49
Nominal value = 25–30

Using these values, test cases can be designed as shown below:

Test Case ID	A	B	C	Expected Output
1	1	25	27	C is largest
2	2	25	28	C is largest
3	49	25	25	B and C are largest
4	50	25	29	A is largest
5	25	1	30	C is largest
6	25	2	26	C is largest
7	25	49	27	B is largest
8	25	50	28	B is largest
9	25	28	1	B is largest
10	25	27	2	B is largest
11	25	26	49	C is largest
12	25	26	50	C is largest
13	25	25	25	Three are equal

(b) **Test cases using robust testing** Since there are three variables, *A*, *B*, and *C*, the total number of test cases will be $6n + 1 = 19$.

The set of boundary values is shown below:

Min value = 1
Min^- value = 0
Min^+ value = 2
Max value = 50
Max^+ value = 51
Max^- value = 49
Nominal value = 25–30

Using these values, test cases can be designed as shown below:

Test Case ID	A	B	C	Expected Output
1	0	25	27	Invalid input
2	1	25	27	C is largest
3	2	25	28	C is largest
4	49	25	25	B and C are largest
5	50	25	29	A is largest
6	51	27	25	Invalid input
7	25	0	26	Invalid input
8	25	1	30	C is largest
9	25	2	26	C is largest
10	25	49	27	B is largest
11	25	50	28	B is largest
12	26	51	25	Invalid input
13	25	25	0	Invalid input
14	25	28	1	B is largest
15	25	27	2	B is largest
16	25	26	49	C is largest
17	25	26	50	C is largest
18	25	29	51	Invalid input
19	25	25	25	Three are equal

(c) **Test cases using worst-case testing** Since there are three variables, A, B, and C, the total number of test cases will be $5^n = 125$.

The set of boundary values is shown below:

Min value = 1
Min^+ value = 2
Max value = 50
Max^- value = 49
Nominal value = 25–30

There may be more than one variable at extreme values in this case. Therefore, test cases can be design as shown below:

Test Case ID	A	B	C	Expected Output
1	1	1	1	All three are equal
2	1	1	2	C is greatest
3	1	1	25	C is greatest
4	1	1	49	C is greatest
5	1	1	50	C is greatest
6	1	2	1	B is greatest
7	1	2	2	B and C
8	1	2	25	C is greatest
9	1	2	49	C is greatest
10	1	2	50	C is greatest
11	1	25	1	B is greatest
12	1	27	2	B is greatest
13	1	26	25	B is greatest
14	1	25	49	B is greatest
15	1	27	50	C is greatest
16	1	49	1	B is greatest
17	1	49	2	B is greatest
18	1	49	25	B is greatest
19	1	49	49	B and C
20	1	49	50	C is greatest
21	1	50	1	B is greatest
22	1	50	2	B is greatest
23	1	50	25	B is greatest
24	1	50	49	B is greatest
25	1	50	50	B and C
26	2	1	1	A is largest
27	2	1	2	A and C
28	2	1	25	C is greatest

29	2	1	49	C is greatest
30	2	1	50	C is greatest
31	2	2	1	A and B
32	2	2	2	All three are equal
33	2	2	25	C is greatest
34	2	2	49	C is greatest
35	2	2	50	C is greatest
36	2	25	1	B is greatest
37	2	27	2	B is greatest
38	2	28	25	B is greatest
39	2	26	49	C is greatest
40	2	28	50	C is greatest
41	2	49	1	B is greatest
42	2	49	2	B is greatest
43	2	49	25	B is greatest
44	2	49	49	B and C
45	2	49	50	C is greatest
46	2	50	1	B is greatest
47	2	50	2	B is greatest
48	2	50	25	B is greatest
49	2	50	49	B is greatest
50	2	50	50	B and C
51	25	1	1	A is greatest
52	25	1	2	A is greatest
53	25	1	25	A and C
54	25	1	49	C is greatest
55	25	1	50	C is greatest
56	25	2	1	A is greatest
57	25	2	2	A is greatest
58	25	2	25	A and C
59	25	2	49	C is greatest
60	25	2	50	C is greatest
61	25	27	1	B is greatest
62	25	26	2	B is greatest
63	25	25	25	All three are equal
64	25	28	49	C is greatest
65	25	29	50	C is greatest
66	25	49	1	B is greatest
67	25	49	2	B is greatest
68	25	49	25	B is greatest

69	25	49	49	B is greatest
70	25	49	50	C is greatest
71	25	50	1	B is greatest
72	25	50	2	B is greatest
73	25	50	25	B is greatest
74	25	50	49	B is greatest
75	25	50	50	B is greatest
76	49	1	1	A is greatest
77	49	1	2	A is greatest
78	49	1	25	A is greatest
79	49	1	49	A and C
80	49	1	50	C is greatest
81	49	2	1	A is greatest
82	49	2	2	A is greatest
83	49	2	25	A is greatest
84	49	2	49	A and C
85	49	2	50	C is greatest
86	49	25	1	A is greatest
87	49	29	2	A is greatest
88	49	25	25	A is greatest
89	49	27	49	A and C
90	49	28	50	C is greatest
91	49	49	1	A and B
92	49	49	2	A and B
93	49	49	25	A and B
94	49	49	49	All three are equal
95	49	49	50	C is greatest
96	49	50	1	B is greatest
97	49	50	2	B is greatest
98	49	50	25	B is greatest
99	49	50	49	B is greatest
100	49	50	50	B and C
101	50	1	1	A is greatest
102	50	1	2	A is greatest
103	50	1	25	A is greatest
104	50	1	49	A is greatest
105	50	1	50	A and C
106	50	2	1	A is greatest
107	50	2	2	A is greatest
108	50	2	25	A is greatest

109	50	2	49	A is greatest
110	50	2	50	A and C
111	50	26	1	A is greatest
112	50	25	2	A is greatest
113	50	27	25	A is greatest
114	50	29	49	A is greatest
115	50	30	50	A and C
116	50	49	1	A is greatest
117	50	49	2	A is greatest
118	50	49	26	A is greatest
119	50	49	49	A is greatest
120	50	49	50	A and C
121	50	50	1	A and B
122	50	50	2	A and B
123	50	50	26	A and B
124	50	50	49	A and B
125	50	50	50	All three are equal

Example 4.4

A program determines the next date in the calendar. Its input is entered in the form of <ddmmyyyy> with the following range:

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

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

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

Its output would be the next date or it will display 'invalid date.' Design test cases for this program using BVC, robust testing, and worst-case testing methods.

Solution

(a) **Test cases using BVC** Since there are three variables, month, day, and year, the total number of test cases will be $4n + 1 = 13$. The set of boundary values is shown below:

	Month	Day	Year
Min value	1	1	1900
Min^+ value	2	2	1901
Max value	12	31	2025
Max^- value	11	30	2024
Nominal value	6	15	1962

Using these values, test cases can be designed as shown below:

Test Case ID	Month	Day	Year	Expected Output
1	1	15	1962	16-1-1962
2	2	15	1962	16-2-1962
3	11	15	1962	16-11-1962
4	12	15	1962	16-12-1962
5	6	1	1962	2-6-1962
6	6	2	1962	3-6-1962
7	6	30	1962	1-7-1962
8	6	31	1962	Invalid input
9	6	15	1900	16-6-1900
10	6	15	1901	16-6-1901
11	6	15	2024	16-6-2024
12	6	15	2025	16-6-2025
13	6	15	1962	16-6-1962

(b) **Test cases using robust testing** The total number of test cases will be $6n + 1 = 19$. The set of boundary values is shown below:

	Month	Day	Year
Min^- value	0	0	1899
Min value	1	1	1900
Min^+ value	2	2	1901
Max value	12	31	2025
Max^- value	11	30	2024
Max^+ value	13	32	2026
Nominal value	6	15	1962

Using these values, test cases can be designed as shown below:

Test Case ID	Month	Day	Year	Expected Output
1	0	15	1962	Invalid date
2	1	15	1962	16-1-1962
3	2	15	1962	16-2-1962
4	11	15	1962	16-11-1962
5	12	15	1962	16-12-1962
6	13	15	1962	Invalid date
7	6	0	1962	Invalid date
8	6	1	1962	2-6-1962
9	6	2	1962	3-6-1962
10	6	30	1962	1-7-1962

11	6	31	1962	Invalid input
12	6	32	1962	Invalid date
13	6	15	1899	Invalid date
14	6	15	1900	16-6-1900
15	6	15	1901	16-6-1901
16	6	15	2024	16-6-2024
17	6	15	2025	16-6-2025
18	6	15	2026	Invalid date
19	6	15	1962	16-6-1962

(c) **Test cases using worst-case testing** The total number of test cases will be $5^n = 125$. The set of boundary values is shown below:

	Month	Day	Year
Min value	1	1	1900
Min^+ value	2	2	1901
Max value	12	31	2025
Max^- value	11	30	2024
Nominal value	6	15	1962

Using these values, test cases can be designed as shown below:

Test Case ID	Month	Day	Year	Expected Output
1	1	1	1900	2-1-1900
2	1	1	1901	2-1-1901
3	1	1	1962	2-1-1962
4	1	1	2024	2-1-2024
5	1	1	2025	2-1-2025
6	1	2	1900	3-1-1900
7	1	2	1901	3-1-1901
8	1	2	1962	3-1-1962
9	1	2	2024	3-1-2024
10	1	2	2025	3-1-2025
11	1	15	1900	16-1-1900
12	1	15	1901	16-1-1901
13	1	15	1962	16-1-1962
14	1	15	2024	16-1-2024
15	1	15	2025	16-1-2025
16	1	30	1900	31-1-1900

17	1	30	1901	31-1-1901
18	1	30	1962	31-1-1962
19	1	30	2024	31-1-2024
20	1	30	2025	31-1-2025
21	1	31	1900	1-2-1900
22	1	31	1901	1-2-1901
23	1	31	1962	1-2-1962
24	1	31	2024	1-2-2024
25	1	31	2025	1-2-2025
26	2	1	1900	2-2-1900
27	2	1	1901	2-2-1901
28	2	1	1962	2-2-1962
29	2	1	2024	2-1-2024
30	2	1	2025	2-2-2025
31	2	2	1900	3-2-1900
32	2	2	1901	3-2-1901
33	2	2	1962	3-2-1962
34	2	2	2024	3-2-2024
35	2	2	2025	3-2-2025
36	2	15	1900	16-2-1900
37	2	15	1901	16-2-1901
38	2	15	1962	16-2-1962
39	2	15	2024	16-2-2024
40	2	15	2025	16-2-2025
41	2	30	1900	Invalid date
42	2	30	1901	Invalid date
43	2	30	1962	Invalid date
44	2	30	2024	Invalid date
45	2	30	2025	Invalid date
46	2	31	1900	Invalid date
47	2	31	1901	Invalid date
48	2	31	1962	Invalid date
49	2	31	2024	Invalid date
50	2	31	2025	Invalid date
51	6	1	1900	2-6-1900
52	6	1	1901	2-6-1901
53	6	1	1962	2-6-1962
54	6	1	2024	2-6-2024

55	6	1	2025	2-6-2025
56	6	2	1900	3-6-1900
57	6	2	1901	3-6-1901
58	6	2	1962	3-6-1962
59	6	2	2024	3-6-2024
60	6	2	2025	3-6-2025
61	6	15	1900	16-6-1900
62	6	15	1901	16-6-1901
63	6	15	1962	16-6-1962
64	6	15	2024	16-6-2024
65	6	15	2025	16-6-2025
66	6	30	1900	1-7-1900
67	6	30	1901	1-7-1901
68	6	30	1962	1-7-1962
69	6	30	2024	1-7-2024
70	6	30	2025	1-7-2025
71	6	31	1900	Invalid date
72	6	31	1901	Invalid date
73	6	31	1962	Invalid date
74	6	31	2024	Invalid date
75	6	31	2025	Invalid date
76	11	1	1900	2-11-1900
77	11	1	1901	2-11-1901
78	11	1	1962	2-11-1962
79	11	1	2024	2-11-2024
80	11	1	2025	2-11-2025
81	11	2	1900	3-11-1900
82	11	2	1901	3-11-1901
83	11	2	1962	3-11-1962
84	11	2	2024	3-11-2024
85	11	2	2025	3-11-2025
86	11	15	1900	16-11-1900
87	11	15	1901	16-11-1901
88	11	15	1962	16-11-1962
89	11	15	2024	16-11-2024
90	11	15	2025	16-11-2025
91	11	30	1900	1-12-1900
92	11	30	1901	1-12-1901

93	11	30	1962	1-12-1962
94	11	30	2024	1-12-2024
95	11	30	2025	1-12-2025
96	11	31	1900	Invalid date
97	11	31	1901	Invalid date
98	11	31	1962	Invalid date
99	11	31	2024	Invalid date
100	11	31	2025	Invalid date
101	12	1	1900	2-12-1900
102	12	1	1901	2-12-1901
103	12	1	1962	2-12-1962
104	12	1	2024	2-12-2024
105	12	1	2025	2-12-2025
106	12	2	1900	3-12-1900
107	12	2	1901	3-12-1901
108	12	2	1962	3-12-1962
109	12	2	2024	3-12-2024
110	12	2	2025	3-12-2025
111	12	15	1900	16-12-1900
112	12	15	1901	16-12-1901
113	12	15	1962	16-12-1962
114	12	15	2024	16-12-2024
115	12	15	2025	16-12-2025
116	12	30	1900	31-12-1900
117	12	30	1901	31-12-1901
118	12	30	1962	31-12-1962
119	12	30	2024	31-12-2024
120	12	30	2025	31-12-2025
121	12	31	1900	1-1-1901
122	12	31	1901	1-1-1902
123	12	31	1962	1-1-1963
124	12	31	2024	1-1-2025
125	12	31	2025	1-1-2026

4.2 Equivalence Class Testing

We know that the input domain for testing is too large to test every input. So we can divide or partition the input domain based on a common feature or

a class of data. Equivalence partitioning is a method for deriving test cases wherein classes of input conditions called *equivalence classes* are identified such that each member of the class causes the same kind of processing and output to occur. Thus, instead of testing every input, only one test case from each partitioned class can be executed. It means only one test case in the equivalence class will be sufficient to find errors. This test case will have a representative value of a class which is equivalent to a test case containing any other value in the same class. If one test case in an equivalence class detects a bug, all other test cases in that class have the same probability of finding bugs. Therefore, instead of taking every value in one domain, only one test case is chosen from one class. In this way, testing covers the whole input domain, thereby reduces the total number of test cases. In fact, it is an attempt to get a good *hit rate* to find maximum errors with the smallest number of test cases.

Equivalence partitioning method for designing test cases has the following goals:

Completeness Without executing all the test cases, we strive to touch the completeness of testing domain.

Non-redundancy When the test cases are executed having inputs from the same class, then there is redundancy in executing the test cases. Time and resources are wasted in executing these redundant test cases, as they explore the same type of bug. Thus, the goal of equivalence partitioning method is to reduce these redundant test cases.

To use equivalence partitioning, one needs to perform two steps:

1. Identify equivalence classes
2. Design test cases

4.2.1 Identification of Equivalent Classes

How do we partition the whole input domain? Different equivalence classes are formed by grouping inputs for which the behaviour pattern of the module is similar. The rationale of forming equivalence classes like this is the assumption that if the specifications require exactly the same behaviour for each element in a class of values, then the program is likely to be constructed such that it either succeeds or fails for each value in that class. For example, the specifications of a module that determines the absolute value for integers specify different behaviour patterns for positive and negative integers. In this case, we will form two classes: one consisting of positive integers and another consisting of negative integers [14].

Two types of classes can always be identified as discussed below:

Valid equivalence classes These classes consider valid inputs to the program.

Invalid equivalence classes One must not be restricted to valid inputs only. We should also consider invalid inputs that will generate error conditions or unexpected behaviour of the program, as shown in Fig. 4.4.

Figure 4.4 Equivalence classes

There are no well-defined rules for identifying equivalence classes, as it is a heuristic process. However, some guidelines are defined for forming equivalence classes:

- If there is no reason to believe that the entire range of an input will be treated in the same manner, then the range should be split into two or more equivalence classes.
- If a program handles each valid input differently, then define one valid equivalence class per valid input.
- Boundary value analysis can help in identifying the classes. For example, for an input condition, say $0 \leq a \leq 100$, one valid equivalent class can be formed from the valid range of a. And with BVA, two invalid classes that cross the minimum and maximum values can be identified, i.e. $a < 0$ and $a > 100$.
- If an input variable can identify more than one category, then for each category, we can make equivalent classes. For example, if the input is a character, then it can be an alphabet, a number, or a special character. So we can make three valid classes for this input and one invalid class.
- If the requirements state that the number of items input by the system at some point must lie within a certain range, specify one valid class where the number of inputs is within the valid range, one invalid class where there are very few inputs, and one invalid class where there are too many inputs. For example, specifications state that a maximum of 4 purchase orders can be registered against a product. The equivalence classes are: the valid equivalence class ($1 \leq$ no. of purchase orders ≤ 4), the invalid class (no. of purchase orders > 4), and the invalid class (no. of purchase orders < 1).

- If an input condition specifies a 'must be' situation (e.g., 'first character of the identifier must be a letter'), identify a valid equivalence class (it is a letter) and an invalid equivalence class (it is not a letter).
- Equivalence classes can be of the output desired in the program. For an output equivalence class, the goal is to generate test cases such that the output for that test case lies in the output equivalence class. Determining test cases for output classes may be more difficult, but output classes have been found to reveal errors that are not revealed by just considering the input classes.
- Look for membership of an input condition in a set or group and identify valid (within the set) and invalid (outside the set) classes. For example, if the requirements state that a valid province code is ON, QU, and NB, then identify: the valid class (code is one of ON, QU, NB) and the invalid class (code is not one of ON, QU, NB).
- If the requirements state that a particular input item match a set of values and each case will be dealt with differently, identify a valid equivalence class for each element and only one invalid class for values outside the set. For example, if a discount code must be input as *P* for a preferred customer, *R* for a standard reduced rate, or *N* for none, and if each case is treated differently, identify: the valid class code = *P*, the valid class code = *R*, the valid class code = *N*, the invalid class code is not one of *P*, *R*, *N*.
- If an element of an equivalence class will be handled differently than the others, divide the equivalence class to create an equivalence class with only these elements and an equivalence class with none of these elements. For example, a bank account balance may be from 0 to Rs 10 lakh and balances of Rs 1,000 or more are not subject to service charges. Identify: the valid class: (0 ≤ balance < Rs 1,000), i.e. balance is between 0 and Rs 1,000 – not including Rs 1,000; the valid class: (Rs 1,000 ≤ balance ≤ Rs 10 lakh, i.e. balance is between Rs 1,000 and Rs 10 lakh inclusive the invalid class: (balance < 0) the invalid class: (balance > Rs 10 lakh).

4.2.2 Identifying the Test Cases

A few guidelines are given below to identify test cases through generated equivalence classes:

- Assign a unique identification number to each equivalence class.
- Write a new test case covering as many of the uncovered valid equivalence classes as possible, until all valid equivalence classes have been covered by test cases.

- Write a test case that covers one, and only one, of the uncovered invalid equivalence classes, until all invalid equivalence classes have been covered by test cases. The reason that invalid cases are covered by individual test cases is that certain erroneous-input checks mask or supersede other erroneous-input checks. For instance, if the specification states 'Enter type of toys (Automatic, Mechanical, Soft toy) and amount (1–10000)', the test case [ABC 0] expresses two error (invalid inputs) conditions (invalid toy type and invalid amount) will not demonstrate the invalid amount test case, hence the program may produce an output 'ABC is unknown toy type' and not bother to examine the remainder of the input.

Remember that there may be many possible solutions for one problem in this technique, depending on the criteria chosen for partitioning the test domain.

Example 4.5

A program reads three numbers, *A*, *B*, and *C*, with a range [1, 50] and prints the largest number. Design test cases for this program using equivalence class testing technique.

Solution

1. First we partition the domain of input as valid input values and invalid values, getting the following classes:

$$I_1 = \{<A,B,C> : 1 \le A \le 50\}$$
$$I_2 = \{<A,B,C> : 1 \le B \le 50\}$$
$$I_3 = \{<A,B,C> : 1 \le C \le 50\}$$
$$I_4 = \{<A,B,C> : A < 1\}$$
$$I_5 = \{<A,B,C> : A > 50\}$$
$$I_6 = \{<A,B,C> : B < 1\}$$
$$I_7 = \{<A,B,C> : B > 50\}$$
$$I_8 = \{<A,B,C> : C < 1\}$$
$$I_9 = \{<A,B,C> : C > 50\}$$

Now the test cases can be designed from the above derived classes, taking one test case from each class such that the test case covers maximum valid input classes, and separate test cases for each invalid class.

The test cases are shown below:

Test case ID	A	B	C	Expected result	Classes covered by the test case
1	13	25	36	C is greatest	I_1, I_2, I_3
2	0	13	45	Invalid input	I_4
3	51	34	17	Invalid input	I_5
4	29	0	18	Invalid input	I_6
5	36	53	32	Invalid input	I_7
6	27	42	0	Invalid input	I_8
7	33	21	51	Invalid input	I_9

2. We can derive another set of equivalence classes based on some possibilities for three integers, A, B, and C. These are given below:

$$I_1 = \{<A,B,C> : A > B,\ A > C\}$$
$$I_2 = \{<A,B,C> : B > A,\ B > C\}$$
$$I_3 = \{<A,B,C> : C > A,\ C > B\}$$
$$I_4 = \{<A,B,C> : A = B,\ A \neq C\}$$
$$I_5 = \{<A,B,C> : B = C,\ A \neq B\}$$
$$I_6 = \{<A,B,C> : A = C,\ C \neq B\}$$
$$I_7 = \{<A,B,C> : A = B = C\}$$

Test case ID	A	B	C	Expected Result	Classes Covered by the test case
1	25	13	13	A is greatest	I_1, I_5
2	25	40	25	B is greatest	I_2, I_6
3	24	24	37	C is greatest	I_3, I_4
4	25	25	25	All three are equal	I_7

Example 4.6

A program determines the next date in the calendar. Its input is entered in the form of <ddmmyyyy> with the following range:

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

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

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

Its output would be the next date or an error message 'invalid date.' Design test cases using equivalence class partitioning method.

Solution

First we partition the domain of input in terms of valid input values and invalid values, getting the following classes:

$$I_1 = \{<m, d, y> : 1 \le m \le 12\}$$
$$I_2 = \{<m, d, y> : 1 \le d \le 31\}$$
$$I_3 = \{<m, d, y> : 1900 \le y \le 2025\}$$
$$I_4 = \{<m, d, y> : m < 1\}$$
$$I_5 = \{<m, d, y> : m > 12\}$$
$$I_6 = \{<m, d, y> : d < 1\}$$
$$I_7 = \{<m, d, y> : d > 31\}$$
$$I_8 = \{<m, d, y> : y < 1900\}$$
$$I_9 = \{<m, d, y> : y > 2025\}$$

The test cases can be designed from the above derived classes, taking one test case from each class such that the test case covers maximum valid input classes, and separate test cases for each invalid class. The test cases are shown below:

Test case ID	mm	dd	yyyy	Expected result	Classes covered by the test case
1	5	20	1996	21-5-1996	I_1, I_2, I_3
2	0	13	2000	Invalid input	I_4
3	13	13	1950	Invalid input	I_5
4	12	0	2007	Invalid input	I_6
5	6	32	1956	Invalid input	I_7
6	11	15	1899	Invalid input	I_8
7	10	19	2026	Invalid input	I_9

Example 4.7

A program takes an angle as input within the range [0, 360] and determines in which quadrant the angle lies. Design test cases using equivalence class partitioning method.

Solution

1. First we partition the domain of input as valid and invalid values, getting the following classes:

$$I_1 = \{<\text{Angle}> : 0 \le \text{Angle} \le 360\}$$
$$I_2 = \{<A,B,C> : \text{Angle} < 0\}$$
$$I_3 = \{<A,B,C> : \text{Angle} > 0\}$$

The test cases designed from these classes are shown below:

Test Case ID	Angle	Expected results	Classes covered by the test case
1	50	I Quadrant	I_1
2	–1	Invalid input	I_2
3	361	Invalid input	I_3

2. The classes can also be prepared based on the output criteria as shown below:

$$O_1 = \{\text{<Angle>: First Quadrant, if } 0 \le \text{Angle} \le 90\}$$
$$O_2 = \{\text{<Angle>: Second Quadrant, if } 91 \le \text{Angle} \le 180\}$$
$$O_3 = \{\text{<Angle>: Third Quadrant, if } 181 \le \text{Angle} \le 270\}$$
$$O_4 = \{\text{<Angle>: Fourth Quadrant, if } 271 \le \text{Angle} \le 360\}$$
$$O_5 = \{\text{<Angle>: Invalid Angle}\};$$

However, O_5 is not sufficient to cover all invalid conditions this way. Therefore, it must be further divided into equivalence classes as shown below:

$$O_{51} = \{\text{<Angle>: Invalid Angle, if Angle} < 0\}$$
$$O_{52} = \{\text{<Angle>: Invalid Angle, if Angle} > 360\}$$

Now the test cases can be designed from the above derived classes as shown below:

Test Case ID	Angle	Expected results	Classes covered by the test case
1	50	I Quadrant	O_1
2	135	II Quadrant	O_2
3	250	III Quadrant	O_3
4	320	IV Quadrant	O_4
5	370	Invalid angle	O_{51}
6	–1	Invalid angle	O_{52}

4.3 State Table-Based Testing

Tables are useful tools for representing and documenting many types of information relating to test case design. These are beneficial for the applications which can be described using state transition diagrams and state tables. First we define some basic terms related to state tables which are discussed below.

4.3.1 Finite State Machine (FSM)

An FSM is a behavioural model whose outcome depends upon both previous and current inputs. FSM models can be prepared for software structure or software behaviour. And it can be used as a tool for functional testing. Many testers prefer to use FSM model as a guide to design functional tests.

4.3.2 State Transition Diagrams or State Graph

A system or its components may have a number of states depending on its input and time. For example, a task in an operating system can have the following states:

1. **New State**: When a task is newly created.
2. **Ready**: When the task is waiting in the ready queue for its turn.
3. **Running**: When instructions of the task are being executed by CPU.
4. **Waiting**: When the task is waiting for an I/O event or reception of a signal.
5. **Terminated**: The task has finished execution.

States are represented by *nodes*. Now with the help of nodes and transition links between the nodes, a state transition diagram or state graph is prepared. A state graph is the pictorial representation of an FSM. Its purpose is to depict the states that a system or its components can assume. It shows the events or circumstances that cause or result from a change from one state to another [21].

Whatever is being modeled is subjected to inputs. As a result of these inputs, when one state is changed to another, it is called a *transition*. Transitions are represented by links that join the nodes. The state graph of task states is shown in Fig. 4.5.

Figure 4.5 State graph

Each arrow link provides two types of information:

1. Transition events like admitted, dispatch, interrupt, etc.

2. The resulting output from a state like T1, T2, T3, etc.

   T0 = Task is in new state and waiting for admission to ready queue

   T1 = A new task admitted to ready queue

   T2 = A ready task has started running

   T3 = Running task has been interrupted

   T4 = Running task is waiting for I/O or event

   T5 = Wait period of waiting task is over

   T6 = Task has completed execution

4.3.3 State Table

State graphs of larger systems may not be easy to understand. Therefore, state graphs are converted into tabular form for convenience sake, which are known as state tables. State tables also specify states, inputs, transitions, and outputs. The following conventions are used for state table [9]:

- Each row of the table corresponds to a state.
- Each column corresponds to an input condition.
- The box at the intersection of a row and a column specifies the next state (transition) and the output, if any.

The state table for task states is given in Table 4.1.

Table 4.1 State table

State\Input Event	Admit	Dispatch	Interrupt	I/O or Event Wait	I/O or Event Wait Over	Exit
New	**Ready/ T1**	New / T0	New / T0	New / T0	New / T0	New / T0
Ready	Ready/ T1	**Running/ T2**	Ready / T1	Ready / T1	Ready / T1	Ready / T1
Running	Running/T2	Running/ T2	**Ready / T3**	**Waiting/ T4**	Running/ T2	**Terminated/T6**
Waiting	Waiting/T4	Waiting / T4	Waiting/T4	Waiting / T4	**Ready / T5**	Waiting / T4

The highlighted cells of the table are valid inputs causing a change of state. Other cells are invalid inputs which do not cause any transition in the state of a task.

4.3.4 State Table-based Testing

After reviewing the basics, we can start functional testing with state tables. A state graph and its companion state table contain information that is converted into test cases.

The procedure for converting state graphs and state tables into test cases is discussed below.

1. **Identify the states** The number of states in a state graph is the number of states we choose to recognize or model. In practice, the state is directly or indirectly recorded as a combination of values of variables that appear in the database. As an example, the state could be composed of the values of a counter whose possible values ranged from 0 to 9, combined with the setting of two bit flags, leading to a total of $2 \times 2 \times 10 = 40$ states. When the state graph represents an explicit state table implementation, this value is encoded so that bugs in the number of states are less likely; but the encoding can be wrong. Failing to account for all the states is one of the more common bugs in the software that can be modeled by state graphs. As an explicit state table mechanization is not typical, the opportunities for missing states abound. Find the number of states as follows [9]:
   - Identify all the component factors of the state.
   - Identify all the allowable values for each factor.
   - The number of states is the product of the number of allowable values of all the factors.
2. **Prepare state transition diagram after understanding transitions between states** After having all the states, identify the inputs on each state and transitions between states and prepare the state graph. Every input state combination must have a specified transition. If the transition is impossible, then there must be a mechanism that prevents that input from occurring in that state.
   A program cannot have contradictions or ambiguities. Ambiguities are impossible because the program will do something for every input. Even if the state does not change, by definition, this is a transition to the same state. A seeming contradiction could come about in a model if all the factors that constitute the state and all the inputs are not taken care of. If someone as a designer says while debugging, 'sometimes it works and sometimes it doesn't', it means there is a state factor about which he is not aware—a factor probably caused by a bug. Exploring the real state graph and recording the transitions and outputs for each combination of input and state may lead to discovering the bug.
3. **Convert the state graph into the state table as discussed earlier**
4. **Analyse the state table for its completeness**
5. **Create the corresponding test cases from the state table**
   Test cases are produced in a tabular form known as the *test case table* which contains six columns as shown below [22]:

Test case ID : a unique identifier for each test case

Test Source : a trace back to the corresponding cell in the state table

Current State : the initial condition to run the test

Event : the input triggered by the user

Output : the current value returned

Next State : the new state achieved

The test cases derived from the state table (Table 4.1) are shown below in Table 4.2.

Table 4.2 Deriving test cases from state table

Test Case ID	Test Source	Input		Expected Results	
		Current State	Event	Output	Next State
TC1	Cell 1	New	Admit	T1	Ready
TC2	Cell 2	New	Dispatch	T0	New
TC3	Cell 3	New	Interrupt	T0	New
TC4	Cell 4	New	I/O wait	T0	New
TC5	Cell 5	New	I/O wait over	T0	New
TC6	Cell 6	New	exit	T0	New
TC7	Cell 7	Ready	Admit	T1	Ready
TC8	Cell 8	Ready	Dispatch	T2	Running
TC9	Cell 9	Ready	Interrupt	T1	Ready
TC10	Cell 10	Ready	I/O wait	T1	Ready
TC11	Cell 11	Ready	I/O wait	T1	Ready
TC12	Cell 12	Ready	Exit	T1	Ready
TC13	Cell 13	Running	Admit	T2	Running
TC14	Cell 14	Running	Dispatch	T2	Running
TC15	Cell 15	Running	Interrupt	T3	Ready
TC16	Cell 16	Running	I/O wait	T4	Waiting
TC17	Cell 17	Running	I/O wait over	T2	Running
TC18	Cell 18	Running	Exit	T6	Terminated
TC19	Cell 19	Waiting	Admit	T4	Waiting
TC20	Cell 20	Waiting	Dispatch	T4	Waiting
TC21	Cell 21	Waiting	Interrupt	T4	Waiting
TC22	Cell 22	Waiting	I/O wait	T4	Waiting
TC23	Cell 23	Waiting	I/O wait over	T5	Ready
TC24	Cell 24	Waiting	Exit	T4	Waiting

Test case TC1 gets information from Cell 1 and indicates that one task is created *New* in the system. This new task is getting the input event *Admit* and transitions to the new state *Ready* and the output is T1 (i.e. task is present in Ready queue). Similarly other test cases can be derived. From the state graph, we can recognize all input sequences and can form some detailed test cases also.

4.4 Decision Table-Based Testing

Boundary value analysis and equivalence class partitioning methods do not consider combinations of input conditions. These consider each input separately. There may be some critical behaviour to be tested when some combinations of input conditions are considered.

Decision table is another useful method to represent the information in a tabular method. It has the specialty to consider complex combinations of input conditions and resulting actions. Decision tables obtain their power from logical expressions. Each operand or variable in a logical expression takes on the value, TRUE or FALSE.

4.4.1 Formation of Decision Table

A decision table is formed with the following components (see Table 4.3):

Table 4.3 Decision table structure

ENTRY

		Rule 1	Rule 2	Rule 3	Rule 4	...
Condition Stub	C1	True	True	False	I	
	C2	False	True	False	True	
	C3	True	True	True	I	
Action Stub	A1		X			
	A2	X			X	
	A3			X		

Condition stub It is a list of input conditions for which the complex combination is made.

Action stub It is a list of resulting actions which will be performed if a combination of input condition is satisfied.

Condition entry It is a specific entry in the table corresponding to input conditions mentioned in the condition stub. When we enter TRUE or FALSE

for all input conditions for a particular combination, then it is called a *Rule*. Thus, a rule defines which combination of conditions produces the resulting action. When the condition entry takes only two values—TRUE or FALSE, then it is called *Limited Entry Decision Table*. When the condition entry takes several values, then it is called *Extended Entry Decision Table*. In limited entry decision table, condition entry, which has no effect whether it is True or False, is called a *Don't-Care* state or *immaterial state* (represented by *I*). The state of a don't-care condition does not affect the resulting action.

Action entry It is the entry in the table for the resulting action to be performed when one rule (which is a combination of input condition) is satisfied. 'X' denotes the action entry in the table.

The guidelines to develop a decision table for a problem are discussed below [7]:

- List all actions that can be associated with a specific procedure (or module).
- List all conditions (or decision made) during execution of the procedure.
- Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions.
- Define rules by indicating what action occurs for a set of conditions.

4.4.2 Test Case Design using Decision Table

For designing test cases from a decision table, following interpretations should be done:

- Interpret condition stubs as the inputs for the test case.
- Interpret action stubs as the expected output for the test case.
- Rule, which is the combination of input conditions, becomes the test case itself.
- If there are k rules over n binary conditions, there are at least k test cases and at the most 2^n test cases.

Example 4.8

A program calculates the total salary of an employee with the conditions that if the working hours are less than or equal to 48, then give normal salary. The hours over 48 on normal working days are calculated at the rate of 1.25 of the salary. However, on holidays or Sundays, the hours are calculated at the rate of 2.00 times of the salary. Design test cases using decision table testing.

Solution

The decision table for the program is shown below:

ENTRY

		Rule 1	Rule 2	Rule3
Condition Stub	C1: Working hours > 48	I	F	T
	C2: Holidays or Sundays	T	F	F
Action Stub	A1: Normal salary		X	
	A2: 1.25 of salary			X
	A3: 2.00 of salary	X		

The test cases derived from the decision table are given below:

Test Case ID	Working Hour	Day	Expected Result
1	48	Monday	Normal Salary
2	50	Tuesday	1.25 of salary
3	52	Sunday	2.00 of salary

Example 4.9

A wholesaler has three commodities to sell and has three types of customers. Discount is given as per the following procedure:

(i) For DGS & D orders, 10% discount is given irrespective of the value of the order.

(ii) For orders of more than Rs 50,000, agents get a discount of 15% and the retailer gets a discount of 10%.

(iii) For orders of Rs 20,000 or more and up to Rs 50,000, agents get 12% and the retailer gets 8% discount.

(iv) For orders of less than Rs 20,000, agents get 8% and the retailer gets 5% discount.

The above rules do not apply to the furniture items wherein a flat rate of 10% discount is admissible to all customers irrespective of the value of the order.

Design test cases for this system using decision table testing.

Solution

		ENTRY							
		R1	R2	R3	R4	R5	R6	R7	R8
Condition Stub	C1: DGS & D	T	F	F	F	F	F	F	F
	C2: Agent	F	T	F	T	F	T	F	I
	C3: Retailer	F	F	T	F	T	F	T	I
	C4: Order > 50,000	I	T	T	F	F	F	F	I
	C5: Order ≥ 20000 to < 50,000	I	F	F	T	T	F	F	I
	C6: Order < 20,000	I	F	F	F	F	T	T	I
	C7: Furniture	F	F	F	F	F	F	F	T
Action Stub	A1: Discount of 5%							X	
	A2: Discount of 8%					X	X		
	A3: Discount of 10%	X		X					X
	A4: Discount of 12%				X				
	A5: Discount of 15%		X						

The test cases derived from the decision table are given below:

Test Case ID	Type of Customer	Product Furniture?	Order Value (Rs)	Expected Result
1	DGS & D	No	51,000	10% Discount
2	Agent	No	52,000	15% Discount
3	Retailer	No	53,000	10% Discount
4	Agent	No	23,000	12% Discount
5	Retailer	No	27,000	8% Discount
6	Agent	No	15,000	8% Discount
7	Retailer	No	18,000	5% Discount
8	Agent	Yes	34,000	10% Discount

Example 4.10

A university is admitting students in a professional course subject to the following conditions:

(a) Marks in Java ≥ 70

(b) Marks in C++ ≥ 60

(c) Marks in OOAD ≥ 60

(d) Total in all three subjects ≥ 220 OR Total in Java and C++ ≥ 150

If the aggregate mark of an eligible candidate is more than 240, he will be eligible for scholarship course, otherwise he will be eligible for normal course. The program reads the marks in the three subjects and generates the following outputs:

(i) Not eligible

(ii) Eligible for scholarship course

(iii) Eligible for normal course

Design test cases for this program using decision table testing.

Solution

ENTRY

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
C1: marks in Java ≥ 70	T	T	T	T	F	I	I	I	T	T
C2: marks in C++ ≥ 60	T	T	T	T	I	F	I	I	T	T
C3: marks in OOAD ≥ 60	T	T	T	T	I	I	F	I	T	T
C4:Total in three subjects ≥ 220	T	F	T	T	I	I	I	F	T	T
C5:Total in Java & C++ ≥ 150	F	T	F	T	I	I	I	F	T	T
C6:Aggregate marks > 240	F	F	T	T	I	I	I	I	F	T
A1: Eligible for normal course	X	X							X	
A2: Eligible for scholarship course			X	X						X
A3: Not eligible					X	X	X	X		

Test Case ID	Java	C++	OOAD	Aggregate Marks	Expected Output
1	70	75	60	224	Eligible for normal course
2	75	75	70	220	Eligible for normal course
3	75	74	91	242	Eligible for scholarship course
4	76	77	89	242	Eligible for scholarship course
5	68	78	80	226	Not eligible
6	78	45	78	201	Not eligible
7	80	80	50	210	Not eligible
8	70	72	70	212	Not eligible
9	75	75	70	220	Eligible for normal course
10	76	80	85	241	Eligible for scholarship course

4.4.3 Expanding the Immaterial Cases in Decision Table

Immaterial cases (I) have been shown in the decision table which are *don't-care conditions*. These conditions mean that the value of a particular condition in the specific rule does not make a difference whether it is TRUE or FALSE. However, we should always test both the values of a don't-care condition. So the rules in the table can be expanded. Sometimes expanding the decision table to spell out don't-care conditions can reveal hidden problems. Original decision table shown till now in the examples above and the expanded table are logically equivalent, implying that the combinations of conditions result in tests that exercise the same circumstances.

Example 4.11

Consider Example 4.8 once again whose decision table is shown below with immaterial cases.

ENTRY

		Rule 1	Rule 2	Rule 3
Condition Stub	C1: Working hours > 48	I	F	T
	C2: Holidays or Sundays	T	F	F
Action Stub	A1: Normal salary		X	
	A2: 1.25 of salary			X
	A3: 2.00 of salary	X		

The immaterial test case in Rule 1 of the above table can be expanded by taking both T and F values of C1. The expanded decision table is shown below:

ENTRY

		Rule 1-1	Rule 1-2	Rule 2	Rule 3
Condition Stub	C1: Working hours > 48	F	T	F	T
	C2: Holidays or Sundays	T	T	F	F
Action Stub	A1: Normal salary			X	
	A2: 1.25 of salary				X
	A3: 2.00 of salary	X	X		

The test cases derived from the expanded decision table are given below:

Test Case ID	Working Hour	Day	Expected Result
1	48	Monday	Normal salary
2	50	Tuesday	1.25 of salary
3	52	Sunday	2.00 of salary
4	30	Sunday	2.00 of salary

4.5 Cause-Effect Graphing Based Testing

As said earlier, boundary value analysis and equivalence class partitioning methods do not consider combinations of input conditions. Like decision tables, cause-effect graphing is another technique for combinations of input conditions. But cause-effect graphing takes the help of decision table to design a test case. Therefore, cause-effect graphing is the technique to represent the situations of combinations of input conditions and then we convert the cause-effect graph into decision table for the test cases.

One way to consider all valid combinations of input conditions is to consider all valid combinations of the equivalence classes of input conditions. This simple approach will result in an unusually large number of test cases, many of which will not be useful for revealing any new errors. For example, if there are n different input conditions, such that a combination is valid, we will have 2^n test cases. Cause-effect graphing techniques help in selecting combinations of input conditions in a systematic way, such that the number of test cases does not become unmanageably large [14].

The following process is used to derive the test cases [2].

Division of specification The specification is divided into workable pieces, as cause-effect graphing becomes complex when used on large specifications.

Identification of causes and effects The next step is to identify *causes and effects* in the specifications. A cause is a distinct input condition identified in the problem. It may also be an equivalence class of input conditions. Similarly, an effect is an output condition.

Transformation of specification into a cause-effect graph Based on the analysis of the specification, it is transformed into a Boolean graph linking the causes and effects. This is the cause-effect graph. Complete the graph by adding the constraints, if any, between causes and effects.

Conversion into decision table The cause-effect graph obtained is converted into a limited-entry decision table by verifying state conditions in the graph. Each column in the table represents a test case.

Deriving test cases The columns in the decision table are converted into test cases.

4.5.1 Basic Notations for Cause-Effect Graph

Identity

According to the identity function, if x is 1, y is 1; else y is 0.

NOT

This function states that if x is 1, y is 0; else y is 1.

OR

The OR function states that if x or y or z is 1, A is 1; else A is 0.

AND

This function states that if both x and y are 1, z is 1; else z is 0.

Exclusive

Sometimes, the specification contains an impossible combination of causes such that two causes cannot be set to 1 simultaneously. For this, Exclusive function is used. According to this function, it always holds that either x or y can be 1, i.e. x and y cannot be 1 simultaneously.

Inclusive

It states that at least one of x, y, and z must always be 1 (x, y, and z cannot be 0 simultaneously).

One and Only One

It states that one and only one of x and y must be 1.

Requires

It states that for x to be 1, y must be 1, i.e. it is impossible for x to be 1 and y to be 0.

Mask

It states that if x is 1, y is forced to 0.

Example 4.12

A program has been designed to determine the nature of roots of a quadratic equation. The quadratic equation takes three input values from the range [0, 100]. Design the test cases using cause-effect graphing technique.

Solution

First we identify the causes and effects in this problem as given below:

C1: $a \neq 0$
C2: $b = 0$
C3: $c = 0$
C4: $D > 0$ where D is $b^2 - 4ac$
C5: $D < 0$
C6: $D = 0$
C7: $a = b = c$
C8: $a = c = b/2$
E1: Not a quadratic equation
E2: Real roots
E3: Imaginary roots
E4: Equal roots

Based on these causes and effects and analysing the constraints, the cause-effect graph is prepared, as shown in Fig. 4.6.

Figure 4.6 Cause-effect graph for Example 4.12

Now this graph is converted into a limited entry decision table as shown below.

	R1	R2	R3	R4	R5	R6	R7
C1: a ≠ 0	T	T	T	T	T	T	F
C2: b = 0	F	I	I	T	F	F	I
C3: c = 0	I	F	I	T	F	F	I
C4: D > 0	T	F	F	F	F	F	I
C5: D < 0	F	T	F	F	T	F	I
C6: D = 0	F	F	T	T	F	T	I
C7: a = b = c	F	I	F	F	T	F	I
C8: a = c = b/2	F	F	I	F	F	T	I
A1: Not a quadratic equation							X
A2: Real roots	X						
A3: Imaginary roots		X			X		
A4: Equal roots			X	X		X	

The test cases designed based on this table are given below.

Test Case ID	a	b	c	Expected Output
1	1	50	50	Real roots
2	100	50	50	Imaginary roots
3	1	6	9	Equal
4	100	0	0	Equal
5	99	99	99	Imaginary
6	50	100	50	Equal
7	0	50	30	Not a quadratic equation

4.6 Error Guessing

Error guessing is the preferred method used when all other methods fail. Sometimes it is used to test some special cases. According to this method, errors or bugs can be guessed which do not fit in any of the earlier defined situations. So test cases are generated for these special cases.

It is a very practical case wherein the tester uses his intuition and makes a guess about where the bug can be. The tester does not have to use any particular testing technique. However, this capability comes with years of experience in a particular field of testing. This is the reason that experienced managers can easily smell out errors as compared to a novice tester.

As discussed earlier, every testing activity is recorded. The history of bugs can help in identifying some special cases in the project. There is a high probability that errors made in a previous project is repeated again. In these situations, error guessing is an *ad hoc approach*, based on intuition, experience, knowledge of project, and bug history. Any of these can help to expose the errors. The basic idea is to make a list of possible errors in error-prone situations and then develop the test cases. Thus, there is no general procedure for this technique, as it is largely an intuitive and ad hoc process.

For example, consider the system for calculating the roots of a quadratic equation. Some special cases in this system are as follows:

- What will happen when $a = 0$? Though, we do consider this case, there are chances that we overlook it while testing, as it has two cases:
  - (i) If $a = 0$ then the equation is no longer quadratic.
  - (ii) For calculation of roots, division is by zero.
- What will happen when all the inputs are negative?
- What will happen when the input list is empty?

SUMMARY

Black-box testing is a dynamic testing technique, wherein the software is tested using a set of inputs and expected outputs. It does not consider the logics of the program and how it has been developed. It considers only the functionality of the program and how it behaves under test. This chapter discusses black-box testing techniques. All the methods have been discussed taking sufficient number of examples such that a new method can be learnt easily and the reader can design the test cases using a particular method.

Let us review the important concepts described in this chapter:

- Black-box testing is a technique, wherein the structure of program is overlooked. The internal logic of the program is unknown to the tester.
- Boundary value analysis is the method to uncover the bugs by looking at the boundary of the inputs used in the program. The idea is that the programmer tests the program by having nominal values of the inputs but does not check its boundary values and thus leaves the bugs.
- Equivalence class partitioning is a method for deriving test cases, wherein classes of input conditions called equivalence classes are identified such that each member of the class causes the same kind of processing and output to occur. It reduces the input domain space and thereby the testing effort.
- State table-based testing is a convenient method for testing systems where states and transitions are specified.
- Decision table is another useful method to represent the information in a tabular method. It has the speciality to consider complex combinations of input conditions and resulting actions.

- Decision table consists of: condition stub, action stub, condition entry, and action entry.
- In a decision table, when we enter TRUE or FALSE for all input conditions for a particular combination, then it is called a Rule.
- In a decision table, when the condition entry takes only two values – TRUE or FALSE, then it is called a Limited Entry Decision Table. When the condition entry takes several values, then it is called an Extended Entry Decision Table.
- Cause-effect graphing is the technique to represent combinations of input conditions.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Black-box testing is a _____.
   (a) Static testing
   (b) Dynamic testing
   (c) None of the above
2. It has been observed that test cases, which are designed with boundary input values, have a _____ chance of finding errors.
   (a) High
   (b) Low
   (c) Medium
   (d) Zero
3. How many test cases are there in BVC if there are 5 variables in a module?
   (a) 23
   (b) 13
   (c) 10
   (d) 21
4. How many test cases are there in robustness testing if there are 5 variables in a module?
   (a) 23
   (b) 31
   (c) 10
   (d) 21
5. How many test cases are there in worst-case testing if there are 4 variables in a module?
   (a) 623
   (b) 513
   (c) 625
   (d) 521

6. Each row of state table corresponds to _____.
   (a) Input
   (b) State
   (c) Transition
   (d) None of the above
7. Each column of state table corresponds to _____.
   (a) Input
   (b) State
   (c) Transition
   (d) None of the above
8. Intersection of a row and a column specifies _____.
   (a) Input
   (b) State
   (c) Transition and output
   (d) None of the above
9. What are the components of a decision table?
   (a) Condition stub
   (b) Condition entry
   (c) Action stub
   (d) All
10. If there are *k* rules over *n* binary conditions, there are at least _____ test cases and at the most _____ test cases.
    (a) $k+2$, 2^{n+2}
    (b) $k+3$, 2^{n+3}
    (c) k, 2^{n}
    (d) None of the above
11. Boundary value analysis and equivalence class partitioning methods do not consider _____.
    (a) Combinations of input conditions
    (b) Inputs
    (c) Outputs
    (d) None

Review Questions

1. What are the types of errors detected by black-box testing?
2. Which type of testing is possible with BVA?
3. Which type of testing is possible with equivalence class partitioning?

4. A program calculates the GCD of three numbers in the range [1, 50]. Design test cases for this program using BVC, robust testing, and worst-case testing methods.
5. A program takes as input a string (5–20 characters) and a single character and checks whether that single character is present in the string or not. Design test cases for this program using BVC, robust testing, and worst-case testing methods.
6. A program reads the data of employees in a company by taking the following inputs and prints them:

   Name of Employee (Max. 15 valid characters A–Z, a–z, space)

   Employee ID (10 characters)

   Designation (up to 20 characters)

   Design test cases for this program using BVC, robust testing, and worst-case testing methods.
7. A mobile phone service provider uses a program that computes the monthly bill of customers as follows:

   Minimum Rs 300 for up to 120 calls

   Plus Re 1 per call for the next 70 calls

   Plus Rs 0.80 per call for the next 50 calls

   Plus Rs 0.40 per call for any call beyond 220 calls.

   Design test cases for this program using equivalence class testing technique.
8. A program reads players' records with the following detail and prints a team-wise list containing player name with their batting average:

   Player name (max. 30 characters)

   Team name (max. 20 characters)

   Batting average

   Design test cases for this program using BVC, robust testing, and worst-case testing methods.
9. Consider Example 4.10 and design test cases for this program using equivalence class testing technique.
10. Consider a system having an FSM for a stack having the following states and transitions:

    States

    Initial: Before creation

    Empty: Number of elements = 0

    Holding: Number of elements > 0, but less than the maximum capacity

    Full: Number elements = maximum

    Final: After destruction

Transitions

Initial to Empty: Create

Empty to Holding, Empty to Full, Holding to Holding, Holding to Full: Add

Empty to Final, Full to Final, Holding to Final: Destroy

Holding to Empty, Full to Holding, Full to Empty: Delete

Design test cases for this FSM using state table-based testing.

11. Passengers who travel more than 50,000 km. per calendar year and in addition, pay cash for tickets or have been traveling regularly for more than eight years are to receive a free round trip ticket around India. Passengers who travel less than 50,000 km. per calendar year and have been availing railway services regularly for more than eight years also get a free round ticket around India.

    Design test cases for this system using decision table testing.

12. Consider Problem 6 and design test cases using cause-effect graphing technique.
13. How do you expand immaterial test cases in decision table testing? Expand the immaterial test cases in Problem 11.
14. 'Error guessing is an ad hoc approach for testing.' Comment on this statement.
15. A program takes as input three angles and determines the type of triangle. If all the three angles are less than 90, it is an acute angled triangle. If one angle is greater than 90, it is an obtuse angled triangle. If one angle is equal to 90, it is a right angled triangle.

    Design test cases for this program using equivalence class testing technique.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter

5

Dynamic Testing: White-Box Testing Techniques

White-box testing is another effective testing technique in dynamic testing. It is also known as *glass-box* testing, as everything that is required to implement the software is visible. The entire design, structure, and code of the software have to be studied for this type of testing. It is obvious that the developer is very close to this type of testing. Often, developers use white-box testing techniques to test their own design and code. This testing is also known as *structural* or *development testing*.

In white-box testing, structure means the logic of the program which has been implemented in the language code. The intention is to test this logic so that required results or functionalities can be achieved. Thus, white-box testing ensures that the internal parts of the software are adequately tested.

OBJECTIVES

After reading this chapter, you should be able to understand:

- White-box testing demands complete understanding of the program logic/structure
- Test case designing using white-box testing techniques
- Basis path testing method
- Building a path testing tool using graph matrices
- Loop testing
- Data flow testing method
- Mutation testing method

5.1 Need of White-box testing

Is white-box testing really necessary? Can't we write the code and simply test the software using black-box testing techniques? The supporting reasons for white-box testing are given below:

1. In fact, white-box testing techniques are used for testing the module for initial stage testing. Black-box testing is the second stage for testing the software. Though test cases for black box can be designed earlier than white-box test cases, they cannot be executed until the code is produced and checked using white-box testing techniques. Thus, white-box testing is not an alternative but an essential stage.
2. Since white-box testing is complementary to black-box testing, there are categories of bugs which can be revealed by white-box testing, but

not through black-box testing. There may be portions in the code which are not checked when executing functional test cases, but these will be executed and tested by white-box testing.

3. Errors which have come from the design phase will also be reflected in the code, therefore we must execute white-box test cases for verification of code (unit verification).
4. We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis. White-box testing explores these paths too.
5. Some typographical errors are not observed and go undetected and are not covered by black-box testing techniques. White-box testing techniques help detect these errors.

5.2 Logic Coverage Criteria

Structural testing considers the program code, and test cases are designed based on the logic of the program such that every element of the logic is covered. Therefore the intention in white-box testing is to cover the whole logic. Discussed below are the basic forms of logic coverage.

Statement Coverage

The first kind of logic coverage can be identified in the form of statements. It is assumed that if all the statements of the module are executed once, every bug will be notified.

Consider the following code segment shown in Fig. 5.1.

```
scanf ("%d", &x);
scanf ("%d", &y);
while (x != y)
{
     if (x > y)
          x = x - y;
     else
          y = y - x;
 }
 printf ("x = ", x);
 printf ("y = ", y);
```

Figure 5.1 Sample code

If we want to cover every statement in the above code, then the following test cases must be designed:

Test case 1: $x = y = n$, where n is any number

Test case 2: $x = n$, $y = n'$, where n and n' are different numbers.

Test case 1 just skips the while loop and all loop statements are not executed. Considering test case 2, the loop is also executed. However, every statement inside the loop is not executed. So two more cases are designed:

Test case 3: $x > y$

Test case 4: $x < y$

These test cases will cover every statement in the code segment, however statement coverage is a poor criteria for logic coverage. We can see that test case 3 and 4 are sufficient to execute all the statements in the code. But, if we execute only test case 3 and 4, then conditions and paths in test case 1 will never be tested and errors will go undetected. Thus, *statement coverage is a necessary but not a sufficient criteria for logic coverage.*

Decision or Branch Coverage

Branch coverage states that each decision takes on all possible outcomes (True or False) at least once. In other words, each branch direction must be traversed at least once. In the previous sample code shown in Figure 5.1, *while* and *if* statements have two outcomes: True and False. So test cases must be designed such that both outcomes for *while* and *if* statements are tested. The test cases are designed as:

Test case 1: $x = y$

Test case 2: $x \mathrel{!}= y$

Test case 3: $x < y$

Test case 4: $x > y$

Condition Coverage

Condition coverage states that each condition in a decision takes on all possible outcomes at least once. For example, consider the following statement:

while ((I≤5) && (J< COUNT))

In this loop statement, two conditions are there. So test cases should be designed such that both the conditions are tested for True and False outcomes. The following test cases are designed:

Test case 1: $I \leq 5, J < \text{COUNT}$

Test case 2: $I < 5, J > \text{COUNT}$

Decision/condition Coverage

Condition coverage in a decision does not mean that the decision has been covered. If the decision

if (A && B)

is being tested, the condition coverage would allow one to write two test cases:

Test case 1: *A* is True, *B* is False.

Test case 2: *A* is False, *B* is True.

But these test cases would not cause the THEN clause of the IF to execute (i.e. execution of decision). The obvious way out of this dilemma is a criterion called decision/condition coverage. It requires sufficient test cases such that each condition in a decision takes on all possible outcomes at least once, each decision takes on all possible outcomes at least once, and each point of entry is invoked at least once [2].

Multiple condition coverage In case of multiple conditions, even decision/condition coverage fails to exercise all outcomes of all conditions. The reason is that we have considered all possible outcomes of each condition in the decision, but we have not taken all combinations of different multiple conditions. Certain conditions mask other conditions. For example, if an AND condition is False, none of the subsequent conditions in the expression will be evaluated. Similarly, if an OR condition is True, none of the subsequent conditions will be evaluated. Thus, condition coverage and decision/condition coverage need not necessarily uncover all the errors.

Therefore, multiple condition coverage requires that we should write sufficient test cases such that all possible combinations of condition outcomes in each decision and all points of entry are invoked at least once. Thus, as in decision/condition coverage, all possible combinations of multiple conditions should be considered. The following test cases can be there:

Test case 1: $A =$ True, $B =$ True

Test case 2: $A =$ True, $B =$ False

Test case 3: $A =$ False, $B =$ True

Test case 4: $A =$ False, $B =$ False

5.3 Basis Path Testing

Basis path testing is the oldest structural testing technique. The technique is based on the control structure of the program. Based on the control structure, a flow graph is prepared and all the possible paths can be covered and executed during testing. Path coverage is a more general criterion as compared to other coverage criteria and useful for detecting more errors. But the problem with path criteria is that programs that contain loops may have an infinite number of possible paths and it is not practical to test all the paths. Some criteria should be devised such that selected paths are executed for maximum

coverage of logic. Basis path testing is the technique of selecting the paths that provide a basis set of execution paths through the program.

The guidelines for effectiveness of path testing are discussed below:

1. Path testing is based on control structure of the program for which flow graph is prepared.
2. Path testing requires complete knowledge of the program's structure.
3. Path testing is closer to the developer and used by him to test his module.
4. The effectiveness of path testing gets reduced with the increase in size of software under test [9].
5. Choose enough paths in a program such that maximum logic coverage is achieved.

5.3.1 Control Flow Graph

The control flow graph is a graphical representation of control structure of a program. Flow graphs can be prepared as a directed graph. A directed graph (V, E) consists of a set of vertices V and a set of edges E that are ordered pairs of elements of V. Based on the concepts of directed graph, following notations are used for a flow graph:

- ***Node*** It represents one or more procedural statements. The nodes are denoted by a circle. These are numbered or labeled.
- ***Edges or links*** They represent the flow of control in a program. This is denoted by an arrow on the edge. An edge must terminate at a node.
- ***Decision node*** A node with more than one arrow leaving it is called a decision node.
- ***Junction node*** A node with more than one arrow entering it is called a junction.
- ***Regions*** Areas bounded by edges and nodes are called regions. When counting the regions, the area outside the graph is also considered a region.

5.3.2 Flow Graph Notations for Different Programming Constructs

Since a flow graph is prepared on the basis of control structure of a program, some fundamental graphical notations are shown here (see Fig. 5.2) for basic programming constructs.

Using the above notations, a flow graph can be constructed. Sequential statements having no conditions or loops can be merged in a single node. That is why, the flow graph is also known as *decision-to-decision-graph* or *DD graph*.

Figure 5.2

5.3.3 Path Testing Terminology

Path A path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision, or exit. A path may go through several junctions, processes, or decisions, one or more times.

Segment Paths consist of segments. The smallest segment is a link, that is, a single process that lies between two nodes (e.g., junction-process-junction, junction-process-decision, decision-process-junction, decision-process-decision). A direct connection between two nodes, as in an unconditional GOTO, is also called a process by convention, even though no actual processing takes place.

Path segment A path segment is a succession of consecutive links that belongs to some path.

Length of a path The length of a path is measured by the number of links in it and not by the number of instructions or statements executed along the path. An alternative way to measure the length of a path is by the number of nodes traversed. This method has some analytical and theoretical benefits. If programs are assumed to have an entry and an exit node, then the number of links traversed is just one less than the number of nodes traversed.

Independent path An independent path is any path through the graph that introduces at least one new set of processing statements or new conditions. An independent path must move along at least one edge that has not been traversed before the path is defined [9,28].

5.3.4 Cyclomatic Complexity

McCabe [24] has given a measure for the logical complexity of a program by considering its control flow graph. His idea is to measure the complexity by considering the number of paths in the control graph of the program. But even for simple programs, if they contain at least one cycle, the number of paths is infinite. Therefore, he considers only independent paths.

Figure 5.3 Sample graph

In the graph shown in Figure 5.3, there are six possible paths: *acei, acgh, acfh, bdei, bdgh, bdfj*.

In this case, we would see that, of the six possible paths, only four are independent, as the other two are always a linear combination of the other four paths. Therefore, the number of independent paths is 4. In graph theory, it can be demonstrated that in a strongly connected graph (in which each node can be reached from any other node), the number of independent paths is given by

$$V(G) = e - n + 1$$

where n is the number of nodes and e is the number of arcs/edges.

However, it may be possible that the graph is not strongly connected. In that case, the above formula does not fit. Therefore, to make the graph strongly connected, we add an arc from the last node to the first node of the graph. In this way, the flow graph becomes a strongly connected graph. But by doing this, we increase the number of arcs by 1 and therefore, the number of independent paths (as a function of the original graph) is given by

$$V(G) = e - n + 2$$

This is called the *cyclomatic number* of a program. We can calculate the cyclomatic number only by knowing the number of choice points (decision nodes) d in the program. It is given by

$$V(G) = d + 1$$

This is also known as *Miller's theorem.* We assume that a k-way decision point contributes for $k-1$ choice points.

The program may contain several procedures also. These procedures can be represented as separate flow graphs. These procedures can be called from any point but the connections for calling are not shown explicitly. The cyclomatic number of the whole graph is then given by the sum of the numbers of each graph. It is easy to demonstrate that, if p is the number of graphs and e and n are referred to as the whole graph, the cyclomatic number is given by

$$V(G) = e - n + 2p$$

And Miller's theorem becomes

$$V(G) = d + p$$

Formulae Based on Cyclomatic Complexity

Based on the cyclomatic complexity, the following formulae are being summarized.

Cyclomatic complexity number can be derived through any of the following three formulae

1. $V(G) = e - n + 2p$
   where e is number of edges, n is the number of nodes in the graph, and p is number of components in the whole graph.
2. $V(G) = d + p$
   where d is the number of decision nodes in the graph.
3. $V(G) =$ number of regions in the graph

Calculating the number of decision nodes for Switch-Case/Multiple If-Else When a decision node has exactly two arrows leaving it, then we count it as a single decision node. However, switch-case and multiple if-else statements have more than two arrows leaving a decision node, and in these cases, the formula to calculate the number of nodes is

$d = k - 1$, where k is the number of arrows leaving the node.

Calculating the cyclomatic complexity number of the program having many connected components Let us say that a program P has three components: X, Y, and Z. Then we prepare the flow graph for P and for components, X, Y, and Z. The complexity number of the whole program is

$$V(G) = V(P) + V(X) + V(Y) + V(Z)$$

We can also calculate the cyclomatic complexity number of the full program with the first formula by counting the number of nodes and edges in all the components of the program collectively and then applying the formula

$$V(G) = e - n + 2P$$

The complexity number derived collectively will be same as calculated above. Thus,

$$V(P \cup X \cup Y \cup Z) = V(P) + V(X) + V(Y) + V(Z)$$

Guidelines for Basis Path Testing

We can use the cyclomatic complexity number in basis path testing. Cyclomatic number, which defines the number of independent paths, can be utilized as an upper bound for the number of tests that must be conducted to ensure that all the statements have been executed at least once. Thus, independent paths are prepared according to the upper limit of the cyclomatic number. The set of independent paths becomes the basis set for the flow graph of the program. Then test cases can be designed according to this basis set.

The following steps should be followed for designing test cases using path testing:

- Draw the flow graph using the code provided for which we have to write test cases.
- Determine the cyclomatic complexity of the flow graph.
- Cyclomatic complexity provides the number of independent paths. Determine a basis set of independent paths through the program control structure.
- The basis set is in fact the base for designing the test cases. Based on every independent path, choose the data such that this path is executed.

Example 5.1

Consider the following program segment:

```
main()
{
    int number, index;
1.  printf("Enter a number");
2.  scanf("%d, &number);
3.  index = 2;
4.  while(index <= number - 1)
5.  {
6.      if (number % index == 0)
7.      {
8.          printf("Not a prime number");
9.          break;
10.     }
11.     index++;
12.     }
```

```
13.     if(index == number)
14.         printf("Prime number");
15. } //end main
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all the methods.
(c) List all independent paths.
(d) Design test cases from independent paths.

Solution

(a) DD graph

For a DD graph, the following actions must be done:

- Put the line numbers on the execution statements of the program, as shown in Fig. 5.4. Start numbering the statements after declaring the variables, if no variables have been initialized. Otherwise, start from the statement where a variable has been initialized.

Figure 5.4 DD graph for Example 5.1

- Put the sequential statements in one node. For example, statements 1, 2, and 3 have been put inside one node.
- Put the edges between the nodes according to their flow of execution.
- Put alphabetical numbering on each node like *A*, *B*, etc.

The DD graph of the program is shown in Figure 5.4.

(b) Cyclomatic complexity

(i) $V(G) = e - n + 2 * p$

$= 10 - 8 + 2$

$= 4$

Figure 5.5 DD graph for Example 5.1 showing regions

(ii) $V(G) =$ Number of predicate nodes + 1

$= 3$ (Nodes *B*, *C*, and *F*) + 1

$= 4$

(iii) $V(G) = \text{Number of regions}$
$= 4(R1, R2, R3, R4)$

(c) Independent paths

Since the cyclomatic complexity of the graph is 4, there will be 4 independent paths in the graph as shown below:

(i) A-B-F-H
(ii) A-B-F-G-H
(iii) A-B-C-E-B-F-G-H
(iv) A-B-C-D-F-H

(d) Test case design from the list of independent paths

Test case ID	Input num	Expected result	Independent paths covered by test case
1	1	No output is displayed	A-B-F-H
2	2	Prime number	A-B-F-G-H
3	4	Not a prime number	A-B-C-D-F-H
4	3	Prime number	A-B-C-E-B-F-G-H

Example 5.2

Consider the following program that reads in a string and then checks the type of each character.

```
    main()
    {
        char string [80];
        int index;
1.      printf("Enter the string for checking its characters");
2.      scanf("%s", string);
3.      for(index = 0; string[index] != '\0'; ++index)   {
4.          if((string[index] >= '0' && (string[index] <='9'
5.                      printf("%c is a digit", string[index]);
6.          else if ((string[index] >= 'A' && string[index] <'Z'))  ||
                    ((string[index] >= 'a' && (string[index] <'z')))
7.                  printf("%c is an alphabet", string[index]);
8.          else
9.                  printf("%c is a special character", string[index]);
10.         }
11. }
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all the methods.
(c) List all independent paths.
(d) Design test cases from independent paths.

Solution

(a) DD graph

The DD graph of the program is shown in Fig. 5.6.

Figure 5.6 DD graph for Example 5.2

(b) Cyclomatic complexity

(i) $V(G) = e - n + 2 * P$
$= 10 - 8 + 2$
$= 4$

(ii) $V(G) =$ Number of predicate nodes + 1
$= 3$ (Nodes B, C) + 1
$= 4$

Node C is a multiple IF-THEN-ELSE, so for finding out the number of predicate nodes for this case, follow the following formula:

Number of predicated nodes

= Number of links out of main node − 1

= 3 − 1 = 2 (For node C)

(iii) $V(G)$ = Number of regions

= 4

(c) Independent paths

Since the cyclomatic complexity of the graph is 4, there will be 4 independent paths in the graph as shown below:

(i) *A-B-H*

(ii) *A-B-C-D-G-B-H*

(iii) *A-B-C-E-G-B-H*

(iv) *A-B-C-F-G-B-H*

(d) Test case design from the list of independent paths

Test Case ID	Input Line	Expected Output	Independent paths covered by Test case
1	0987	0 is a digit 9 is a digit 8 is a digit 7 is a digit	A-B-C-D-G-B-H A-B-H
2	AzxG	A is a alphabet z is a alphabet x is a alphabet G is a alphabet	A-B-C-E-G-B-H A-B-H
3	@#	@ is a special character # is a special character	A-B-C-F- G-B-H A-B-H

Example 5.3

Consider the following program:

```
    main()
    {
        char chr;
1.      printf ("Enter the special character\n");
2.      scanf (%c", &chr);
3.      if (chr != 48) && (chr != 49) && (chr != 50) && (chr != 51) &&
           (chr != 52) && (chr != 53) && (chr != 54) && (chr != 55) &&
           (chr != 56) && (chr != 57)
4.      {
5.          switch(chr)
```

```
6.          {
7.          Case '*': printf("It is a special character");
8.          break;
9.          Case '#': printf("It is a special character");
10.         break;
11.         Case '@': printf("It is a special character");
12.         break;
13.         Case '!':  printf("It is a special character");
14.         break;
15.         Case '%': printf("It is a special character");
16.         break;
17.         default : printf("You have not entered a special character");
18.         break;
19.         }// end of switch
20.     } // end of If
21.     else
22.         printf("You have not entered a character");
23. } // end of main()
```

(a) Draw the DD graph for the program.

(b) Calculate the cyclomatic complexity of the program using all the methods.

(c) List all independent paths.

(d) Design test cases from independent paths.

Solution

(a) DD graph

The DD graph of the program is shown in Fig. 5.7.

Figure 5.7 DD graph for Example 5.3

(b) Cyclomatic complexity

(i) $V(G) = e - n + 2p$
$= 17 - 12 + 2$
$= 7$

(ii) $V(G) =$ Number of predicate nodes + 1
$= 2$ (Nodes B, C) + 1
$= 7$

Node C is a switch-case, so for finding out the number of predicate nodes for this case, follow the following formula:

Number of predicated nodes
= Number of links out of main node –1
= 6 – 1 = 5 (For node C)

(iii) $V(G) =$ Number of regions = 7

(c) Independent paths

Since the cyclomatic complexity of the graph is 7, there will be 7 independent paths in the graph as shown below:

1. *A-B-D-L*
2. *A-B-C-E-K-L*
3. *A-B-C-F-K-L*
4. *A-B-C-G-K-L*
5. *A-B-C-H-K-L*
6. *A-B-C-I-K-L*
7. *A-B-C-J-K-L*

(d) Test Case Design from the list of Independent Paths

Test Case ID	Input Character	Expected Output	Independent path covered by Test Case
1	(	You have not entered a character	A-B-D-L
2	*	It is a special character	A-B-C-E-K-L
3	#	It is a special character	A-B-C-F-K-L
4	@	It is a special character	A-B-C-G-K-L
5	!	It is a special character	A-B-C-H-K-L
6	%	It is a special character	A-B-C-I-K-L
7	$	You have not entered a special character	A-B-C-J-K-L

Example 5.4

Consider a program to arrange numbers in ascending order from a given list of *N* numbers.

```
      main()
      {
      int num,small;
1.    int i,j,sizelist,list[10],pos,temp;
      clrscr();
      printf("\nEnter the size of list :\n ");
      scanf("%d",&sizelist);
2.    for(i=0;i<sizelist;i++)
      {
3.    printf("\nEnter the number");
      scanf ("%d",&list[i]);
      }
4.    for(i=0;i<sizelist;i++)
      {
5.    small=list[i];
      pos=i;
6.    for(j=i+1;j<sizelist;j++)
      {
7.    if(small>list[j])
      {
8.    small=list[j];
      pos=j;
      }
9.    }
      temp=list[i];
10.   list[i]=list[pos];
      list[pos]=temp;
11.   }
12.   printf("\nList of the numbers in ascending order : ");
13.   for(i=0;i<sizelist;i++)
14.   printf("\n%d",list[i]);
      getch();
15.   }
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all the methods.
(c) List all independent paths.
(d) Design test cases from independent paths.

Solution

(a) DD graph

The DD graph of the program is shown in Fig. 5.8.

Figure 5.8 DD graph for Example 5.4

(b) Cyclomatic complexity

1. $V(G) = e - n + 2p$

$= 19 - 15 + 2$

$= 6$

2. $V(G) = \text{Number of predicate nodes} + 1$
   $= 5 + 1$
   $= 6$
3. $V(G) = \text{Number of regions}$
   $= 6$

(c) Independent paths

Since the cyclomatic complexity of the graph is 6, there will be 6 independent paths in the graph as shown below:

1. 1-2-3-2-4-5-6-7-8-9-6-10-11-4-12-13-14-13-15
2. 1-2-3-2-4-5-6-7-9-6-10-11-4-12-13-14-13-15
3. 1-2-3-2-4-5-6-10-11-4-12-13-14-13-15
4. 1-2-3-2-4-12-13-14-13-15 (path not feasible)
5. 1-2-4-12-13-15
6. 1-2-3-2-4-12-13-15 (path not feasible)

(d) Test case design from the list of independent paths

Test Case ID	Input	Expected Output	Independent path covered by Test Case
1	Sizelist = 5 List[] = {17,6,7,9,1}	1,6,7,9,17	1
2	Sizelist = 5 List[] = {1,3,9,10,18}	1,3,9,10,18	2
3	Sizelist = 1 List[] = {1}	1	3
4	Sizelist = 0	blank	blank

Example 5.5

Consider the program for calculating the factorial of a number. It consists of main() program and the module fact(). Calculate the individual cyclomatic complexity number for main() and fact() and then, the cyclomatic complexity for the whole program.

```
        main()
        {
            int number;
            int fact();
1.          clrscr();
2.          printf("Enter the number whose factorial is to be found out");
```

```
3.        scanf("%d", &number);
4.        if(number <0)
5.           printf("Facorial cannot be defined for this number);
6.        else
7.           printf("Factorial is %d", fact(number));
8.    }

      int fact(int number)
      {
          int index;
1.        int product =1;
2.        for(index=1; index<=number; index++)
3.           product = product * index;
4.        return(product);
5.    }
```

Solution

DD graph

The DD graph of the program is shown in Fig. 5.9

Figure 5.9

Cyclomatic complexity of main()

(a) $V(M) = e - n + 2p$

$= 5 - 5 + 2$

$= 2$

(b) $V(M) = d + p$

$= 1 + 1$

$= 2$

(c) $V(M) = \text{Number of regions} = 2$

Cyclomatic complexity of fact()

(a) $V(R) = e - n + 2p$
$= 4 - 4 + 2$
$= 2$

(b) $V(R) =$ Number of predicate nodes + 1
$= 1 + 1$
$= 2$

(c) $V(R) =$ Number of regions $= 2$

Cyclomatic complexity of the whole graph considering the full program

(a) $V(G) = e - n + 2p$
$= 9 - 9 + 2 \times 2$
$= 4$
$= V(M) + V(R)$

(b) $V(G) = d + p$
$= 2 + 2$
$= 4$
$= V(M) + V(R)$

(c) $V(G) =$ Number of regions
$= 4$
$= V(M) + V(R)$

5.3.5 Applications of Path Testing

Path testing has been found better suitable as compared to other testing methods. Some of its applications are discussed below.

Thorough testing / More coverage Path testing provides us the best code coverage, leading to a thorough testing. Path coverage is considered better as compared to statement or branch coverage methods because the basis path set provides us the number of test cases to be covered which ascertains the number of test cases that must be executed for full coverage. Generally, branch coverage or other criteria gives us less number of test cases as compared to path testing. Cyclomatic complexity along with basis path analysis employs more comprehensive scrutiny of code structure and control flow, providing a far superior coverage technique.

Unit testing Path testing is mainly used for structural testing of a module. In unit testing, there are chances of errors due to interaction of decision outcomes or control flow problems which are hidden with branch testing. Since each decision outcome is tested independently, path testing uncovers these errors in module testing and prepares them for integration.

Integration testing Since modules in a program may call other modules or be called by some other module, there may be chances of interface errors during calling of the modules. Path testing analyses all the paths on the interface and explores all the errors.

Maintenance testing Path testing is also necessary with the modified version of the software. If you have earlier prepared a unit test suite, it should be run on the modified software or a selected path testing can be done as a part of regression testing. In any case, path testing is still able to detect any security threats on the interface with the called modules.

Testing effort is proportional to complexity of the software Cyclomatic complexity number in basis path testing provides the number of tests to be executed on the software based on the complexity of the software. It means the number of tests derived in this way is directly proportional to the complexity of the software. Thus, path testing takes care of the complexity of the software and then derives the number of tests to be carried out.

Basis path testing effort is concentrated on error-prone software Since basis path testing provides us the number of tests to be executed as a measure of software cyclomatic complexity, the cyclomatic number signifies that the testing effort is only on the error-prone part of the software, thus minimizing the testing effort.

5.4 Graph Matrices

Flow graph is an effective aid in path testing as seen in the previous section. However, path tracing with the use of flow graphs may be a cumbersome and time-consuming activity. Moreover, as the size of graph increases, manual path tracing becomes difficult and leads to errors. A link can be missed or covered twice. So the idea is to develop a software tool which will help in basis path testing.

Graph matrix, a data structure, is the solution which can assist in developing a tool for automation of path tracing. The reason being the properties of graph matrices are fundamental to test tool building. Moreover, testing theory

can be explained on the basis of graphs. Graph theorems can be proved easily with the help of graph matrices. So graph matrices are very useful for understanding the testing theory.

5.4.1 Graph Matrix

A graph matrix is a square matrix whose rows and columns are equal to the number of nodes in the flow graph. Each row and column identifies a particular node and matrix entries represent a connection between the nodes.

The following points describe a graph matrix:

- Each cell in the matrix can be a direct connection or link between one node to another node.
- If there is a connection from node 'a' to node 'b', then it does not mean that there is connection from node 'b' to node 'a'.
- Conventionally, to represent a graph matrix, digits are used for nodes and letter symbols for edges or connections.

Example 5.6

Consider the above graph and represent it in the form of a graph matrix.

Solution

The graph matrix is shown below.

	1	2	3	4
1	a	b	c	
2				d
3				e
4				

Example 5.7

Consider the graph and represent it in the form of a graph matrix.

Solution

The graph matrix is shown below.

	1	2	3	4
1		a+b	c	
2				
3				d
4				

5.4.2 Connection Matrix

Till now, we have learnt how to represent a flow graph into a matrix representation. But this matrix is just a tabular representation and does not provide any useful information. If we add link weights to each cell entry, then graph matrix can be used as a powerful tool in testing. The links between two nodes are assigned a link weight which becomes the entry in the cell of matrix. The link weight provides information about control flow.

In the simplest form, when the connection exists, then the link weight is 1, otherwise 0 (But 0 is not entered in the cell entry of matrix to reduce the complexity). A matrix defined with link weights is called a connection matrix. The connection matrix for Example 5.6 is shown below.

	1	2	3	4
1	1	1	1	
2				1
3				1
4				

The connection matrix for Example 5.7 is shown below.

	1	2	3	4
1		1	1	
2				
3				1
4				

5.4.3 Use of Connection Matrix in Finding Cyclomatic Complexity Number

Connection matrix is used to see the control flow of the program. Further, it is used to find the cyclomatic complexity number of the flow graph. Given below is the procedure to find the cyclomatic number from the connection matrix:

Step 1: For each row, count the number of 1s and write it in front of that row.

Step 2: Subtract 1 from that count. Ignore the blank rows, if any.

Step 3: Add the final count of each row.

Step 4: Add 1 to the sum calculated in Step 3.

Step 5: The final sum in Step 4 is the cyclomatic number of the graph.

The cyclomatic number calculated from the connection matrix of Example 5.6 is shown below.

	1	2	3	4	
1	1	1	1		**3 – 1 = 2**
2				1	**1 – 1 = 0**
3				1	**1 – 1 = 0**
4					
	Cyclomatic number = 2+1 = 3				

The cyclomatic number calculated from the connection matrix of Example 5.7 is shown below.

	1	2	3	4	
1		1	1		**2 – 1 = 1**
2					
3				1	**1 – 1 = 0**
4					
	Cyclomatic number = 1+1 = 2				

5.4.4 Use of Graph Matrix for Finding Set of all Paths

Another purpose of developing graph matrices is to produce a set of all paths between all nodes. It may be of interest in path tracing to find *k*-link paths from one node. For example, how many 2-link paths are there from one node to another node? This process is done for every node resulting in the set of all paths. This set can be obtained with the help of matrix operations. The main objective is to use matrix operations to obtain the set of all paths between all nodes. The set of all paths between all nodes is easily expressed in terms of matrix operations.

The power operation on matrix expresses the relation between each pair of nodes via intermediate nodes under the assumption that the relation is transitive (mostly, relations used in testing are transitive). For example, the square of matrix represents path segments that are 2-links long. Similarly, the cube power of matrix represents path segments that are 3-links long.

Generalizing, we can say that mth power of the matrix represents path segments that are m-links long.

Example 5.8

Consider the graph matrix in Example 5.6 and find 2-link paths for each node.

Solution

For finding 2-link paths, we should square the matrix. Squaring the matrix yields a new matrix having 2-link paths.

$$\begin{pmatrix} a & b & c & 0 \\ 0 & 0 & 0 & d \\ 0 & 0 & 0 & e \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} a & b & c & 0 \\ 0 & 0 & 0 & d \\ 0 & 0 & 0 & e \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} a^2 & ab & ac & bd+ce \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

The resulting matrix shows all the 2-link paths from one node to another. For example, from node 1 to node 2, there is one 2-link, i.e., *ab*.

Example 5.9

Consider the following graph. Derive its graph matrix and find 2-link and 3-link set of paths.

Solution

The graph matrix of the graph is shown below.

$$\begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

First we find 2-link set of paths by squaring this matrix as shown below:

$$\begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & bd & ac & ae+bf \\ 0 & cd & 0 & cf \\ 0 & 0 & dc & de \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Next, we find 3-link set of paths by taking the cube of matrix as shown below:

$$\begin{pmatrix} 0 & bd & ac & ae+bf \\ 0 & cd & 0 & cf \\ 0 & 0 & dc & de \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & acd & bdc & bde+acf \\ 0 & 0 & cdc & cde \\ 0 & dcd & 0 & dcf \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

It can be generalized that for n number of nodes, we can get the set of all paths of $(n-1)$ links length with the use of matrix operations. These operations can be programmed and can be utilized as a software testing tool.

5.5 Loop Testing

Loop testing can be viewed as an extension to branch coverage. Loops are important in the software from the testing viewpoint. If loops are not tested

properly, bugs can go undetected. This is the reason that loops are covered in this section exclusively. Loop testing can be done effectively while performing development testing (unit testing by the developer) on a module. Sufficient test cases should be designed to test every loop thoroughly.

There are four different kinds of loops. How each kind of loop is tested, is discussed below.

Simple loops Simple loops mean, we have a single loop in the flow, as shown in Fig. 5.9.

Figure 5.9 (a)

Figure 5.9 (b)

The following test cases should be considered for simple loops while testing them [9]:

- Check whether you can bypass the loop or not. If the test case for bypassing the loop is executed and, still you enter inside the loop, it means there is a bug.
- Check whether the loop control variable is negative.
- Write one test case that executes the statements inside the loop.
- Write test cases for a typical number of iterations through the loop.
- Write test cases for checking the boundary values of the maximum and minimum number of iterations defined (say min and max) in the loop. It means we should test for min, min+1, min—1, max—1, max, and max+1 number of iterations through the loop.

Nested loops When two or more loops are embedded, it is called a nested loop, as shown in Fig. 5.10. If we have nested loops in the program, it becomes difficult to test. If we adopt the approach of simple tests to test the nested loops, then the number of possible test cases grows geometrically. Thus, the strategy is to start with the innermost loops while holding outer loops to their

minimum values. Continue this outward in this manner until all loops have been covered [9].

Figure 5.10 Nested loops

Concatenated loops The loops in a program may be concatenated (Fig. 5.11). Two loops are concatenated if it is possible to reach one after exiting the other, while still on a path from entry to exit. If the two loops are not on the same path, then they are not concatenated. The two loops on the same path may or may not be independent. If the loop control variable for one loop is used for another loop, then they are concatenated, but nested loops should be treated like nested only.

Figure 5.11 Concatenated loops

Unstructured loops This type of loops is really impractical to test and they must be redesigned or at least converted into simple or concatenated loops.

5.6 Data Flow Testing

In path coverage, the stress was to cover a path using statement or branch coverage. However, data and data integrity is as important as code and code integrity of a module. We have checked every possibility of the control flow of a module. But what about the data flow in the module? Has every data object been initialized prior to use? Have all defined data objects been used for something? These questions can be answered if we consider data objects in the control flow of a module.

Data flow testing is a white-box testing technique that can be used to detect improper use of data values due to coding errors. Errors may be unintentionally introduced in a program by programmers. For instance, a programmer might use a variable without defining it. Moreover, he may define a variable, but not initialize it and then use that variable in a predicate. For example,

```
int a;
if(a == 67) { }
```

In this way, data flow testing gives a chance to look out for inappropriate data definition, its use in predicates, computations, and termination. It identifies potential bugs by examining the patterns in which that piece of data is used. For example, if an out-of-scope data is being used in a computation, then it is a bug. There may be several patterns like this which indicate data anomalies.

To examine the patterns, the control flow graph of a program is used. This test strategy selects the paths in the module's control flow such that various sequences of data objects can be chosen. The major focus is on the points at which the data receives values and the places at which the data initialized has been referenced. Thus, we have to choose enough paths in the control flow to ensure that every data is initialized before use and all the defined data have been used somewhere. Data flow testing closely examines the state of the data in the control flow graph, resulting in a richer test suite than the one obtained from control flow graph based path testing strategies like branch coverage, all statement coverage, etc.

5.6.1 State of a Data Object

A data object can be in the following states:

- ***Defined (d)*** A data object is called *defined* when it is initialized, i.e. when it is on the left side of an assignment statement. Defined state can also be used to mean that a file has been opened, a dynamically allocated object has been allocated, something is pushed onto the stack, a record written, and so on [9].

- ***Killed/Undefined/Released (k)*** When the data has been reinitialized or the scope of a loop control variable finishes, i.e. exiting the loop or memory is released dynamically or a file has been closed.
- ***Usage (u)*** When the data object is on the right side of assignment or used as a control variable in a loop, or in an expression used to evaluate the control flow of a case statement, or as a pointer to an object, etc. In general, we say that the usage is either *computational use* (c-use) or *predicate use* (p-use).

5.6.2 Data-Flow Anomalies

Data-flow anomalies represent the patterns of data usage which may lead to an incorrect execution of the code. An anomaly is denoted by a two-character sequence of actions. For example, 'dk' means a variable is defined and killed without any use, which is a potential bug. There are nine possible two-character combinations out of which only four are data anomalies, as shown in Table 5.1.

Table 5.1 Two-character data-flow anomalies

Anomaly	Explanation	Effect of Anomaly
du	Define-use	Allowed. Normal case.
dk	**Define-kill**	**Potential bug. Data is killed without use after definition.**
ud	Use-define	Data is used and then redefined. Allowed. Usually not a bug because the language permits reassignment at almost any time.
uk	Use-kill	Allowed. Normal situation.
ku	**Kill-use**	**Serious bug because the data is used after being killed.**
kd	Kill-define	Data is killed and then redefined. Allowed.
dd	**Define-define**	**Redefining a variable without using it. Harmless bug, but not allowed.**
uu	Use-use	Allowed. Normal case.
kk	**Kill-kill**	**Harmless bug, but not allowed.**

It can be observed that not all data-flow anomalies are harmful, but most of them are suspicious and indicate that an error can occur. In addition to the above two-character data anomalies, there may be single-character data anomalies also. To represent these types of anomalies, we take the following conventions:

$\sim x$: indicates all prior actions are not of interest to x.

$x\sim$: indicates all post actions are not of interest to x.

All single-character data anomalies are listed in Table 5.2.

Table 5.2 Single-character data-flow anomalies

Anomaly	Explanation	Effect of Anomaly
~d	First definition	Normal situation. Allowed.
~u	**First Use**	**Data is used without defining it. Potential bug.**
~k	**First Kill**	**Data is killed before defining it. Potential bug.**
D~	**Define last**	**Potential bug.**
U~	Use last	Normal case. Allowed.
K~	Kill last	Normal case. Allowed.

5.6.3 Terminology Used in Data Flow Testing

In this section, some terminology [9,20], which will help in understanding all the concepts related to data-flow testing, is being discussed. Suppose P is a program that has a graph $G(P)$ and a set of variables V. The graph has a single entry and exit node.

Definition node Defining a variable means assigning value to a variable for the very first time in a program. For example, input statements, assignment statements, loop control statements, procedure calls, etc.

Usage node It means the variable has been used in some statement of the program. Node n that belongs to $G(P)$ is a usage node of variable v, if the value of variable v is used at the statement corresponding to node n. For example, output statements, assignment statements (right), conditional statements, loop control statements, etc.

A usage node can be of the following two types:

(i) Predicate Usage Node: If usage node n is a predicate node, then n is a predicate usage node.

(ii) Computation Usage Node: If usage node n corresponds to a computation statement in a program other than predicate, then it is called a computation usage node.

Loop-free path segment It is a path segment for which every node is visited once at most.

Simple path segment It is a path segment in which at most one node is visited twice. A simple path segment is either loop-free or if there is a loop, only one node is involved.

Definition-use path (du-path) A du-path with respect to a variable v is a path between the definition node and the usage node of that variable. Usage node can either be a p-usage or a c-usage node.

Definition-clear path(dc-path) A dc-path with respect to a variable *v* is a path between the definition node and the usage node such that no other node in the path is a defining node of variable *v*.

The du-paths which are not dc-paths are important from testing viewpoint, as these are potential problematic spots for testing persons. Those du-paths which are definition-clear are easy to test in comparison to du-paths which are not dc-paths. The application of data flow testing can be extended to debugging where a testing person finds the problematic areas in code to trace the bug. So the du-paths which are not dc-paths need more attention.

5.6.4 Static Data Flow Testing

With static analysis, the source code is analysed without executing it. Let us consider an example of an application given below.

Example 5.10

Consider the program given below for calculating the gross salary of an employee in an organization. If his basic salary is less than Rs 1500, then HRA = 10% of basic salary and DA = 90% of the basic. If his salary is either equal to or above Rs 1500, then HRA = Rs 500 and DA = 98% of the basic salary. Calculate his gross salary.

```
    main()
    {
1.      float bs, gs, da, hra = 0;
2.      printf("Enter basic salary");
3.      scanf("%f", &bs);
4.      if(bs < 1500)
5.      {
6.          hra = bs * 10/100;
7.          da = bs * 90/100;
8.      }
9.      else
10.     {
11.         hra = 500;
12.         da = bs * 98/100;
13.     }
14.     gs = bs + hra + da;
15.     printf("Gross Salary = Rs. %f", gs);
16. }
```

Find out the define-use-kill patterns for all the variables in the source code of this application.

Solution

For variable 'bs', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	3	Normal case. Allowed
du	3-4	Normal case. Allowed
uu	4-6, 6-7, 7-12, 12-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

For variable 'gs', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	14	Normal case. Allowed
du	14-15	Normal case. Allowed
uk	15-16	Normal case. Allowed
K~	16	Normal case. Allowed

For variable 'da', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	7	Normal case. Allowed
du	7-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

For variable 'hra', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	1	Normal case. Allowed
dd	1-6 or 1-11	Double definition. Not allowed. Harmless bug.
du	6-14 or 11-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

From the above static analysis, it was observed that static data flow testing for the variable 'hra' discovered one bug of double definition in line number 1.

Static Analysis is not Enough

It is not always possible to determine the state of a data variable by just static analysis of the code. For example, if the data variable in an array is used as an index for a collection of data elements, we cannot determine its state by static analysis. Or it may be the case that the index is generated dynamically during execution, therefore we cannot guarantee what the state of the array element is referenced by that index. Moreover, the static data-flow testing might denote a certain piece of code to be anomalous which is never executed and hence, not completely anomalous. Thus, all anomalies using static analysis cannot be determined and this problem is provably unsolvable.

5.6.5 Dynamic Data Flow Testing

Dynamic data flow testing is performed with the intention to uncover possible bugs in data usage during the execution of the code. The test cases are designed in such a way that every definition of data variable to each of its use is traced and every use is traced to each of its definition. Various strategies are employed for the creation of test cases. All these strategies are defined below.

All-du Paths (ADUP) It states that every du-path from every definition of every variable to every use of that definition should be exercised under some test. It is the strongest data flow testing strategy, since it is a superset of all other data flow testing strategies. Moreover, this strategy requires the maximum number of paths for testing.

All-uses (AU) This states that for every use of the variable, there is a path from the definition of that variable (nearest to the use in backward direction) to the use.

All-p-uses/Some-c-uses (APU + C) This strategy states that for every variable and every definition of that variable, include at least one dc-path from the definition to every predicate use. If there are definitions of the variable with no p-use following it, then add computational use (c-use) test cases as required to cover every definition.

All-c-uses/Some-p-uses (ACU + P) This strategy states that for every variable and every definition of that variable, include at least one dc-path from the

definition to every computational use. If there are definitions of the variable with no c-use following it, then add predicate use (c-use) test cases as required to cover every definition.

All-Predicate-Uses (APU) It is derived from the APU+C strategy and states that for every variable, there is a path from every definition to every p-use of that definition. If there is a definition with no p-use following it, then it is dropped from contention.

All-Computational-Uses (ACU) It is derived from the strategy ACU+P strategy and states that for every variable, there is a path from every definition to every c-use of that definition. If there is a definition with no c-use following it, then it is dropped from contention.

All-Definition (AD) It states that every definition of every variable should be covered by at least one use of that variable, be that a computational use or a predicate use.

Example 5.11

Consider the program given below. Draw its control flow graph and data flow graph for each variable used in the program, and derive data flow testing paths with all the strategies discussed above.

```
    main()
    {
    int work;
 0. double payment =0;
 1. scanf("%d", work);
 2. if (work > 0) {
 3.     payment = 40;
 4. if (work > 20)
 5. {
 6.     if(work <= 30)
 7.         payment = payment + (work - 25) * 0.5;
 8.     else
 9.     {
10.         payment = payment + 50 + (work -30) * 0.1;
11.             if (payment >= 3000)
12.                 payment = payment * 0.9;
13.     }
14. }
15. }
16. printf("Final payment", payment);
```

Solution

Figure 5.12 shows the control flow graph for the given program.

Figure 5.12 DD graph for Example 5.11

Figure 5.13 shows the data flow graph for the variable 'payment'.

Figure 5.13 Data flow graph for 'payment'

Figure 5.14 shows the data flow graph for the variable 'work'.

Figure 5.14 Data flow graph for variable 'work'

Prepare a list of all the definition nodes and usage nodes for all the variables in the program.

Variable	Defined At	Used At
Payment	0,3,7,10,12	7,10,11,12,16
Work	1	2,4,6,7,10

Data flow testing paths for each variable are shown in Table 5.3.

Table 5.3 Data flow testing paths

Strategy	Payment	Work
All Uses(AU)	3-4-5-6-7 10-11 10-11-12 12-13-14-15-16 3-4-5-6-8-9-10	1-2 1-2-3-4 1-2-3-4-5-6 1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10
All p-uses (APU)	0-1-2-3-4-5-6-8-9-10-11	1-2 1-2-3-4 1-2-3-4-5-6

All c-uses (ACU)	0-1-2-16 3-4-5-6-7 3-4-5-6-8-9-10 3-4-15-16 7-14-15-16 10-11-12 10-11-13-14-15-16 12-13-14-15-16	1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10
All-p-uses / Some-c-uses (APU + C)	0-1-2-3-4-5-6-8-9-10-11 10-11-12 12-13-14-15-16	1-2 1-2-3-4 1-2-3-4-5-6 1-2-3-4-5-6-8-9-10
All-c-uses / Some-p-uses (ACU + P)	0-1-2-16 3-4-5-6-7 3-4-5-6-8-9-10 3-4-15-16 7-14-15-16 10-11-12 10-11-13-14-15-16 12-13-14-15-16 0-1-2-3-4-5-6-8-9-10-11	1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10 1-2-3-4-5-6
All-du-paths (ADUP)	0-1-2-3-4-5-6-8-9-10-11 0-1-2-16 3-4-5-6-7 3-4-5-6-8-9-10 3-4-15-16 7-14-15-16 10-11-12 10-11-13-14-15-16 12-13-14-15-16	1-2 1-2-3-4 1-2-3-4-5-6 1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10
All Definitions (AD)	0-1-2-16 3-4-5-6-7 7-14-15-16 10-11 12-13-14-15-16	1-2

5.6.6 Ordering of Data Flow Testing Strategies

While selecting a test case, we need to analyse the relative strengths of various data flow testing strategies. Figure 5.15 depicts the relative strength of the data flow strategies. In this figure, the relative strength of testing strategies reduces

along the direction of the arrow. It means that all-du-paths (ADPU) is the strongest criterion for selecting the test cases.

Figure 5.15 Data-flow testing strategies

5.7 Mutation Testing

Mutation testing is the process of mutating some segment of code (putting some error in the code) and then, testing this mutated code with some test data. If the test data is able to detect the mutations in the code, then the test data is quite good, otherwise we must focus on the quality of test data. Therefore, mutation testing helps a user create test data by interacting with the user to iteratively strengthen the quality of test data.

During mutation testing, faults are introduced into a program by creating many versions of the program, each of which contains one fault. Test data are used to execute these faulty programs with the goal of causing each faulty program to fail. Faulty programs are called *mutants* of the original program and a mutant is said to be *killed* when a test case causes it to fail. When this happens, the mutant is considered *dead* and no longer needs to remain in the testing process, since the faults represented by that mutant have been detected, and more importantly, it has satisfied its requirement of identifying a useful test case. Thus, the main objective is to select efficient test data which have error-detection power. The criterion for this test data is to differentiate the initial program from the mutant. This distinguish-ability between the initial program and its mutant will be based on test results.

5.7.1 Primary Mutants

Let us take an example of a C program to understand primary mutants.

```
...
if (a > b)
    x = x + y;
else
    x = y;
printf("%d", x);
...
```

We can consider the following mutants for the above example:

M1: `x = x - y;`
M2: `x = x / y;`
M3: `x =  x + 1;`
M4: `printf("%d", y);`

When the mutants are single modifications of the initial program using some operators as shown above, they are called *primary mutants*. Mutation operators are dependent on programming languages. Note that each of the mutated statements represents a separate program. The results of the initial program and its mutants are shown below.

Test Data	x	y	Initial Program Result	Mutant Result
TD1	2	2	4	0 (M1)
TD2(x and y # 0)	4	3	7	1.4 (M2)
TD3 (y #1)	3	2	5	4 (M3)
TD4(y #0)	5	2	7	2 (M4)

5.7.2 Secondary Mutants

Let us take another example program as shown below:

```
if (a < b)
    c = a;
```

Now, mutants for this code may be as follows:

```
M1 : if (a <= b-1)
                c = a;
M2: if (a+1 <= b)
                c = a;
M3: if (a == b)
                c = a+1;
```

This class of mutants is called *secondary mutants* when multiple levels of mutation are applied on the initial program. In this case, it is very difficult to identify the initial program from its mutants.

Example 5.12

Consider the program *P* shown below.

```
r = 1;
for (i = 2; i<=3; ++i) {
    if (a[i] > a[r]
        r = i;
}
```

The mutants for *P* are:

M1:

```
r = 1;
for (i = 1; i<=3; ++i) {
    if (a[i] > a[r])
        r = i;
}
```

M2:

```
r = 1;
for (i = 2; i<=3; ++i) {
    if (i > a[r])
        r = i;
}
```

M3:

```
r = 1;
for (i = 2; i<=3; ++i) {
    if (a[i] >= a[r])
        r = i;
}
```

M4:

```
r = 1;
for (i = 1; i<=3; ++i) {
    if (a[r] > a[r])
        r = i;
}
```

Let us consider the following test data selection:

	a[1]	a[2]	a[3]
TD1	1	2	3
TD2	1	2	1
TD3	3	1	2

We apply these test data to mutants, M1, M2, M3, and M4.

	P	M1	M2	M3	M4	Killed Mutants
TD1	3	3	3	3	1	M4
TD2	2	2	3	2	1	M2 and M4
TD3	1	1	1	1	1	none

We need to look at the efficiency of the proposed test data. It can be seen that M1 and M3 are not killed by the test data selection. Therefore, the test data is incomplete. Therefore, we need to add a new test data, TD4 = {2,2,1}, then this test data kills M3.

5.7.3 Mutation Testing Process

The mutation testing process is discussed below:

- Construct the mutants of a test program.
- Add test cases to the mutation system and check the output of the program on each test case to see if it is correct.
- If the output is incorrect, a fault has been found and the program must be modified and the process restarted.
- If the output is correct, that test case is executed against each live mutant.
- If the output of a mutant differs from that of the original program on the same test case, the mutant is assumed to be incorrect and is killed.
- After each test case has been executed against each live mutant, each remaining mutant falls into one of the following two categories.
    - One, the mutant is functionally equivalent to the original program. An equivalent mutant always produces the same output as the original program, so no test case can kill it.
    - Two, the mutant is killable, but the set of test cases is insufficient to kill it. In this case, new test cases need to be created, and the process iterates until the test set is strong enough to satisfy the tester.

- The mutation score for a set of test data is the percentage of non-equivalent mutants killed by that data. If the mutation score is 100%, then the test data is called *mutation-adequate.*

SUMMARY

White-box testing is a dynamic testing category under which the software is tested using the structure or logic of the program under test. This technique is closer to a developer who needs to test his module while developing it. The developer cannot design and code the module in one go and therefore he is supposed to verify his logic using various test case design methods.

White-box testing is necessary, as it is one of the steps to verify the correctness of the program and consequently in enhancing the quality of the program. All the methods of white-box testing have been discussed in this chapter, taking sufficient number of examples such that a new method can be learnt easily and the reader is ready to design the test cases using a particular method.

Let us review the important concepts described in this chapter:

- White-box testing is the technique wherein it is required to understand the structure and logic of the program to test the software.
- White-box testing is largely done by the developer of the software.
- White-box testing covers the logic of the program. The forms of logic coverage are: statement coverage, decision/branch coverage, condition coverage, decision-condition coverage, and multiple condition.
- Basis path testing is the technique of selecting the paths that provide a basis set of execution paths through the program.
- The basis set is derived by calculating the cyclomatic complexity number. Having the knowledge of this number, independent paths are derived from the flow graph of the program. Corresponding to independent paths, test cases are prepared which form the basis set.
- An independent path is any path through the graph that introduces at least one new set of processing statements or new conditions.
- Cyclomatic complexity is the number which tells us about the complexity of the design of a module. It should be less than 10, otherwise the module should be redesigned.
- Cyclomatic complexity of a module can be calculated by the following methods:
  - $V(G) = e - n + 2P$

    where e is number of edges, n is the number of nodes in the graph, and P is the number of components in the whole graph
  - $V(G) = d + P$

    where d is the number of decision nodes in the graph.
  - $V(G)$ = number of regions in the graph
  - Using graph matrices
- Graph matrix, a data structure, can assist in developing a tool for automation of path tracing.

- A graph matrix is a square matrix whose rows and columns are equal to the number of nodes in the flow graph.
- A matrix defined with link weights is called a connection matrix.
- The connection matrix is used to find the cyclomatic complexity number of the flow graph.
- Graph matrices are also used to produce a set of all paths between all nodes. An *m*th power of the matrix represents all path segments *m* links long.
- Data flow testing is a white-box testing technique that can be used to detect improper use of data values due to coding errors.
- Data flow anomalies represent the patterns of data usage which may lead to an incorrect execution of the code.
- Definition node in a data flow graph is the node wherein a variable is assigned a value for the very first time in the program.
- Usage node in a data flow graph is the node wherein the variable has already been used in some statement of the program.
- A du-path with respect to a variable *v* is a path between the definition node and the usage node of that variable.
- A dc-path with respect to a variable *v* is a path between the definition node and the usage node such that no other node in the path is a defining node of variable *v*.
- du-paths which are not definition-clear paths are important from testing viewpoint, as these are potential problematic spots for testing persons. du-paths which are definition-clear are easy to test in comparison to those which are not dc-paths.
- The application of data flow testing can be extended to debugging where a testing person finds the problematic areas in the code to trace the bug. So the du-paths which are not definition-clear need more attention of the tester.
- During mutation testing, faults are introduced into a program by creating many versions of the program, each of which contains a fault. Test data are used to execute these faulty programs with the goal of causing each faulty program to fail. Faulty programs are called mutants of the original program and a mutant is said to be killed when a test case causes it to fail.
- When the mutants are single modifications of the initial program using some operators, they are called primary mutants.
- When multiple levels of mutation are applied on the initial program, they are called secondary mutants.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. White-box testing is _____ to black-box testing.
   (a) mutually exclusive
   (b) complementary
   (c) not related

2. The effectiveness of path testing rapidly _____ as the size of software under test.
   (a) decreases
   (b) increases
   (c) does not change
   (d) none of the above
3. A node with more than one arrow leaving it is called a _____
   (a) decision node
   (b) junction node
   (c) region
   (d) all of the above
4. A node with more than one arrow entering it is called a _____
   (a) decision node
   (b) junction node
   (c) region
   (d) all of the above
5. Areas bounded by edges and nodes are called _____
   (a) decision node
   (b) junction node
   (c) region
   (d) all of the above
6. The length of a path is measured by the number of _____
   (a) instructions
   (b) junction nodes
   (c) decision nodes
   (d) links
7. An independent path is any path through the graph that introduces at least _____ new set of processing statements or new conditions.
   (a) 4
   (b) 3
   (c) 1
   (d) 2
8. The number of independent paths is given by _____.
   (a) $V(G) = e - n + 1$
   (b) $V(G) = 2e - n + 1$
   (c) $V(G) = e - n + 2$
   (d) none of the above
9. According to Mill's Theorem, _____
   (a) $V(G) = d + 2P$

(b) $V(G) = d + P$
(c) $V(G) = 2d + P$
(d) None of the above

10. In data flow anomalies, dd is a _____
    (a) serious bug
    (b) normal case
    (c) harmless bug
    (d) none of the above
11. In data flow anomalies, du is a _____
    (a) serious bug
    (b) normal case
    (c) harmless bug
    (d) none of the above
12. In data flow anomalies, ku is a _____
    (a) serious bug
    (b) normal case
    (c) harmless bug
    (d) none of the above
13. In single-character data anomalies, ~d is _____
    (a) potential bug
    (b) normal situation
    (c) none of the above
15. In single-character data anomalies, ~k is _____
    (a) potential bug
    (b) normal situation
    (c) none of the above
16. _____ is the strongest criterion for selecting test cases.
    (a) AD
    (b) APU
    (c) AU
    (d) ADPU

Review Questions

1. What is the need of white-box testing?
2. What are the different criteria for logic coverage?
3. What is basis path testing?

4. Distinguish between decision node and junction node?
5. What is an independent path?
6. What is the significance of cyclomatic complexity?
7. How do you calculate the number of decision nodes for switch-case?
8. How do you calculate the cyclomatic complexity number of the program having many connected components?
9. Consider the program.

```
#include <stdio.h>
main()
{
        int a,b, c,d;
        clrscr();
        printf("enter the two variables a,b");
        scanf("%d %d",&a,&b);
        printf("enter the option 1:Addition,
               2:subtraction,3:multiplication,4:division");
        scanf("%d",&c);
        switch(c)
        {
                case 1:d = a+b;
                printf("Addition of two no.=%d", d);
                break;
                case 2:d = a-b;
                printf("Subtraction of two no.=%d", d);
                break;
                case 3:d = a*b;
                printf("Multiplication of two no.=%d", d);
                break;
                case 4:d = a/b;
                printf("division of two no.=%d",d);
                break;
        }
}
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all four methods.
(c) List all independent paths.
(d) Design all test cases from independent paths.
(e) Derive all du-paths and dc-paths using data flow testing.

10. Consider the program to find the greatest number:

```
#include <stdio.h>
main()
{
        float x,y,z;
        clrscr();
        printf("enter the three variables x,y,z");
        scanf("%f %f %f",&x,&y,&z);
        if(x > y)
        {
            if(x > z)
                printf("x is greatest");
            else
                printf("z is greatest");
        }
        else
        {
            if(y > z)
                printf("y is greatest");
            else
                printf("z is greatest");
        }
    getch();
}
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all four methods.
(c) List all independent paths.
(d) Design all test cases from independent paths.
(e) Derive all du-paths and dc-paths using data flow testing.

11. Consider the following program which multiplies two matrices:

```
#include <stdio.h>
main()
{
int a[SIZE][SIZE], b[SIZE][SIZE], c[SIZE][SIZE], i, j, k, rowl,
colml, row2, colm2;

printf("Enter the order of first matrix <= %d %d \n", SIZE, SIZE);
scanf("%d%d",&row1, colm1);
printf("Enter the order of second matrix <= %d %d \n", SIZE, SIZE);
scanf("%d%d",&row2, colm2);
if(colm1==row2)
```

```
        {
        printf("Enter first matrix");
        for(i=0; i<row1; i++)
                {
                for(j=0; j<colml; j++)
                scanf("%d", &a[i][j]);
                }
        printf("Enter second matrix");
        for(i=0; i<row2; i++)
                {
                for(j=0; j<colm2; j++)
                scanf("%d", &b[i][j]);
                }
        printf("Multiplication of two matrices is");
        for(i=0; i<row1; i++)
                {
                for(j=0; j<colm1; j++)
                        {
                        c[i] [j] = 0;
                        for(k=0; k<row2; k++)
                        c[i][j]+ = a[i][k] + b[k][j];
                        printf("%6d", c[i][j]);
                        }
                }
        }
        else
        {
                printf("Matrix multiplication is not possible");
        }
}
```

(a) Draw the DD graph for the program.

(b) Calculate the cyclomatic complexity of the program using all four methods.

(c) List all independent paths.

(d) Design all test cases from the independent paths.

(e) Derive all du-paths and dc-paths using data flow testing.

12. Consider the following program for finding the prime numbers, their sum, and count:

```
main()
{
        int num, flag, sum, count;
        int CheckPrime(int n);
        sum = count = 0;
        printf("Prime number between 1 and 100 are");
        for(num=1; num<=50; num++)
```

```
        {
                flag = CheckPrime(num);
                if(flag)
                {
                        printf("%d", num);
                        sum+ = num;
                        count++;
                }
        }
        printf("Sum of primes %d", count);
}
int CheckPrime(int n)
{
        int srt, d;
        srt = sqrt(n);
        d = 2;
        while(d <= srt)
        {
                If(n%d == 0)
                break;
                d++;
        }
        if(d > srt)
                return(1);
        else
                return(0);
}
```

(a) Draw the DD graph for the program.
(b) This program consists of main() and one module. Calculate the individual cyclomatic complexity number for both and collectively for the whole program. Show that individual cyclomatic complexity of main() and CheckPrime() and cyclomatic complexity of whole program is equal.
(c) List all independent paths.
(d) Design all test cases from independent paths.
(e) Derive all du-paths and dc-paths using data flow testing.

13. Consider the following program for calculating the grade of a student:

```
main()
{
     char grade;
     int s1, s2, s3, s4, total;
     float average;
     printf("Enter the marks of 4 subjects");
     scanf("%d  %d  %d  %d", &s1,&s2,&s3,&s4);
```

```
        if(s1 < 40 || s2 < 40 || s3 < 40 || s4 < 40)
              printf("Student has failed");
        else
        {
              total = s1 + s2 + s3 + s4;
              average = total/4.0;
              if(average > 80.0)
                    grade = 'A';
              else if(average >= 70.0)
                    grade = 'B';
              else if(average >= 60.0)
                    grade = 'C';
              else if(average >= 50.0)
                    grade = 'D';
              else if(average >= 45.0)
                    grade = 'E';
              else
                    grade = 'F';
              switch(grade)
              {
                    case 'A':  printf("Student has got A grade");
                    case 'B':  printf("Student has got B grade");
                    case 'C':  printf("Student has got C grade");
                    case 'D':  printf("Student has got D grade");
                    case 'E':  printf("Student has got E grade");
                    case 'F':  printf("Student has got F grade");
              }
        }
}
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all four methods.
(c) List all independent paths.
(d) Design all test cases from independent paths.

14. Consider the following graph:

(a) Represent this graph in the form of a graph matrix.
(b) Represent this graph in the form of a connection matrix.
(c) Find 2-link paths and 3-link paths for each node.

15. 'Nested loops are problematic areas for testers.' Comment on this.
16. Give examples of each type of data anomaly.
17. Consider Question 9 and perform static data flow analysis by finding out the define-use-kill patterns for all the variables in the source code.
18. Consider Question 9 and draw data flow graph for each variable used in the program and derive data flow testing paths with all the strategies.
19. What is the difference between primary and secondary mutants?
20. Consider Example 5.10. Find out the possible mutants and check how many of them are killed by a set of test data. Add new test data if required.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter

6

Static Testing

OBJECTIVES

After reading this chapter, you should be able to understand:

- Static testing is complementary to dynamic testing
- Static testing also improves the software quality
- Some bugs can be detected only through static testing
- Three types of static testing: Inspection, Walkthroughs, and Reviews
- Inspections are the most widely used technique for static testing which is a formal process to detect the bugs early
- Benefits and effectiveness of inspection process
- Variants of inspection process
- Walkthrough is a less formal and less rigorous method as compared to inspection process
- Review is a higher level technique as compared to inspection or walkthrough, as it also includes management representatives

We have discussed dynamic testing techniques that execute the software being built with a number of test cases. However, it suffers from the following drawbacks:

- Dynamic testing uncovers the bugs at a later stage of SDLC and hence is costly to debug.
- Dynamic testing is expensive and time-consuming, as it needs to create, run, validate, and maintain test cases.
- The efficiency of code coverage decreases with the increase in size of the system.
- Dynamic testing provides information about bugs. However, debugging is not always easy. It is difficult and time-consuming to trace a failure from a test case back to its root cause.
- Dynamic testing cannot detect all the potential bugs.

While dynamic testing is an important aspect of any quality assurance program, it is not a universal remedy. Thus, it alone cannot guarantee defect-free product, nor can it ensure a sufficiently high level of software quality.

In response to the above discussion, static testing is a complimentary technique to dynamic testing technique to acquire higher quality software. Static testing techniques do not execute the software and they do not require the bulk of test cases. This type of testing is also known as non-computer based testing or human testing. All the bugs cannot be caught alone by the dynamic testing technique; static testing reveals the errors which are not shown by dynamic testing. Static testing can be applied for most of the verification activities dis-

cussed earlier. Since verification activities are required at every stage of SDLC till coding, static testing also can be applied at all these phases.

Static testing techniques do not demonstrate that the software is operational or that one function of software is working; rather they check the software product at each SDLC stage for conformance with the required specifications or standards. Requirements, design specifications, test plans, source code, user's manuals, maintenance procedures are some of the items that can be statically tested.

Static testing has proved to be a cost-effective technique of error detection. An empirical comparison between static and dynamic testing [26, 27], proves the effectiveness of static testing. Further, Fagan [28] reported that more than 60% of the errors in a program can be detected using static testing. Another advantage of static testing is that it provides the exact location of a bug, whereas dynamic testing provides no indication of the exact source code location of the bug. In other words, we can say that static testing finds the in-process errors before they become bugs.

Static testing techniques help to produce a better product. Given below are some of the benefits of adopting a static testing approach:

- As defects are found and fixed, the quality of the product increases.
- A more technically correct base is available for each new phase of development.
- The overall software life cycle cost is lower, since defects are found early and are easier and less expensive to fix.
- The effectiveness of the dynamic test activity is increased and less time needs to be devoted for testing the product.
- Immediate evaluation and feedback to the author from his/her peers which will bring about improvements in the quality of future products.

The objectives of static testing can be summarized as follows:

- To identify errors in any phase of SDLC as early as possible
- To verify that the components of software are in conformance with its requirements
- To provide information for project monitoring
- To improve the software quality and increase productivity

Types of Static Testing

Static testing can be categorized into the following types:

- Software inspections
- Walkthroughs
- Technical reviews

6.1 Inspections

Software inspections were first introduced at IBM by Fagan in the early 1970s [43]. These can be used to tackle software quality problems because they allow the detection and removal of defects after each phase of the software development process. Inspection process is an in-process manual examination of an item to detect bugs. It may be applied to any product or partial product of the software development process, including requirements, design and code, project management plan, SQA plan, software configuration plan (SCM plan), risk management plan, test cases, user manual, etc. Inspections are embedded in the process of developing products and are done in the early stages of each product's development.

This process does not require executable code or test cases. With inspection, bugs can be found on infrequently executed paths that are not likely to be included in test cases. Software inspection does not execute the code, so it is machine-independent, requires no target system resources or changes to the program's operational behaviour, and can be used much before the target hardware is available for dynamic testing purposes.

The inspection process is carried out by a group of peers. The group of peers first inspect the product at the individual level. After this, they discuss the potential defects of the product observed in a formal meeting. The second important thing about the inspection process is that it is a formal process of verifying a software product. The documents which can be inspected are SRS, SDD, code, and test plan.

An inspection process involves the interaction of the following elements:

- Inspection steps
- Role for participants
- Item being inspected

The entry and exit criteria are used to determine whether an item is ready to be inspected. Entry criteria mean that the item to be inspected is mature enough to be used. For example, for code inspection, the entry criterion is that the code has been compiled successfully. Exit criterion is that once the item has been given for inspection, it should not be updated, otherwise it will not know how many bugs have been reported and corrected through the inspection process and the whole purpose of the inspection is lost.

6.1.1 Inspection Team

For the inspection process, a minimum of the following four team members are required.

Author/Owner/Producer A programmer or designer responsible for producing the program or document. He is also responsible for fixing defects discovered during the inspection process.

Inspector A peer member of the team, i.e. he is not a manager or supervisor. He is not directly related to the product under inspection and may be concerned with some other product. He finds errors, omissions, and inconsistencies in programs and documents.

Moderator A team member who manages the whole inspection process. He schedules, leads, and controls the inspection session. He is the key person with the responsibility of planning and successful execution of the inspection.

Recorder One who records all the results of the inspection meeting.

6.1.2 Inspection Process

A general inspection process (see Fig. 6.1) has the following stages [29,14]:

Figure 6.1 Inspection process

Planning During this phase, the following is executed:

- The product to be inspected is identified.
- A moderator is assigned.
- The objective of the inspection is stated, i.e. whether the inspection is to be conducted for defect detection or something else. If the objective is defect detection, then the type of defect detection like design error, interface error, code error must be specified. The aim is to define an objective for the meeting so that the effort spent in inspections is properly utilized.

During planning, the moderator performs the following activities:

- Assures that the product is ready for inspection
- Selects the inspection team and assigns their roles
- Schedules the meeting venue and time
- Distributes the inspection material like the item to be inspected, checklists, etc.

Overview In this stage, the inspection team is provided with the background information for inspection. The author presents the rationale for the product, its relationship to the rest of the products being developed, its function and intended use, and the approach used to develop it. This information is necessary for the inspection team to perform a successful inspection.

The opening meeting may also be called by the moderator. In this meeting, the objective of inspection is explained to the team members. The idea is that every member should be familiar with the overall purpose of the inspection.

Individual preparation After the overview, the reviewers individually prepare themselves for the inspection process by studying the documents provided to them in the *overview* session. They point out potential errors or problems found and record them in a log. This log is then submitted to the moderator. The moderator compiles the logs of different members and gives a copy of this compiled list to the author of the inspected item.

The inspector reviews the product for general problems as well as those related to their specific area of expertise. Checklists are used during this stage for guidance on typical types of defects that are found in the type of product being inspected. The product being inspected is also checked against standard documents to assure compliance and correctness. After reviewing, the inspectors record the defects found on a log and the time spent during preparation. Completed preparation logs are submitted to the moderator prior to the inspection meeting.

The moderator reviews the logs submitted by each inspector to determine whether the team is adequately prepared. The moderator also checks for trouble spots that may need extra attention during inspection, common defects that can be categorized quickly, and the areas of major concern. If the logs indicate that the team is not adequately prepared, the moderator should reschedule the inspection meeting. After this, the compiled log file is submitted to the author.

Inspection meeting Once all the initial preparation is complete, the actual inspection meeting can start. The inspection meeting starts with the author of the inspected item who has created it. The author first discusses every issue raised by different members in the compiled log file. After the discussion, all the members arrive at a consensus whether the issues pointed out are in fact errors and if they are errors, should they be admitted by the author. It may be possible that during the discussion on any issue, another error is found. Then, this new error is also discussed and recorded as an error by the author.

The basic goal of the inspection meeting is to uncover any bug in the item. However, no effort is made in the meeting to fix the bug. It means that bugs are only being notified to the author, which he will fix later. If there is any clarification regarding these bugs, then it should be asked or discussed with other members during the meeting.

Another fact regarding the inspection is that the members of the meeting should be sensitive to the feelings of the author. The author should not be attacked by other members such that the author feels guilty about the product. Every activity in the meeting should be a constructive engagement so that more and more bugs can be discovered. It is the duty of the moderator that the meeting remains focused towards its objective and the author is not discouraged in any way.

At the end, the moderator concludes the meeting and produces a summary of the inspection meeting. This summary is basically a list of errors found in the item that need to be resolved by the author.

Rework The summary list of the bugs that arise during the inspection meeting needs to be reworked by the author. The author fixes all these bugs and reports back to the moderator.

Follow-up It is the responsibility of the moderator to check that all the bugs found in the last meeting have been addressed and fixed. He prepares a report and ascertains that all issues have been resolved. The document is then approved for release. If this is not the case, then the unresolved issues are mentioned in a report and another inspection meeting is called by the moderator.

6.1.3 Benefits of Inspection Process

Bug reduction The number of bugs is reduced through the inspection process. L.H. Fenton [86] reported that through the inspection process in IBM, the number of bugs per thousand lines of code has been reduced by two-thirds. Thus, inspection helps reduce bug injection and detection rates. According to Capers Jones, 'Inspection is by far the most effective way to remove bugs.'

Bug prevention Inspections can also be used for bug prevention. Based on the experiences of previous inspections, analysis can be made for future inspections or projects, thereby preventing the bugs which have appeared earlier. Programmers must understand why bugs appear and what can be done to avoid them in future. At the same time, they should provide inspection results to the quality assurance team.

Productivity Since all phases of SDLC may be inspected without waiting for code development and its execution, the cost of finding bugs decreases, resulting in an increase in productivity. Moreover, the errors are found at their exact places, therefore reducing the need of dynamic testing and debugging. In the article by Fagan [43], an increase of 23% in coding productivity and a 25% reduction in schedules were reported.

Real-time feedback to software engineers The inspections also benefit software engineers/developers because they get feedback on their products on a relatively real-time basis. Developers find out the type of mistakes they make and what is the error density. Since they get this feedback in the early stages of development, they may improve their capability. Thus, inspections benefit software engineers/developers in the sense that they can recognize their weakness and improve accordingly, which in turn benefits the cost of the project.

Reduction in development resource The cost of rework is surprisingly high if inspections are not used and errors are found during development or testing. Therefore, techniques should be adopted such that errors are found and fixed as close to their place of origin as possible. Inspections reduce the effort required for dynamic testing and any rework during design and code, thereby causing an overall net reduction in the development resource. Without inspections, more resources may be required during design and dynamic testing. But, with inspection, the resource requirement is greatly reduced.

Quality improvement We know that the direct consequence of testing is improvement in the quality of software. The direct consequence of static testing also results in the improvement of quality of the final product. Inspections help to improve the quality by checking the standard compliance, modularity, clarity, and simplicity.

Project management A project needs monitoring and control. It depends on some data obtained from the development team. However, this data cannot be relied on forever. Inspection is another effective tool for monitoring the progress of the project.

Checking coupling and cohesion The modules' coupling and cohesion can be checked easily through inspection as compared to dynamic testing. This also reduces the maintenance work.

Learning through inspection Inspection also improves the capability of different team members, as they learn from the discussions on various types of bugs and the reasons why they occur. It may be more beneficial for new members. They can learn about the project in a very short time. This helps them in the later stages of development and testing.

Process improvement There is always scope of learning from the results of one process. An analysis of why the errors occurred or the frequent places where the errors occurred can be done by the inspection team members. The analysis results can then be used to improve the inspection process so that the current as well as future projects can benefit. Discussed below are the issues of process improvement.

Finding most error-prone modules Through the inspection process, the modules can be analysed based on the error-density of the individual module, as shown in Table 6.1.

Table 6.1 Error-prone modules

Module Name	Error density (Error/ KLoC)
A	23
B	13
C	45

From the example modules given in Table 6.1, Module C is more error-prone. This information can be used and some decision should be taken as to:

(i) Redesign the module

(ii) Check and rework on the code of Module C

(iii) Take extra precautions and efficient test cases to test the module

Distribution of error-types Inspections can also be used to provide data according to the error-types. It means that we can analyse the data of the percentage of bugs in a particular type of bug category, as shown in Table 6.2.

Table 6.2 Bug type distribution

Bug Type	No. of errors	%
Design Bugs	78	57.8
Interface Bugs	45	33.4
Code Bugs	12	8.8

If we get this data very early in the development process, then we can analyse which particular type of bugs are repeating. We can modify the process well in time and the results can also be used in future.

In this way, as the organization gains experience, the process can be improved.

6.1.4 Effectiveness of Inspection Process

In an analysis done by Oliver Laitenberger [107], the inspection process was found to be effective as compared to structural testing because the inspection process alone found 52% errors. Of the remaining 48% of defects, on an average, only 17% are detected by structural testing.

The effectiveness of the inspection process lies in its rate of inspection. The rate of inspection refers to how much evaluation of an item has been done by the team. If the rate is very fast, it means the coverage of item to be evaluated is high. Similarly if the rate is too slow, it means that the coverage is not much. However, the rate should also be considered in the perspective of detection of errors. If the rate is too fast, then it may be possible that it reveals only few errors. If the rate is too slow, then the cost of project increases. Thus, a balanced rate of inspection should be followed so that it concentrates on a reasonable number of defects. It may be calculated as the error detection efficiency of the inspection process, as given below:

$$\text{Error detection efficiency} = \frac{\text{Error found by an inspection}}{\text{Total errors in the item before inspection}} \times 100$$

The rate of inspection also depends on some factors like the experience of the inspection team, the programming language, and the application domain.

6.1.5 Cost of Inspection Process

With at least four members involved in an inspection team, the cost of inspecting 100 lines of code is roughly equivalent to one person-day of effort. This

assumes that the inspection itself takes about an hour and that each member spends 1-2 hours preparing for the inspection. Testing costs are variable and depend on the number of faults in the software. However, the effort required for the program inspection is less than half the effort that would be required for equivalent dynamic testing. Further, it has been estimated that the cost of inspection can be 5–10% of the total cost of the project.

6.1.6 Variants of Inspection process

After Fagan's original formal inspection concept, many researchers proposed modifications in it. Table 6.3 lists some of the variants of the formal inspection.

Table 6.3 Formal inspection variants

Active Design Reviews (ADRs)	Several reviews are conducted targeting a particular type of bugs and conducted by the reviewers who are experts in that area.
Formal Technical Asynchronous review method (FTArm)	Inspection process is carried out without really having a meeting of the members. This is a type of asynchronous inspection in which the inspectors never have to simultaneously meet.
Gilb Inspection	Defect detection is carried out by individual inspector at his level rather than in a group.
Humphrey's Inspection Process	Preparation phase emphasizes the finding and logging of bugs, unlike Fagan inspections. It also includes an analysis phase wherein individual logs are analysed and combined into a single list.
N-Fold inspections	Inspection process's effectiveness can be increased by replicating it by having multiple inspection teams.
Phased Inspection	Phased inspections are designed to verify the product in a particular domain by experts in that domain only.
Structured Walkthrough	Described by Yourdon. Less formal and rigorous than formal inspections. Roles are coordinator, scribe, presenter, reviewers, maintenance oracle, standards bearer, user representative. Process steps are Organization, Preparation, Walkthrough, and Rework. Lacks data collection requirements of formal inspections.

Active Design Reviews

Active Design Reviews [87] are for inspecting the design stage of SDLC. In this process, several reviews are conducted targeting a particular type of bugs and conducted by the reviewers who are experts in that area. Thus, it covers all the sections of the design document based on several small reviews instead of only one inspection. It is also based on the use of questionnaires to give the reviewers better-defined responsibilities and to make them play a more active role. The questionnaires are designed to make the reviewers take an active

stand on issues and to use the documentation. The technique has been used in the Naval Research Laboratory's (Washington) software cost reduction (SCR) project for several years with good results. The SCR project involves the experimental redevelopment of the operational flight program (OFP) for the Navy's A-7E aircraft.

These are conducted in the following stages (see Fig. 6.2):

Figure 6.2 Active design reviews process

Overview A brief overview of the module being reviewed is presented. The overview explains the modular structure in case it is unfamiliar to reviewers, and shows them where the module belongs in the structure.

Review Reviewers are assigned sections of the document to be reviewed and questionnaires based on the bug type. Reviewers answer the questions in the questionnaires. They are also assigned a timeframe during which they may raise any questions they have, and a time to meet with designers after the designers have read the completed questionnaires.

Meeting The designers read the completed questionnaires and meet the reviewers to resolve any queries that the designers may have about the reviewer's answers to the questionnaires. The reviewers may be asked to defend their answers. This interaction continues until both designers and reviewers understand the issues, although an agreement on these issues may not be reached. After the review, the designers produce a new version of the documentation.

Formal Technical Asynchronous Review Method (FTArm)

In this process, the meeting phase of inspection is considered expensive and therefore, the idea is to eliminate this phase. The inspection process is carried out without having a meeting of the members. This is a type of asynchronous inspection [88] in which the inspectors never have to simultaneously meet. For this process, an online version of the document is made available to every member where they can add their comments and point out the bugs. This process consists of the following steps, as shown in Fig. 6.3.

Setup It involves choosing the members and preparing the document for an asynchronous inspection process. The document is prepared as a hypertext document.

Figure 6.3 Asynchronous method

Orientation It is same as the overview step of the inspection process discussed earlier.

Private review It is same as the preparation phase discussed in the inspection process. Here, each reviewer or inspector individually gives his comments on the document being inspected. However, each inspector provides comments individually and is unable to see the other inspector's comments.

Public review In this step, all comments provided privately are made public and all inspectors are able to see each other's comments and put forward their suggestions. Based on this feedback and with consultation of the author, it is decided that there is a bug.

Consolidation In this step, the moderator analyses the result of private and public reviews and lists the findings and unresolved issues, if any.

Group meeting If required, any unresolved issues are discussed in this step. But the decision to conduct a group meeting is taken in the previous step only by the moderator.

Conclusion The final report of the inspection process along with the analysis is produced by the moderator.

Gilb Inspection

Gilb and Graham [89] defined this process. It differs from Fagan inspection in that the defect detection is carried out by individual inspectors at their own level rather than in a group. Therefore, a checking phase has been introduced. Three different roles are defined in this type of inspection:

- **Leader** is responsible for planning and running the inspection.
- **Author of the document**
- **Checker** is responsible for finding and reporting the defects in the document.

The inspection process consists of the following steps, as shown in Fig. 6.4.

Figure 6.4 Gilb inspection process

Entry The document must pass through an entry criteria so that the inspection time is not wasted on a document which is fundamentally flawed.

Planning The leader determines the inspection participants and schedules the meeting.

Kick-off The relevant documents are distributed, participants are assigned roles and briefed about the agenda of the meeting.

Checking Each checker works individually and finds defects.

Logging Potential defects are collected and logged.

Brainstorm In this stage, process improvement suggestions are recorded based on the reported bugs.

Edit After all the defects have been reported, the author takes the list and works accordingly.

Follow-up The leader ensures that the edit phase has been executed properly.

Exit The inspection must pass the exit criteria as fixed for the completion of the inspection process.

Humphrey's Inspection Process

It was described by Watts Humphrey [90]. In this process, the preparation phase emphasizes on finding and logging of bugs, unlike Fagan inspections. It also includes an analysis phase between preparation and meeting. In the analysis phase, individual logs are analysed and combined into a single list. The steps of this process are shown in Fig. 6.5.

Figure 6.5 Humphrey's process

N-Fold Inspection

It is based on the idea that the effectiveness of the inspection process can be increased by replicating it [91]. If we increase the number of teams inspecting the item, the percentage of defects found may increase. But sometimes the cost of organizing multiple teams is higher as compared to the number of defects found. A proper evaluation of the situation is required. Originally, this process was used for inspecting requirement specifications, but it can also be used for any phase.

As discussed, this process consists of many independent inspection teams. This process needs a coordinator who coordinates various teams, collects and collates the inspection data received by them. For this purpose, he also meets the moderator of every inspection team. This process consists of the following stages, as shown in Fig. 6.6.

Planning and overview This is the formal planning and overview stage. In addition, it also includes the planning of how many teams will participate in the inspection process.

Inspection stages There are many inspection processes adopted by many teams. It is not necessary that every team will choose the same inspection process. The team is free to adopt any process.

Collation phase The results from each inspection process are gathered, collated, and a master list of all detected defects is prepared.

Rework and follow-up This step is same as the tradition Fagan inspection process.

Figure 6.6 N-fold inspection

Phased Inspection

Phased inspections [92] are designed to verify the product in a particular domain. If we feel that in a particular area, we may encounter errors, then phased inspections are carried out. But for this purpose, only experts who have experience in that particular domain are called. Thus, this inspection process provides a chance to utilize human resources. In this process, inspection is divided into more than one phase. There is an ordered set of phases

and each phase is designed such that it will check a particular feature in the product. The next phase in the ordered set of phases assumes that the particular feature has been verified in the previous phase. There are two types of phases, as discussed below.

Single inspector In this phase, a rigorous checklist is used by a single inspector to verify whether the features specified are there in the item to be inspected.

Multiple inspector Here, a checklist cannot be used. It means that the item cannot be verified just with the questions mentioned in the checklist. There are many inspectors who are distributed the required documents for verification of an item. Each inspector examines this information and develops a list of questions of their own based on a particular feature. These questions are not in binary form as in single inspection. The item is then inspected individually by all the inspectors based on a self-developed checklist which is either application or domain specific. After individual checking by the inspectors, a reconciliation meeting is organized where inspectors compare their findings about the item.

Structured Walkthrough

Though structured walkthrough is a variant of the inspection process, we have now categorized it as a separate category for static testing. We will discuss structured walkthrough in Section 6.2.

6.1.7 Reading Techniques

A reading technique can be defined as a series of steps or procedures whose purpose is to guide an inspector to acquire a deep understanding of the inspected software product. Thus, a reading technique can be regarded as a mechanism or strategy for the individual inspector to detect defects in the inspected product. Most of the techniques found in the literature support individual inspection work. The various reading techniques are discussed below.

Ad hoc method In this method, there is no direction or guidelines provided for inspection. However, ad hoc does not mean that inspection participants do not scrutinize the inspected product systematically. The word *ad hoc* only refers to the fact that no technical support on how to detect defects in a software artifact is given to them. In this case, defect detection fully depends on the skills, knowledge, and experience of an inspector.

Checklists A checklist is a list of items that focus the inspector's attention on specific topics, such as common defects or organizational rules, while reviewing a software document [43]. The purpose of checklists is to gather expertise

concerning the most common defects and thereby supporting inspections. The checklists are in the form of questions which are very general and are used by the inspection team members to verify the item.

However, checklists have some drawbacks too:

- The questions, being general in nature, are not sufficiently tailored to take into account a particular development environment.
- Instruction about using a checklist is often missing.
- There is a probability that some defects are not taken care of. It happens in the case of those type of defects which have not been previously detected.

Scenario-based reading Checklists are general in nature and do not cover different types of bugs. Scenario-based reading [93] is another reading technique which stresses on finding different kind of defects. It is based on scenarios, wherein inspectors are provided with more specific instructions than typical checklists. Additionally, they are provided with different scenarios, focusing on different kind of defects.

Basili *et al.* [94] define scenario-based reading as a high-level term, which they break down to more specific techniques. The original method of Porter & Votta [93] is also known as *defect-based reading*. By using the definition by Basili *et al.,* most of the reading techniques are based on scenario-based techniques wherein the inspector has to actively work with the inspected documents instead of mere straightforward reading. The following methods have been developed based on the criteria given by scenario-based reading.

Perspective-based reading The idea behind this method is that a software item should be inspected from the perspective of different stakeholders [95,104,105]. Inspectors of an inspection team have to check the software quality as well as the software quality factors of a software artifact from different perspectives. The perspectives mainly depend upon the roles people have within the software development or maintenance process. For each perspective, either one or multiple scenarios are defined, consisting of repeatable activities an inspector has to perform, and the questions an inspector has to answer. For example, a testing expert first creates a test plan based on the requirements specification and then attempts to find defects from it.

This is designed specifically for requirements inspection, and inspectors who participate in inspection should have expertise in inspecting requirements of their own area of expertise. Later, perspective-based reading has been applied to code inspections and design inspections.

Usage-based reading This method proposed by Thelin *et al.* [96,97,98] is applied in design inspections. Design documentation is inspected based on use

cases, which are documented in requirements specification. Since use-cases are the basis of inspection, the focus is on finding functional defects, which are relevant from the users' point of view.

Abstraction driven reading This method given by Dunsmore *et al.* [99,100,101] is designed for code inspections. In this method, an inspector reads a sequence of statements in the code and abstracts the functions these statements compute. An inspector repeats this procedure until the final function of the inspected code artifact has been abstracted and can be compared to the specification. Thus, the inspector creates an abstraction level specification based on the code under inspection to ensure that the inspector has really understood the code.

Task-driven reading This method is also valid for code inspections proposed by Kelly & Shepard [102]. In this method, the inspector has to create a data dictionary, a complete description of the logic and a cross-reference between the code and the specifications.

Function-point based scenarios This is based on scenarios for defect detection in requirements documents [103]. This approach is based on function-point analysis (FPA). FPA defines a software system in terms of its inputs, files, inquiries, and outputs. The scenarios designed around function-points are known as *function-point scenarios.* It consists of questions and directs the focus of an inspector to a specific function-point item within the inspected requirements document.

6.1.8 Checklists for Inspection Process

The inspection team must have a checklist against which they detect errors. The checklist is according to the item to be inspected. For example, the design document and the code of the module should have different checklists. Checklists can be prepared with the points mentioned in the verification of each phase. Checklists should be prepared in consultation with experienced staff and regularly updated as more experience is gained by the inspection process.

The checklist may vary according to the environment and needs of the organization. Each organization should prepare its own checklists for every item.

6.2 Structured Walkthroughs

The idea of *structured walkthroughs* was proposed by Yourdon [106]. It is a less formal and less rigorous technique as compared to inspection. The common term used for static testing is *inspection* but it is a very formal process. If you

want to go for a less formal process having no bars of organized meeting, then walkthroughs are a good option.

A typical structured walkthrough team consists of the following members:

- **Coordinator** Organizes, moderates, and follows up the walkthrough activities.
- **Presenter/Developer** Introduces the item to be inspected. This member is optional.
- **Scribe/Recorder** Notes down the defects found and suggestion proposed by the members.
- **Reviewer/Tester** Finds the defects in the item.
- **Maintenance Oracle** Focuses on long-term implications and future maintenance of the project.
- **Standards Bearer** Assesses adherence to standards.
- **User Representative/Accreditation Agent** Reflects the needs and concerns of the user.

Walkthroughs differ significantly from inspections. An inspection is a six-step, rigorous, formalized process. The inspection team uses the checklist approach for uncovering errors. A walkthrough is less formal, has fewer steps and does not use a checklist to guide or a written report to document the team's work. Rather than simply reading the program or using error checklists, the participants 'play computer'. The person designated as a tester comes to the meeting armed with a small set of paper test cases—representative sets of inputs and expected outputs for the program or module. During the meeting, each test case is mentally executed. That is, the test data are walked through the logic of the program. The state of the program is monitored on a paper or any other presentation media. The walkthrough should have a follow-up process similar to that described in the inspection process. The steps of a walkthrough process are shown in Fig. 6.7.

Figure 6.7 Walkthrough process

6.3 Technical Reviews

A technical review is intended to evaluate the software in the light of development standards, guidelines, and specifications and to provide the management with evidence that the development process is being carried out according to

the stated objectives. A review is similar to an inspection or walkthrough, except that the review team also includes management. Therefore, it is considered a higher-level technique as compared to inspection or walkthrough.

A technical review team is generally comprised of management-level representatives and project management. Review agendas should focus less on technical issues and more on oversight than an inspection. The purpose is to evaluate the system relative to specifications and standards, recording defects and deficiencies. The moderator should gather and distribute the documentation to all team members for examination before the review. He should also prepare a set of indicators to measure the following points:

- Appropriateness of the problem definition and requirements
- Adequacy of all underlying assumptions
- Adherence to standards
- Consistency
- Completeness
- Documentation

The moderator may also prepare a checklist to help the team focus on the key points. The result of the review should be a document recording the events of the meeting, deficiencies identified, and review team recommendations. Appropriate actions should then be taken to correct any deficiencies and address all recommendations.

SUMMARY

Static testing techniques are not as popular as dynamic testing techniques. However, researchers have found static testing to be an effective aid in finding and preventing the defects at an early stage. In recent years, due to the importance of early testing, these techniques are finding their place in industries.

This chapter introduces static testing as a complementary technique to dynamic testing. Static testing reveals the bugs which cannot be detected by dynamic testing. We have taken three categories of static testing, namely inspection, walkthroughs, and reviews, and distinguished them. Inspection process is a widely used term for static testing. A lot of terminologies are used interchangeably for static testing.

We have discussed every aspect of inspection process including the team members and the inspection process. Moreover, in the literature, many variants of this process are also available. These variants have also been discussed.

Let us review the important concepts discussed in this chapter:

- The inspection process is a formal in-process manual examination of an item to detect bugs. It may be applied to any product or partial product of the software development process. This process is carried out by a group of peers with the help of checklists.

- The rate of inspection is calculated as the error detection efficiency of the inspection process as given below:

$$\text{Error detection efficiency} = \frac{\text{Error found by an inspection}}{\text{Total errors in the item before inspection}} \times 100$$

- Active Design Reviews inspects the design stage of SDLC. In this process, several reviews are conducted targeting a particular type of bugs.
- Formal Technical Asynchronous review method (FTArm) is a type of asynchronous inspection in which the inspectors never have to simultaneously meet. For this process, an online version of the document is made available to every member where they can add their comments and point out the bugs.
- Gilb Inspection is a type of inspection process wherein the defect detection is carried out by individual inspectors at their own level rather than in a group.
- Humphrey's Inspection emphasizes finding and logging the bugs in the preparation phase. It also includes an analysis phase between preparation and meeting. In the analysis phase, individual logs are analysed and combined into a single list.
- N-Fold Inspection process consists of many independent inspection teams to increase the effectiveness of the inspection process and results in detecting more number of bugs quickly. This process needs a coordinator who coordinates various teams, collects and collates the inspection data received by them.
- Phased Inspections are designed to verify the product in a particular domain with the help of experts in that domain. In this process, the inspection is divided into more than one phase. There is an ordered set of phases and each phase is designed such that it will check a particular feature in the product.
- A Reading Technique can be regarded as a mechanism or strategy for the individual inspector to detect defects in the inspected product.
- A Walkthrough is less formal, has fewer steps, and does not use a checklist to guide or a written report to document the team's work.
- A Technical Review is intended to evaluate the software in light of development standards, guidelines, and specifications and to provide the management with evidence that the development process is being carried out according to the stated objectives. A review is similar to an inspection or walkthrough, except that the review team also includes management.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. In static testing, a bug is found at its _____ location.
   (a) Exact
   (b) Nearby
   (c) None of the above
2. Static testing can be applied for most of the _____ .
   (a) Validation activities

(b) Verification activities
(c) SDLC activities
(d) None of the above

3. Formal peer evaluation of a software element whose objective is to verify that the software element satisfies its specifications and conforms to standards, is called _____ .
   (a) Walkthrough
   (b) Inspections
   (c) Reviews
   (d) None of the above
4. The programmer or designer responsible for producing the program or document is known as _____ .
   (a) Author
   (b) Owner
   (c) Producer
   (d) All
5. The person who finds errors, omissions, and inconsistencies in programs and documents during an inspection is known as _____ .
   (a) Inspector
   (b) Moderator
   (c) Author
   (d) Producer
6. The key person with the responsibility of planning and successful execution of inspection is known as _____ .
   (a) Inspector
   (b) Moderator
   (c) Author
   (d) Producer
7. The inspection team points out any potential errors or problems found and records them in _____.
   (a) SDD
   (b) SRS
   (c) STD
   (d) Log Form
8. 'How much evaluation of an item has been done by the team' is called _____ .
   (a) Rate of errors
   (b) Rate of inspection
   (c) Rate of failures
   (d) None of the above
9. _____ is a more formal process.
   (a) Walkthroughs
   (b) Inspection

(c) Reviews
(d) None of the above

10. The efficiency of code coverage performed by dynamic testing _____ with the increase in size of the system.
    (a) Decreases
    (b) Increases
    (c) Remains same
    (d) None of the above
11. Through the inspection process, the modules can be analysed based on _____.
    (a) Error-types
    (b) Inspection reports
    (c) Error-density
    (d) None of the above
12. The inspection in which the inspectors never have to simultaneously meet is known as _____.
    (a) Phased Inspection
    (b) FTArm
    (c) Gilb Inspection
    (d) All
13. Checking phase has been introduced in _____.
    (a) Phased Inspection
    (b) FTArm
    (c) Gilb Inspection
    (d) None of the above
14. Analysis phase between preparation and meeting has been introduced in _____.
    (a) Phased Inspection
    (b) Humphrey's Inspection
    (c) FTArm
    (d) N-fold Inspection
15. Collation Phase has been introduced in _____.
    (a) Phased Inspection
    (b) Humphrey's Inspection
    (c) FTArm
    (d) N-fold Inspection
16. _____ process gives the chance to utilize human resources.
    (a) Phased Inspection
    (b) Humphrey's Inspection
    (c) FTArm
    (d) N-fold Inspection
17. A series of steps or procedures whose purpose is to guide an inspector in acquiring a deep understanding of the inspected software is known as _____.

(a) Checklists
(b) Inspection
(c) Reading Techniques
(d) N-fold Inspection

18. _____ is a reading technique.
    (a) Checklists
    (b) Inspection
    (c) Usage-based method
    (d) Task-based method
19. A review is similar to an inspection or walkthrough, except that the review team also includes _____.
    (a) Customer
    (b) Developer
    (c) Tester
    (d) Management
20. _____ is not an inspection variant.
    (a) Active design review
    (b) FTArm
    (c) Walkthrough
    (d) None of the above

Review Questions

1. What are the advantages of static testing as compared to dynamic testing?
2. What are the benefits of inspection process as compared to dynamic testing?
3. Who can be a member of the inspection team?
4. What are the stages of an inspection process?
5. How does the rate of inspection affect the effectiveness of the inspection process?
6. What is the difference between inspection, walkthrough, and reviews?
7. Make a table indicating the feature of every type of inspection variant.
8. What are the factors that increase the effectiveness of N-Fold inspection?
9. How is scenario-based reading different from checklists? Explain the types of scenario-based reading.
10. Take a small project and apply all the static testing techniques on its SRS, SDD, and code.
11. Develop a list based on usage-based reading method on the project taken in Problem 10.
12. What are the drawbacks of checklists?
13. Is ad-hoc method a random method of reading?

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter

7

Validation Activities

OBJECTIVES

After reading this chapter, you should be able to understand:

- Validation is the next step after verification
- Validation is performed largely by black-box testing techniques
- Unit validation testing and role of stubs and drivers in unit validation
- Integration testing and its types: Decomposition-based integration, call graph-based integration, path graph-based integration testing
- Types of Decomposition-based integration: Top-down integration, Bottom-up integration testing
- Function testing
- System testing and its types: Recovery testing, Security testing, Stress testing, Performance testing, Usability testing, Compatibility testing
- Acceptance testing and its types: Alpha and Beta testing

In this chapter, we will discuss the details of all validation concepts. Once we have verified every activity/stage in SDLC as shown in Fig. 7.1, the software or module built up till this stage should also be validated. By validation, we mean that the module that has been prepared till now is in conformance with the requirements which were set initially in the SRS or user manual.

As shown in Fig. 7.1, every validation testing focuses on a particular SDLC phase and thereby focuses on a particular class of errors. We can see a one-to-one correspondence between development and testing processes. For example, the purpose of unit validation testing is to find discrepancies between the developed module's functionality and its requirements and interfaces specified in the SRS. Similarly, the purpose of system validation testing is to explore whether the product is consistent with its original objectives. The advantage of this structure of validation testing is that it avoids redundant testing and prevents one from overlooking large classes of errors.

Software validation is achieved through a series of black-box tests that demonstrate conformity with requirements [7]. A test plan outlines the classes of tests to be conducted and a test procedure defines specific test cases that will be used to demonstrate conformity with requirements. Both the plan and procedure are designed to ensure that all functional requirements are satisfied, all behavioural character-

istics are achieved, all performance requirements are attained, documentation is correct and human-engineered, and other requirements are met.

Figure 7.1 V&V activities

Validation testing techniques are described in the following sections.

7.1 Unit Validation Testing

Since unit is the smallest building block of the software system, it is the first piece of system to be validated. Before we validate the entire software, units or modules must be validated. Unit testing is normally considered an adjunct to the coding step. However, it has been discussed that testing with the code of a unit is called the *verification step*. Units must also be validated to ensure that every unit of software has been built in the right manner in conformance with user requirements. Unit tests ensure that the software meets at least a baseline level of functionality prior to integration and system testing. While

developing the software, if the developer detects and removes the bug, it offers significant savings of time and costs. This type of testing is largely based on black-box techniques.

Though software is divided into modules but a module is not an isolated entity. The module under consideration might be getting some inputs from another module or the module is calling some other module. It means that a module is not independent and cannot be tested in isolation. While testing the module, all its interfaces must be simulated if the interfaced modules are not ready at the time of testing the module under consideration. Discussed below are the types of interface modules which must be simulated, if required, to test a module.

Drivers Suppose a module is to be tested, wherein some inputs are to be received from another module. However, this module which passes inputs to the module to be tested is not ready and under development. In such a situation, we need to simulate the inputs required in the module to be tested. For this purpose, a main program is prepared, wherein the required inputs are either hard-coded or entered by the user and passed on to the module under test. This module where the required inputs for the module under test are simulated for the purpose of module or unit testing is known as a *driver module*. The driver module may print or interpret the results produced by the module under testing.

For example, see the design hierarchy of the modules, as shown in Fig. 7.2. Suppose module B is under test. In the hierarchy, module A is a superordinate of module B. Suppose module A is not ready and B has to be unit tested. In this case, module B needs inputs from module A. Therefore, a driver module is needed which will simulate module A in the sense that it passes the required inputs to module B and acts as a main program for module B in which its being called, as shown in Fig. 7.3.

Figure 7.2 Design hierarchy of an example system

Figure 7.3 Driver Module for module A

Therefore, it can be said that a test driver is supporting the code and data used to provide an environment for testing a part of a system in isolation. In fact, it drives the unit being tested by creating necessary 'inputs' required for the unit and then invokes the unit. A test driver may take inputs in the following form and call the unit to be tested:

- It may hard-code the inputs as parameters of the calling unit.
- It may take the inputs from the user.
- It may read the inputs from a file.

Thus, a driver can be defined as a software module which is used to invoke a module under test and provide test inputs, control and monitor execution, and report test results or most simplistically a line of code that calls a method and passes a value to that method.

A test driver provides the following facilities to a unit to be tested:

- Initializes the environment desired for testing.
- Provides simulated inputs in the required format to the units to be tested.

Projects where commercial drivers are not available, specialized drivers need to be developed. This happens mainly in defence projects where projects are developed for a special application.

Stubs The module under testing may also call some other module which is not ready at the time of testing. Therefore, these modules need to be simulated for testing. In most cases, dummy modules instead of actual modules, which are not ready, are prepared for these subordinate modules. These dummy modules are called *stubs*.

Thus, a stub can be defined as a piece of software that works similar to a unit which is referenced by the unit being tested, but it is much simpler than the actual unit. A stub works as a 'stand-in' for the subordinate unit and provides the minimum required behaviour for that unit.

For example, consider Fig. 7.2 again. Module B under test needs to call module D and module E. But they are not ready. So there must be some skeletal structure in their place so that they act as dummy modules in place of the actual modules. Therefore, stubs are designed for module D and module E, as shown in Fig. 7.4.

Stubs have the following characteristics:

- Stub is a place holder for the actual module to be called. Therefore, it is not designed with the functionalities performed by the actual module. It is a reduced implementation of the actual module.
- It does not perform any action of its own and returns to the calling unit (which is being tested).
- We may include a display instruction as a trace message in the body of stub. The idea is that the module to be called is working fine by accepting the input parameters.
- A constant or null must be returned from the body of stub to the calling module.
- Stub may simulate exceptions or abnormal conditions, if required.

Figure 7.4 Stubs

Benefits of Designing Stubs and Drivers

The benefit of designing drivers and stubs (see Fig. 7.5) is that a unit will work/behave in the simulated environment as in the actual software environment. Now a unit test case can be written against the interface and the unit will still work properly once the drivers and stubs have been placed correctly.

The benefits of designing stubs and drivers are:

- Stubs allow the programmer to call a method in the code being developed, even if the method does not have the desired behaviour yet.
- By using stubs and drivers effectively, we can cut down our total debugging and testing time by testing small parts of a program individually, helping us to narrow down problems before they expand.

- Stubs and drivers can also be an effective tool for demonstrating progress in a business environment. For example, if you are to implement four specific methods in a class by the end of the week, you can insert stubs for any method and write a short driver program so that you can demonstrate to your manager or client that the requirement has been met. As you continue, you can replace the stubs and drivers with the real code to finish your program.

Figure 7.5 Drivers and Stubs

However, drivers and stubs represent overheads also. Overhead of designing them may increase the time and cost of the entire software system. Therefore, they must be designed simple to keep overheads low. Stubs and drivers are generally prepared by the developer of the module under testing. Developers use them at the time of unit verification. But they can also be used by any other person who is validating the unit.

Example 7.1

Consider the following program:

```
main()
{
        int a,b,c,sum,diff,mul;

        scanf("%d %d %d", &a, &b, &c);
        sum = calsum(a,b,c);
        diff = caldiff(a,b,c);
        mul = calmul(a,b,c);
        printf("%d %d %d", sum, diff, mul);
}

calsum(int x, int y, int z)
{
        int d;
```

```
        d = x + y + z;
        return(d);
}
```

(a) Suppose `main()` module is not ready for the testing of `calsum()` module. Design a driver module for `main()`.

(b) Modules `caldiff()` and `calmul()` are not ready when called in `main()`. Design stubs for these two modules.

Solution

(a) **Driver for `main()` module:**

```
main()
  {
        int a, b, c, sum;

        scanf("%d %d %d", &a, &b, &c);
        sum = calsum(a,b,c);
        printf("The output from calsum module is %d", sum);
  }
```

(b) **Stub for `caldiff()` Module**

```
caldiff(int x, int y, int z)
{
        printf("Difference calculating module");
        return 0;
}
```

Stub for `calmul()` Module

```
calmul(int x, int y, int z)
{
        printf("Multiplication calculation module");
        return 0;
}
```

7.2 Integration Testing

In the modular design of a software system where the system is composed of different modules, integration is the activity of combining the modules together when all the modules have been prepared. Integration of modules is according to the design of software specified earlier. Integration aims at constructing a working software system. But a working software demands full testing and thus, integration testing comes into the picture.

Why do we need integration testing? When all modules have been verified independently, then why is integration testing necessary? As discussed earlier, modules are not standalone entities. They are a part of a software system which comprises of many interfaces. Even if a single interface is mismatched, many modules may be affected. Thus, integration testing is necessary for the following reasons:

- Integration testing exposes inconsistency between the modules such as improper call or return sequences.
- Data can be lost across an interface.
- One module when combined with another module may not give the desired result.
- Data types and their valid ranges may mismatch between the modules.

Thus, integration testing focuses on bugs caused by interfacing between the modules while integrating them.

There are three approaches for integration testing, as shown in Fig. 7.6.

Figure 7.6 Integration Methods

7.2.1 Decomposition-Based Integration

The idea for this type of integration is based on the decomposition of design into functional components or modules. The functional decomposition is shown as a tree in Fig. 7.7. In the tree designed for decomposition-based integration, the nodes represent the modules present in the system and the links/edges between the two modules represent the calling sequence. The nodes on the last level in the tree are *leaf nodes.*

In the tree structure shown in Fig. 7.7, module A is linked to three subordinate modules, B, C, and D. It means that module A calls modules, B, C, and D. All integration testing methods in the decomposition-based integration assume that all the modules have been unit tested in isolation. Thus, with the decomposition-based integration, we want to test the interfaces among separately tested modules.

Figure 7.7 Decomposition Tree

Integration methods in decomposition-based integration depend on the methods on which the activity of integration is based. One method of integrating is to integrate all the modules together and then test it. Another method is to integrate the modules one by one and test them incrementally. Based on these methods, integration testing methods are classified into two categories: (a) non-incremental and (b) incremental.

Non-Incremental Integration Testing

In this type of testing, either all untested modules are combined together and then tested or unit tested modules are combined together. It is also known as Big-Bang integration testing.

Big-Bang method cannot be adopted practically. This theory has been discarded due to the following reasons:

1. Big-Bang requires more work. For example, consider the following hierarchy of a software system.

Figure 7.8 Integration Testing

According to the Big-Bang theory, if all unit tested modules are integrated in this example, then for unit testing of all the modules independently, we require four drivers and seven stubs. This count will grow according to the size of the system.

2. Actual modules are not interfaced directly until the end of the software system.
3. It will be difficult to localize the errors since the exact location of bugs cannot be found easily.

Incremental Integration Testing

In this type, you start with one module and unit test it. Then combine the module which has to be merged with it and perform test on both the modules. In this way, incrementally keep on adding the modules and test the recent environment. Thus, an integrated tested software system is achieved.

Incremental integration testing is beneficial for the following reasons:

1. Incremental approach does not require many drivers and stubs.
2. Interfacing errors are uncovered earlier.
3. It is easy to localize the errors since modules are combined one by one. The first suspect is the recently added module. Thus, debugging becomes easy.
4. Incremental testing is a more thorough testing. Consider the example in Fig. 7.8. If we are testing module 6, then although module 1 and 2 have been tested previously, they will get the chance to be tested again and again. This gives the modules more exposure to the probable bugs by the time the last module in the system is tested.

However, the two integration methods discussed above are sometimes not acceptable. Incremental testing suffers from the problem of serially combining the methods according to the design. But practically, sometimes it is not feasible in the sense that all modules are not ready at the same time. Therefore, incremental testing cannot be adopted in its pure form.

One method that may work is to borrow the good features of both the approaches. According to the big-bang method, all modules should be unit-tested independently as they are developed. In this way, there is parallelism. As soon as one module is ready, it can be combined and tested again in the integrated environment according to the incremental integration testing.

Types of Incremental Integration Testing

Design hierarchy of a software can be seen in a tree-like structure, as shown in Fig. 7.8. In this tree-like structure, incremental integration can be done either from top to bottom or bottom to top. Based on this strategy, incremental integration testing is divided into two categories.

Top-down Integration Testing The strategy in top-down integration is to look at the design hierarchy from top to bottom. Start with the high-level modules and move downward through the design hierarchy.

Modules subordinate to the top module are integrated in the following two ways:

Depth first integration In this type, all modules on a major control path of the design hierarchy are integrated first. In the example shown in Fig. 7.8, modules 1, 2, 6, 7/8 will be integrated first. Next, modules 1, 3, 4/5 will be integrated.

Breadth first integration In this type, all modules directly subordinate at each level, moving across the design hierarchy horizontally, are integrated first. In the example shown in Fig. 7.8, modules 2 and 3 will be integrated first. Next, modules 6, 4, and 5 will be integrated. Modules 7 and 8 will be integrated last.

However, in practice, these two sequences of top-down integration cannot be used every time. In general, there is no best sequence, but the following guidelines can be considered:

1. In practice, the availability of modules matter the most. The module which is ready to be integrated, will be integrated and tested first. We should not wait to test it according to depth first or breadth first sequence, but use the availability of modules.
2. If there are critical sections of the software, design the sequence such that these sections will be added and tested as early as possible. A critical section might be a complex module, a module with a new algorithm or a module suspected to be error prone.
3. Design the sequence such that the I/O modules are added as early as possible so that all interface errors will be detected earlier.

Top-Down Integration Procedure

The procedure for top-down integration process is discussed in the following steps:

1. Start with the top or initial module in the software. Substitute the stubs for all the subordinate modules of top module. Test the top module.
2. After testing the top module, stubs are replaced one at a time with the actual modules for integration.
3. Perform testing on this recent integrated environment.
4. Regression testing may be conducted to ensure that new errors have not appeared.
5. Repeat steps 2–4 for the whole design hierarchy.

The top-down integration for Fig. 7.8 is shown below.

Listed below are the drawbacks of top-down integration testing.

1. Stubs must be prepared as required for testing one module.
2. Stubs are often more complicated than they first appear.
3. Before the I/O functions are added, the representation of test cases in stubs can be difficult.

Bottom-up Integration Testing

The bottom-up strategy begins with the terminal or modules at the lowest level in the software structure. After testing these modules, they are integrated and tested moving from bottom to top level. Since the processing required for modules subordinate to a given level is always available, stubs are not required in this strategy.

Bottom-up integration can be considered as the opposite of top-down approach. Unlike top-down strategy, this strategy does not require the architectural design of the system to be complete. Thus, bottom-up integration can be performed at an early stage in the developmental process. It may be used where the system reuses and modifies components from other systems.

The steps in bottom-up integration are as follows:

1. Start with the lowest level modules in the design hierarchy. These are the modules from which no other module is being called.
2. Look for the super-ordinate module which calls the module selected in step 1. Design the driver module for this super-ordinate module.
3. Test the module selected in step 1 with the driver designed in step 2.
4. The next module to be tested is any module whose subordinate modules (the modules it calls) have all been tested.
5. Repeat steps 2 to 5 and move up in the design hierarchy.
6. Whenever, the actual modules are available, replace stubs and drivers with the actual one and test again.

Bottom-up integration for Fig. 7.8, is shown below:

This type of integration is useful for integrating object-oriented systems, real-time systems, and systems with strict performance requirements. Bottom-up integration has the disadvantage that the software as a whole does not exist until the last module is added. It is also not an optimal strategy for functionally decomposed systems, as it tests the most important subsystem last.

Comparison between Top-Down and Bottom-Up Integration Testing

Table 7.1 provides a comparison between top-down and bottom-up integration testing techniques [29].

Table 7.1 Comparison between top-down and bottom-up testing

Issue	Top-Down Testing	Bottom-Up Testing
Architectural Design	It discovers errors in high-level design, thus detects errors at an early stage.	High-level design is validated at a later stage.
System Demonstration	Since we integrate the modules from top to bottom, the high-level design slowly expands as a working system. Therefore, feasibility of the system can be demonstrated to the top management.	It may not be possible to show the feasibility of the design. However, if some modules are already built as reusable components, then it may be possible to produce some kind of demonstration.
Test Implementation	*(nodes – 1)* stubs are required for the subordinate modules.	*(nodes – leaves)* test drivers are required for super-ordinate modules to test the lower-level modules.

Practical Approach for Integration Testing

There is no single strategy adopted for industry practice. For integrating the modules, one cannot rely on a single strategy. There are situations depending on the project in hand which will force to integrate the modules by combining top-down and bottom-up techniques. This combined approach is sometimes known as *sandwich integration testing.*

Selection of an integration testing strategy depends on software characteristics and sometimes project schedules. In general, sandwich testing strategy that uses top-down tests for upper levels of the program structure with bottom-up tests for subordinate levels is the best compromise.

The practical approach for adopting sandwich testing is driven by the following factors:

Priority There may be situations which force us to prioritize the modules to be integrated. Some of the prioritizing guidelines [7] are given below:

- Integration should follow the lines of first putting together those subsystems that are of great concern or with more important requirements.
- This prioritization might dictate top-down integration if the module has a high level of control on its subordinate modules.
- The modules with more user interfaces should be tested first, as they are more error-prone.
- If the customer is anxious to see a running program early in the testing phase, then top-down integration will help.
- In another situation, the machine interface and performance might be of special interest. Here, bottom-up integration would be dictated. With this approach, we stand better chance of experiencing a high degree of concurrency during our integration.
- The module whose cyclomatic complexity is high must be tested first.

Availability In practice, the availability of modules matter the most. The module which is ready to be integrated, will be integrated and tested first. We should not wait according to top-down or bottom-up sequence, but use the availability of modules. Different development schedules for different modules of the system force us to integrate with available modules only.

Pros and Cons of Decomposition-Based Integration

Decomposition-based integration techniques are better for monitoring the progress of integration testing along the decomposition tree. If there is failure in this integration, the first suspect goes to the recently added module, as the modules are integrated one by one either in top-down or bottom-up sequence. Thus, debugging is easy in decomposition-based integration.

However, there is more effort required in this type of integration, as stubs and drivers are needed for testing. Drivers are more complicated to design as compared to stubs. The integration testing effort is computed as the number of test sessions. A test session is one set of test cases for a specific configuration.

The total number of test sessions in a decomposition-based integration is computed as:

Number of test sessions = nodes − leaves + edges

7.2.2 Call Graph-Based Integration

It is assumed that integration testing detects bugs which are structural. However, it is also important to detect some behavioural bugs. If we can refine the functional decomposition tree into a form of module calling graph, then we are moving towards behavioural testing at the integration level. This can be done with the help of a *call graph*, as given by Jorgensen [78].

A call graph is a directed graph, wherein the nodes are either modules or units, and a directed edge from one node to another means one module has called another module. The call graph can be captured in a matrix form which is known as the *adjacency matrix*. For example, see Fig. 7.9, which is a call graph of a hypothetical program. The figure shows how one unit calls another. Its adjacency matrix is shown in Fig. 7.10. This matrix may help the testers a lot.

Figure 7.9 Example call graph

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	x	x	x	x					x	x									
2																			
3																			
4																			
5					x	x	x												
6																			
7																			
8								x							x				
9																x			
10												x	x						
11											x								
12																	x		
13																		x	
14														x					
15																			x
16																			
17																			
18																			
19																			
20																			

Figure 7.10 Adjacency matrix

The call graph shown in Fig. 7.9 can be used as a basis for integration testing. The idea behind using a call graph for integration testing is to avoid the efforts made in developing the stubs and drivers. If we know the calling sequence, and if we wait for the called or calling function, if not ready, then call graph-based integration can be used.

There are two types of integration testing based on call graph which are discussed next.

Pair-wise Integration

If we consider only one pair of calling and called modules, then we can make a set of pairs for all such modules, as shown in Fig. 7.11, for pairs 1–10 and 1–11. The resulting set will be the total test sessions which will be equal to the sum of all edges in the call graph. For example, in the call graph shown in Fig. 7.11, the number of test sessions is 19 which is equal to the number of edges in the call graph.

Figure 7.11 Pair-wise integration

Neighbourhood Integration

There is not much reduction in the total number of test sessions in pair-wise integration as compared to decomposition-based integration. If we consider the neighbourhoods of a node in the call graph, then the number of test sessions may reduce. The neighbourhood for a node is the immediate predecessor as well as the immediate successor nodes. The neighbourhood of a node, thus, can be defined as the set of nodes that are one edge away from the given node.

The neighbourhoods of each node in the call graph shown in Fig. 7.9 is shown in Table 7.2.

Table 7.2 Neighbourhood integration details

Node	Neighbourhoods	
	Predecessors	Successors
1	-----	2,3,4,5,10,11
5	1	6,7,8
8	5	9,16
9	8	17
10	1	13,14
11	1	12
12	11	18
13	10	19
14	10	15
15	14	20

The total test sessions in neighbourhood integration can be calculated as:

$$\begin{aligned}\textbf{Neighbourhoods} &= \textbf{nodes} - \textbf{sink nodes}\\ &= 20 - 10\\ &= 10\end{aligned}$$

where *sink node* is an instruction in a module at which the execution terminates.

7.2.3 Path-Based Integration

As we have discussed, in a call graph, when a module or unit executes, some path of source instructions is executed (remember flow graph?). And it may be possible that in that path execution, there may be a call to another unit. At that point, the control is transferred from the calling unit to the called unit. This passing of control from one unit to another unit is necessary for integration testing. Also, there should be information within the module regarding instructions that call the module or return to the module. This must be tested at the time of integration. It can be done with the help of path-based integration defined by Paul C. Jorgenson [78]. We need to understand the following definitions for path-based integration.

Source node It is an instruction in the module at which the execution starts or resumes. The nodes where the control is being transferred after calling the module are also source nodes.

Sink node It is an instruction in a module at which the execution terminates. The nodes from which the control is transferred are also sink nodes.

Module execution path (MEP) It is a path consisting of a set of executable statements within a module like in a flow graph.

Message When the control from one unit is transferred to another unit, then the programming language mechanism used to do this is known as a *message*. For example, when there is a function call, then it is a message from one unit (where the call is mentioned; caller module) to another unit (the unit which is being called).

MM-path It is a path consisting of MEPs and messages. The path shows the sequence of executable statements; it also crosses the boundary of a unit when a message is followed to call another unit. In other words, MM-path is a set of MEPs and transfer of control among different units in the form of messages.

MM-path graph It can be defined as an extended flow graph where nodes are MEPs and edges are messages. It returns from the last called unit to the first unit where the call was made. In this graph, messages are highlighted with thick lines.

Now let us see the concept of path-based integration with the help of one example. Fig. 7.12 shows the MM-path as a darken line. The details regarding the example units shown in Fig. 7.12 is given in Table 7.3.

Figure 7.12 MM-path

Table 7.3 MM-path details

	Source Nodes	Sink Nodes	MEPs
Unit A	1,4	3,5	MEP(A,1) = <1,2,5> MEP(A,2) = <1,3> MEP(A,3) = <4,5>
Unit B	1,5	4,6	MEP(B,1) = <1,2,4> MEP(B,2) = <5,6> MEP(B,3) = <1,2,3,4,5,6>
Unit C	1	5	MEP(C,1) = <1,3,4,5> MEP(C,2) = <1,2,4,5>

The MM-path graph for this example is shown in Fig. 7.13.

Figure 7.13 MEP graph

7.3 Function Testing

When an integrated system is tested, all its specified functions and external interfaces are tested on the software. Every functionality of the system specified in the functions is tested according to its external specifications. An external specification is a precise description of the software behaviour from the viewpoint of the outside world (e.g. user). Kit [1] has defined function testing as the *process of attempting to detect discrepancies between the functional specifications of a software and its actual behaviour.*

Thus, the objective of function test is to measure the quality of the functional (business) components of the system. Tests verify that the system behaves correctly from the user/business perspective and functions according to the requirements, models, or any other design paradigm used to specify the application. The function test must determine if each component or business event:

1. performs in accordance to the specifications,
2. responds correctly to all conditions that may present themselves by incoming events/data,
3. moves data correctly from one business event to the next (including data stores), and
4. is initiated in the order required to meet the business objectives of the system.

Function testing can be performed after unit and integration testing, or whenever the development team thinks that the system has sufficient functionality to execute some tests. The test cases are executed such that the execution of a given test case against the software will exercise external functionality of

certain parts. To keep a record of function testing, a function coverage metric is used. Function coverage can be measured with a *function coverage matrix*. It keeps track of those functions that exhibited the greatest number of errors. This information is valuable because it tells us that these functions probably contain the preponderance of errors that have not been detected yet.

An effective function test cycle must have a defined set of processes and deliverables. The primary processes/deliverables for requirements based function test are discussed below.

Test planning During planning, the test leader with assistance from the test team defines the scope, schedule, and deliverables for the function test cycle. He delivers a test plan (document) and a test schedule (work plan)—these often undergo several revisions during the testing cycle.

Partitioning/functional decomposition Functional decomposition of a system (or partitioning) is the breakdown of a system into its functional components or functional areas. Another group in the organization may take responsibility for the functional decomposition (or model) of the system, but the testing organization should still review this deliverable for completeness before accepting it into the test organization. If the functional decompositions or partitions have not been defined or are deemed insufficient, then the testing organization will have to take responsibility for creating and maintaining the partitions.

Requirement definition The testing organization needs specified requirements in the form of proper documents to proceed with the function test. These requirements need to be itemized under an appropriate functional partition.

Test case design A tester designs and implements a test case to validate that the product performs in accordance with the requirements. These test cases need to be itemized under an appropriate functional partition and mapped/traced to the requirements being tested.

Traceability matrix formation Test cases need to be traced/mapped back to the appropriate requirement. A function coverage matrix is prepared. This matrix is a table, listing specific functions to be tested, the priority for testing each function, and test cases that contain tests for each function. Once all the aspects of a function have been tested by one or more test cases, then the test design activity for that function can be considered complete. This approach gives a more accurate picture of the application when coverage analysis is done. For example, in Table 7.4, function F2 test cases must be executed, as its priority is highest; and through the function coverage matrix, we can track which functions are being tested through which test cases.

Table 7.4 Function coverage matrix

Functions/Features	Priority	Test Cases
F1	3	T2,T4,T6
F2	1	T1, T3,T5

Test case execution As in all phases of testing, an appropriate set of test cases need to be executed and the results of those test cases recorded. Which test cases are to be executed should be defined within the context of the test plan and the current state of the application being tested. If the current state of the application does not support the testing of one or more functions, then this testing should be deferred until it justifies the expenditure of testing resources.

7.4 System Testing

When all the modules have been integrated with the software system, then the software is incorporated with other system elements (e.g. hardware, people, information). This integrated system needs to be validated now. But the system is not validated according to functional specifications. System testing should not be misunderstood at this point. In function testing, we have already validated all the requirement functionality. Thus, system testing is not a process of testing the functions of a complete system or program.

System testing is the process of attempting to demonstrate that a program or system does not meet its original requirements and objectives, as stated in the requirement specification. System testing is actually a series of different tests to test the whole system on various grounds where bugs have the probability to occur. The ground can be performance, security, maximum load, etc. The integrated system is passed through various tests based on these grounds and depending on the environment and type of project. After passing through these tests, the resulting system is a system which is ready for acceptance testing which involves the user, as shown in Fig. 7.14.

But it is difficult to test the system on various grounds, since there is no methodology for system testing. One solution to this problem is to think from the perspective of the user and the problem the user is trying to solve.

Unlike function testing, the external specifications cannot be used as the basis for deriving the system test cases, since this would weaken the purpose of system testing. On the other hand, the objectives document cannot be used by itself to formulate test cases, since it does not by definition, contain precise

Figure 7.14 System testing

descriptions of the program's external interfaces. This dilemma is solved by using the program's user documentation. Design the system test by analysing the objectives; formulate test cases by analysing the user documentation. This has a useful side-effect of comparing the program with its objectives and the user documentation, as well as comparing the user documentation with the objectives, as shown in Fig. 7.15.

It is obvious that the central purpose of system testing is to compare the system with its stated objectives. However, there are no test case design methodologies. The reason for this is that objectives state what a program should do and how well the program should do it, but they do not state the representation of the program's functions.

Figure 7.15 Design the system by analysing the objectives

Given the statement of objectives, there is no identifiable methodology that would yield a set of test cases, other than the vague but useful guideline of writing test cases to attempt to show that the system is inconsistent with each sentence in the objectives statement. Therefore, rather than developing a system test methodology, distinct categories of system test cases are taken. Some of the categories are discussed below.

7.4.1 Categories of System Tests

Recovery Testing

Recovery is just like the exception handling feature of a programming language. It is the ability of a system to restart operations after the integrity of the application has been lost. It reverts to a point where the system was functioning correctly and then, reprocesses the transactions to the point of failure.

Some software systems (e.g. operating system, database management systems, etc.) must recover from programming errors, hardware failures, data errors, or any disaster in the system. So the purpose of this type of system testing is to show that these recovery functions do not work correctly.

The main purpose of this test is to determine how good the developed software is when it faces a disaster. Disaster can be anything from unplugging the system which is running the software from power, network etc., also stopping the database, or crashing the developed software itself. Thus, *recovery testing is the activity of testing how well the software is able to recover from crashes, hardware failures, and other similar problems*. It is the forced failure of the software in various ways to verify that the recovery is properly performed. Some examples of recovery testing are given below:

- While the application is running, suddenly restart the computer and thereafter, check the validity of application's data integrity.
- While the application receives data from the network, unplug the cable and plug-in after awhile, and analyse the application's ability to continue receiving data from that point, when the network connection disappeared.
- Restart the system while the browser has a definite number of sessions and after rebooting, check that it is able to recover all of them.

Recovery tests would determine if the system can return to a well-known state, and that no transactions have been compromised. Systems with automated recovery are designed for this purpose. There can be provision of multiple CPUs and/or multiple instances of devices, and mechanisms to detect the failure of a device. A 'checkpoint' system can also be put that meticulously records transactions and system states periodically to preserve them in case of failure. This information allows the system to return to a known state after the failure.

Beizer [49] proposes that testers should work on the following areas during recovery testing:

Restart If there is a failure and we want to recover and start again, then first the current system state and transaction states are discarded. Following the criteria of checkpoints as discussed above, the most recent checkpoint record

is retrieved and the system is initialized to the states in the checkpoint record. Thus, by using checkpoints, a system can be recovered and started again from a new state. Testers must ensure that all transactions have been reconstructed correctly and that all devices are in proper states. The system now is in a position to begin to process new transactions.

Switchover Recovery can also be done if there are standby components and in case of failure of one component, the standby takes over the control. The ability of the system to switch to a new component must be tested.

A good way to perform recovery testing is under maximum load. Maximum load would give rise to transaction inaccuracies and the system would crash, resulting in defects and design flaws.

Security Testing

Safety and security issues are gaining importance due to the proliferation of commercial applications on the Internet and the increasing concern about privacy. Security is a protection system that is needed to assure the customers that their data will be protected. For example, if Internet users feel that their personal data/information is not secure, the system loses its accountability. Security may include controlling access to data, encrypting data in communication, ensuring secrecy of stored data, auditing security events, etc. The effects of security breaches could be extensive and can cause loss of information, corruption of information, misinformation, privacy violations, denial of service, etc.

Types of Security Requirements While performing security testing, the following security requirements must be considered:

- Security requirements should be associated with each functional requirement. Each functional requirement, most likely, has a specific set of related security issues to be addressed in the software implementation. For example, the log-on requirement in a client-server system must specify the number of retries allowed, the action to be taken if the log-on fails, and so on.
- In addition to security concerns that are directly related to particular requirements, a software project has security issues that are global in nature, and are therefore, related to the application's architecture and overall implementation. For example, a Web application may have a global requirement that all private customer data of any kind is stored in encrypted form in the database. In another example, a system-wide security requirement is to use SSL to encrypt the data sent between the client browser and the Web server. Security testing team must verify that SSL is correctly used in all such transmissions.

Security testing is the process of attempting to devise test cases to evaluate the adequacy of protective procedures and countermeasures.

The problem with security testing is that security-related bugs are not as obvious as other type of bugs. It may be possible that the security system has failed and caused the loss of information without the knowledge of loss. Thus, the tester should perform security testing with the goal to identify the bugs that are very difficult to identify.

Security vulnerabilities Vulnerability is an error that an attacker can exploit. Security vulnerabilities are of the following types:

- Bugs at the implementation level, such as local implementation errors or interprocedural interface errors
- Design-level mistakes

Design-level vulnerabilities are the hardest defect category to handle, but they are also the most prevalent and critical. Unfortunately, ascertaining whether a program has design-level vulnerabilities requires great expertise, which makes finding not only difficult but particularly hard to automate. Examples of design-level security flaws include problem in error-handling, unprotected data channels, incorrect or missing access control mechanisms, and timing errors especially in multithreaded systems.

How to perform security testing Testers must use a risk-based approach, grounded in both the system's architectural reality and the attacker's mindset, to gauge software security adequately. By identifying risks and potential loss associated with those risks in the system and creating tests driven by those risks, the tester can properly focus on areas of code in which an attack is likely to succeed. Therefore, risk analysis, especially at the design-level, can help us identify potential security problems and their impacts. Once identified and ranked, software risks can help guide software security testing.

Risk management and security testing Software security practitioners perform many different tasks to manage software security risks, including:

- Creating security abuse/misuse cases
- Listing normative security requirements
- Performing architectural risk analysis
- Building risk-based security test plans
- Wielding static analysis tools
- Performing security tests

Three tasks, i.e. architectural risk analysis, risk-based security test planning, and security testing, are closely linked because a critical aspect of security

testing relies on probing security risks. Based on design-level risk analysis and ranking of security related risks, security test plans are prepared which guide the security testing.

Thus, security testing must necessarily involve two diverse approaches:

- Testing security mechanisms to ensure that their functionality is properly implemented
- Performing risk-based security testing motivated by understanding and simulating the attacker's approach

Elements of security testing The basic security concepts that need to be covered by security testing are discussed below:

Confidentiality A security measure which protects against the disclosure of information to parties other than the intended recipient.

Integrity A measure intended to allow the receiver to determine that the information which it receives has not been altered in transit or by anyone other than the originator of the information. Integrity schemes often use some of the same underlying technologies as confidentiality schemes, but they usually involve adding additional information to a communication to form the basis of an algorithmic check rather than encoding all the communication.

Authentication A measure designed to establish the validity of a transmission, message, or originator. It allows the receiver to have confidence that the information it receives originates from a specific known source.

Authorization It is the process of determining that a requester is allowed to receive a service or perform an operation. Access control is an example of authorization.

Availability It assures that the information and communication services will be ready for use when expected. Information must be kept available for authorized persons when they need it.

Non-repudiation A measure intended to prevent the later denial that an action happened, or a communication took place, etc. In communication terms, this often involves the interchange of authentication information combined with some form of provable timestamp.

Performance Testing

Performance specifications (requirements) are documented in a performance test plan. Ideally, this is done during the requirements development phase of any system development project, prior to any design effort.

Performance specifications (requirements) should ask the following questions, at a minimum:

- In detail, what is the performance test scope? What subsystems, interfaces, components, etc. are in and out of the scope for this test?
- For the user interfaces (UIs) involved, how many concurrent users are expected for each (specify peak vs. nominal)?
- What does the target system (hardware) look like (specify all server and network appliance configurations)?
- What are the time requirements for any/all backend batch processes (specify peak vs. nominal)?

A system along with its functional requirements, must meet the quality requirements. One example of quality requirement is performance level. The users may have objectives for a software system in terms of memory use, response time, throughput, and delays. Thus, performance testing is to test the run-time performance of the software on the basis of various performance factors. Performance testing becomes important for real-time embedded systems, as they demand critical performance requirements.

This testing requires that performance requirements must be clearly mentioned in the SRS and system test plans. The important thing is that these requirements must be quantified. For example, a requirement that the system should return a response to a query in a reasonable amount of time is not an acceptable requirement; the time must be specified in a quantitative way. In another example, for a Web application, you need to know at least two things: (a) expected load in terms of concurrent users or HTTP connections and (b) acceptable response time.

Performance testing is often used as a part of the process of performance profile tuning. The goal is to identify the 'weakest links'—the system often carries a number of parts which, with a little tweak, can significantly improve the overall performance of the system. It is sometimes difficult to identify which parts of the system represent the critical paths. To help identify critical paths, some test tools include (or have as add-ons) instrumentation agents that run on the server and report transaction times, database access times, network overhead, and other server monitors. Without such instrumentation, the use of primitive system tools may be required (e.g. Task Manager in Microsoft Windows). Performance testing usually concludes that it is the software (rather than the hardware) that contribute most to delays (bottlenecks) in data processing.

For performance testing, generally a generic set of sample data is created in the project and because of time constraints, that data is used throughout

development and functional testing. However, these datasets tend to be unrealistic, with insufficient size and variation, which may result in performance surprises when an actual customer, using much more data than contained in the sample dataset, attempts to use the system. Therefore, it becomes necessary that testing team must use production-size realistic databases that include a wide range of possible data combinations.

Using realistic-size databases provides the following benefits:

- Since a large dataset may require significant disk space and processor power, it leads to backup and recoverability concerns.
- If data is transferred across a network, bandwidth may also be a consideration.

Thus, working with realistic datasets as early as possible will help bring these issues to the surface while there is time and when it is more cost-effective to do something about them, rather than after the software is already in production environment, when such issues can become very costly to address in terms of both budget and schedule.

The following tasks must be done for this testing:

- Decide whether to use internal or external resources to perform tests, depending on in-house expertise (or the lack of it).
- Gather performance requirements (specifications) from users and/or business analysts.
- Develop a high-level plan (or project charter), including requirements, resources, timelines, and milestones.
- Develop a detailed performance test plan (including detailed scenarios and test cases, workloads, environment info, etc).
- Choose test tool(s).
- Specify test data needed.
- Develop detailed performance test project plan, including all dependencies and associated timelines.
- Configure the test environment (ideally identical hardware to the production platform), router configuration, deployment of server instrumentation, database test sets developed, etc.
- Execute tests, probably repeatedly (iteratively), in order to see whether any unaccounted factor might affect the results.

Load Testing

Normally, we don't test the system with its full load, i.e. with maximum values of all resources in the system. The normal system testing takes into consider-

ation only nominal values of the resources. However, there is high probability that the system will fail when put under maximum load. Therefore, it is important to test the system with all its limits. *When a system is tested with a load that causes it to allocate its resources in maximum amounts, it is called load testing.* The idea is to create an environment more demanding than the application would experience under normal workloads.

When you are doing performance testing, it may not be possible that you are taking all statistically representative loads for all the parameters in the system. For example, the system is designed for n number of users, t number of transactions per user per unit time, with p unit time response time, etc. Have you tested the performance of the system with all these representative loads? If no, then load testing must be performed as a part of performance testing. Load is varied from a minimum (zero) to the maximum level the system can sustain without running out of resources. As the load is being increased, transactions may suffer (application-specific) excessive delay. For example, when many users work simultaneously on a web server, the server responds slow. In this way, through load testing, we are able to determine the maximum sustainable load the system can handle.

Stress Testing

Stress testing is also a type of load testing, but the difference is that the system is put under loads beyond the limits so that the system breaks. Thus, *stress testing tries to break the system under test by overwhelming its resources in order to find the circumstances under which it will crash.* The areas that may be stressed in a system are:

- Input transactions
- Disk space
- Output
- Communications
- Interaction with users

Stress testing is important for real-time systems where unpredictable events may occur, resulting in input loads that exceed the values described in the specification, and the system cannot afford to fail due to maximum load on resources. Therefore, in real-time systems, the entire threshold values and system limits must be noted carefully. Then, the system must be stress-tested on each individual value.

Stress testing demands the amount of time it takes to prepare for the test and the amount of resources consumed during the actual execution of the test. Therefore, stress testing cost must be weighed against the risk of not identify-

ing volume-related failures. For example, many of the defense systems which are real-time in nature, demand the systems to perform continuously in warfare conditions. Thus, for a real-time defense system, we must stress-test the system; otherwise, there may be loss of equipment as well as life.

Usability Testing

This type of system testing is related to a system's presentation rather than its functionality. System testing can be performed to find human-factors or usability problems on the system. The idea is to adapt the software to users' actual work styles rather than forcing the users to adapt according to the software. Thus, the goal of usability testing is to verify that intended users of the system are able to interact properly with the system while having a positive and convenient experience. *Usability testing identifies discrepancies between the user interfaces of a product and the human engineering requirements of its potential users.*

What the user wants or expects from the system can be determined using several ways as proposed by Dustin [12]:

Area experts The usability problems or expectations can be best understood by the subject or area experts who have worked for years in the same area. They analyse the system's specific requirements from the user's perspective and provide valuable suggestions.

Group meetings Group meeting is an interesting idea to elicit usability requirements from the user. These meetings result in potential customers' comments on what they would like to see in an interface.

Surveys Surveys are another medium to interact with the user. It can also yield valuable information about how potential customers would use a software product to accomplish their tasks.

Analyse similar products We can also analyse the experiences in similar kinds of projects done previously and use it in the current project. This study will also give us clues regarding usability requirements.

Usability characteristics against which testing is conducted are discussed below:

Ease of use The users enter, navigate, and exit with relative ease. Each user interface must be tailored to the intelligence, educational background, and environmental pressures of the end-user.

Interface steps User interface steps should not be misleading. The steps should also not be very complex to understand either.

Response time The time taken in responding to the user should not be so high that the user is frustrated or will move to some other option in the interface.

Help system A good user interface provides help to the user at every step. The help documentation should not be redundant; it should be very precise and easily understood by every user of the system.

Error messages For every exception in the system, there must be an error message in text form so that users can understand what has happened in the system. Error messages should be clear and meaningful.

An effective tool in the development of a usable application is the user-interface prototype. This kind of prototype allows interaction between potential users, requirements personnel, and developers to determine the best approach to the system's interface. Prototypes are far superior because they are interactive and provide a more realistic preview of what the system will look like.

Compatibility/Conversion/Configuration Testing

Compatibility testing is to check the compatibility of a system being developed with different operating system, hardware and software configuration available, etc. Many software systems interact with multiple CPUs. Some software control real-time process and embedded software interact with devices. In many cases, users require that the devices be interchangeable, removable, or re-configurable. Very often the software will have a set of commands or menus that allow users to make these configuration changes. Configuration testing allows developers/testers to evaluate system performance and availability when hardware exchanges and reconfigurations occur. The number of possible combinations for checking the compatibilities of available hardware and software can be too high, making this type of testing a complex job. Discussed below are some guidelines for compatibility testing [12].

Operating systems The specifications must state all the targeted end-user operating systems on which the system being developed will be run.

Software/Hardware The product may need to operate with certain versions of Web browsers, with hardware devices such as printers, or with other softwares such as virus scanners or word processors.

Conversion testing Compatibility may also extend to upgrades from previous versions of the software. Therefore, in this case, the system must be upgraded properly and all the data and information from the previous version should also be considered. It should be specified whether the new system will be backward compatible with the previous version. Also, if other user's preferences or settings are to be preserved or not.

Ranking of possible configurations Since there will be a large set of possible configurations and compatibility concerns, the testers must rank the possible configurations in order, from the most to the least common, for the target system.

Identification of test cases Testers must identify appropriate test cases and data for compatibility testing. It is usually not feasible to run the entire set of possible test cases on every possible configuration, because this would take too much time. Therefore, it is necessary to select the most representative set of test cases that confirms the application's proper functioning on a particular platform.

Updating the compatibility test cases The compatibility test cases must also be continually updated with the following:

(i) Tech-support calls provide a good source of information for updating the compatibility test suite.

(ii) A beta-test program can also provide a large set of data regarding real end-user configurations and compatibility issues prior to the final release of a system.

7.5 Acceptance Testing

Developers/testers must keep in mind that the software is being built to satisfy the user requirements and no matter how elegant its design is, it will not be accepted by the users unless it helps them achieve their goals as specified in the requirements. After the software has passed all the system tests and defect repairs have been made, the customer/client must be involved in the testing with proper planning. The purpose of acceptance testing is to give the end-user a chance to provide the development team with feedback as to whether or not the software meets their needs. Ultimately, it's the user that needs to be satisfied with the application, not the testers, managers, or contract writers.

Acceptance testing is one of the most important types of testing we can conduct on a product. It is more important to worry whether users are happy about the way a program works rather than whether or not the program passes a bunch of tests that were created by testers in an attempt to validate the requirements, that an analyst did their best to capture and a programmer interpreted based on their understanding of those requirements.

Thus, *acceptance testing is the formal testing conducted to determine whether a software system satisfies its acceptance criteria and to enable buyers to determine whether to accept the system or not.*

Acceptance testing must take place at the end of the development process. It consists of tests to determine whether the developed system meets the prede-

termined functionality, performance, quality, and interface criteria acceptable to the user. Therefore, the final acceptance acknowledges that the entire software product adequately meets the customer's requirements. User acceptance testing is different from system testing. System testing is invariably performed by the development team which includes developers and testers. User acceptance testing, on the other hand, should be carried out by end-users.

Thus, acceptance testing is designed to:

- Determine whether the software is fit for the user.
- Making users confident about the product.
- Determine whether a software system satisfies its acceptance criteria.
- Enable the buyer to determine whether to accept the system or not.

The final acceptance marks the completion of the entire development process. It happens when the customer and the developer has no further problems.

Acceptance test might be supported by the testers. It is very important to define the acceptance criteria with the buyer during various phases of SDLC. A well-defined acceptance plan will help development teams to understand users' needs. The acceptance test plan must be created or reviewed by the customer. The development team and the customer should work together and make sure that they:

- Identify interim and final products for acceptance, acceptance criteria, and schedule.
- Plan how and by whom each acceptance activities will be performed.
- Schedule adequate time for the customer to examine and review the product.
- Prepare the acceptance plan.
- Perform formal acceptance testing at delivery.
- Make a decision based on the results of acceptance testing.

Entry Criteria

- System testing is complete and defects identified are either fixed or documented.
- Acceptance plan is prepared and resources have been identified.
- Test environment for the acceptance testing is available.

Exit Criteria

- Acceptance decision is made for the software.
- In case of any warning, the development team is notified.

Types of Acceptance Testing

Acceptance testing is classified into the following two categories:

- *Alpha Testing* Tests are conducted at the development site by the end users. The test environment can be controlled a little in this case.
- *Beta Testing* Tests are conducted at the customer site and the development team does not have any control over the test environment.

7.5.1 Alpha Testing

Alpha is the test period during which the product is complete and usable in a test environment, but not necessarily bug-free. It is the final chance to get verification from the customers that the tradeoffs made in the final development stage are coherent.

Therefore, alpha testing is typically done for two reasons:

(i) to give confidence that the software is in a suitable state to be seen by the customers (but not necessarily released).

(ii) to find bugs that may only be found under operational conditions. Any other major defects or performance issues should be discovered in this stage.

Since alpha testing is performed at the development site, testers and users together perform this testing. Therefore, the testing is in a controlled manner such that if any problem comes up, it can be managed by the testing team.

Entry Criteria to Alpha

- All features are complete/testable (no urgent bugs).
- High bugs on primary platforms are fixed/verified.
- 50% of medium bugs on primary platforms are fixed/verified.
- All features have been tested on primary platforms.
- Performance has been measured/compared with previous releases (user functions).
- Usability testing and feedback (ongoing).
- Alpha sites are ready for installation.

Exit Criteria from Alpha

After alpha testing, we must:

- Get responses/feedbacks from customers.
- Prepare a report of any serious bugs being noticed.
- Notify bug-fixing issues to developers.

7.5.2 Beta Testing

Once the alpha phase is complete, development enters the beta phase. Beta is the test period during which the product should be complete and usable in a production environment. The purpose of the beta ship and test period is to test the company's ability to deliver and support the product (and not to test the product itself). Beta also serves as a chance to get a final 'vote of confidence' from a few customers to help validate our own belief that the product is now ready for volume shipment to all customers.

Versions of the software, known as beta-versions, are released to a limited audience outside the company. The software is released to groups of people so that further testing can ensure the product has few or no bugs. Sometimes, beta-versions are made available to the open public to increase the feedback field to a maximal number of future users.

Testing during the beta phase, informally called beta testing, is generally constrained to black-box techniques, although a core of test engineers are likely to continue with white-box testing parallel to beta tests. Thus, the term *beta test* can refer to a stage of the software–closer to release than being 'in alpha'–or it can refer to the particular group and process being done at that stage. So a tester might be continuing to work in white-box testing while the software is 'in beta' (a stage), but he or she would then not be a part of 'the beta test' (group/activity).

Entry Criteria to Beta

- Positive responses from alpha sites.
- Customer bugs in alpha testing have been addressed.
- There are no fatal errors which can affect the functionality of the software.
- Secondary platform compatibility testing is complete.
- Regression testing corresponding to bug fixes has been done.
- Beta sites are ready for installation.

Guidelines for Beta Testing

- Don't expect to release new builds to beta testers more than once every two weeks.
- Don't plan a beta with fewer than four releases.
- If you add a feature, even a small one, during the beta process, the clock goes back to the beginning of eight weeks and you need another 3–4 releases.

Exit Criteria from Beta

After beta testing, we must:

- Get responses/feedbacks from the beta testers.
- Prepare a report of all serious bugs.
- Notify bug-fixing issues to developers.

SUMMARY

This chapter describes all the validation testing activities. The test plans for all validation testing types are done in advance while doing verification at every stage of SDLC. Validation testing is done at three levels: unit, integration of all modules, and system. Unit validation cannot be done in isolation. It is performed with the help of drivers and stubs which increase the cost of the project. But they are unavoidable if testing has to be performed of a module that gets input from another module which is not ready, or that calls other modules which are not ready.

The next level is integration of all the unit tested modules and then test the integrated modules. Validation testing continues at the system level which has various types depending upon the project, user requirements, etc. The integrated system must be tested on these grounds to test its performance, functionality, etc.

The final tested system then goes to another level of validation, i.e. testing the system with the user so that he accepts it according to his satisfaction. This is known as acceptance testing. Acceptance testing is performed at the developer's site as well as the customer's own site. When the customer is satisfied at his own site, then the final system is installed in the operation status. All these validation activities have been discussed in detail in this chapter.

These validation testing activities are necessary to produce a better system. If we do not perform level-wise validation testing, the system will have many bugs which are also difficult to find. But if we test the system at every level, we reduce the chances of bugs. For example, if we perform unit validation, we test every module's functionality. When we integrate the modules and test them, we uncover the interface bugs, if they are present. Similarly, when we perform system testing on various grounds like security, stress, performance, we try to make a system which will work in severe conditions as a bug-proof system.

Let us review the important concepts described in this chapter:

- Unit testing ensures that the software meets at least a baseline level of functionality at a module-level prior to integration and system testing.
- A test driver can be defined as a software module which is used to invoke a module under test and provide test inputs, control and monitor execution, and report test results or most simplistically, a line of code that calls a method and passes that method a value.
- A stub can be defined as a piece of software that works similar to a unit which is referenced by the unit being tested, but it is much simpler than the actual unit. It works as a 'stand-in' for the subordinate unit and provides the minimum required behaviour for that unit.
- Integration testing focuses on bugs caused by interfacing between the modules while integrating them.

- Various types of integration testing can be seen in a hierarchical tree given below:

- Decomposition-based integration is based on the decomposition of design into functional components or modules in a hierarchical tree form consisting of modules as nodes and interfaces between them as edges between those nodes. Thus, the interfaces among separately tested modules are tested.
- In non-incremental integration testing, either all untested modules are combined together and then tested, or unit-tested modules are combined together.
- In incremental integration testing, start with one module and unit-test it. Then combine the module which has to be merged with it and test both the modules. In this way, incrementally keep on adding the modules and test the recent environment.
- The strategy in top-down integration is to look at the design hierarchy from top to bottom. Start with high-level modules and move downward through the design hierarchy.
- The bottom-up strategy begins with terminal nodes or modules at the lowest level in the software structure. After testing these modules, they are integrated and tested, moving from bottom to top level.
- The total test sessions in a decomposition-based integration is computed as:

  Number of test sessions = nodes – leaves + edges

- Call graph-based integration uses call graphs, wherein the nodes are either modules or units, and a directed edge from one node to another means one module has called another module. The graph and its corresponding adjacency matrix help the testers in integration testing.
- Function testing is the process of attempting to detect discrepancies between the functional specifications of a software and its actual behaviour.
- System testing is the process of attempting to demonstrate that a program or system

does not meet its original requirements and objectives, as stated in the requirement specification.

- Recovery is the ability of a system to restart operations after the integrity of the application has been lost.
- Security is a protection system that is needed to secure confidential information and to assure the customer that their data will be protected.
- Stress testing tries to break the system under test by overwhelming its resources and finding the circumstances under which it will crash.
- Performance testing is to test the system against all performance requirements mentioned in the SRS. It is often used as a part of performance profile tuning. The goal is to identify the 'weakest links' — the system often carries a number of parts which, with a little tweak, can significantly improve the overall performance of the system.
- Usability testing identifies discrepancies between the user interfaces of a product and the human engineering requirements of its potential users.
- Compatibility testing is to check the compatibility of a system being developed with different operating systems, hardware and software configurations available, etc.
- Configuration testing allows developers/testers to evaluate system performance and availability, when hardware exchanges and reconfigurations occur.
- Acceptance testing is the formal testing conducted to determine whether a software system satisfies its acceptance criteria and to enable buyers to determine whether to accept the system or not. It is of two types: alpha testing and beta testing.
- Alpha testing is performed with customers at the developers' site.
- Beta testing is performed by customers at their own site.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Software validation is achieved through a series of _____ tests that demonstrate conformity with requirements.
   (a) white-box
   (b) black-box
   (c) unit tests
   (d) none of the above
2. Before we validate the entire software, _____ must be validated first.
   (a) modules
   (b) system
   (c) functionality
   (d) all of the above
3. Unit tests ensure that the software meets at least a _____ of functionality prior to integration and system testing.
   (a) high-level
   (b) low-level

(c) baseline level
(d) none of the above

4. Two types of interface modules which must be simulated, if required, to test the module are _____.
   (a) unit and integration
   (b) simulators and emulators
   (c) stubs and drivers
   (d) none of the above
5. Overhead of stubs and drivers may increase the _____ of the entire software system.
   (a) test cases
   (b) time
   (c) cost
   (d) time and cost
6. Integration of modules is according to the _____ of software.
   (a) design
   (b) coding
   (c) specifications
   (d) all of the above
7. Recovery is the ability of a system to _____ operations after the integrity of the application has been lost.
   (a) suspend
   (b) observe
   (c) restart
   (d) all of the above
8. A system that meticulously records transactions and system states periodically so that these are preserved in case of a failure is called a _____.
   (a) checkpoint
   (b) transaction system
   (c) recovery system
   (d) none of the above
9. Security requirements should be associated with each _____ requirement.
   (a) functional
   (b) design
   (c) coding
   (d) testing
10. Measures intended to allow the receiver to determine that the information which it receives has not been altered in transit is known as _____.
   (a) confidentiality
   (b) integrity

(c) authentication
(d) none of the above

11. The process of determining that a requester is allowed to receive a service or perform an operation is called _____.
    (a) confidentiality
    (b) integrity
    (c) authentication
    (d) authorization
12. A measure intended to prevent the later denial that an action happened, or a communication took place is called _____.
    (a) confidentiality
    (b) integrity
    (c) non-repudiation
    (d) authorization
13. The type of system testing related to a system's presentation rather than its functionality is called _____.
    (a) usability testing
    (b) stress testing
    (c) conversion testing
    (d) all of the above
14. A system test should not be performed by _____.
    (a) programmers
    (b) testers
    (c) designers
    (d) all of the above
15. Acceptance testing must occur at the _____ of the development process.
    (a) start
    (b) end
    (c) middle
    (d) none of the above
16. Top-down integration testing requires _____ stubs.
    (a) nodes + 2
    (b) nodes – 1
    (c) nodes + 1
    (d) none
17. Bottom-up integration testing requires _____ drivers.
    (a) nodes + 2
    (b) nodes + leaves
    (c) nodes – leaves
    (d) none

18. Total number of sessions in decomposition-based integration testing is _____.
    (a) nodes + 2
    (b) nodes + leaves
    (c) nodes – leaves + edges
    (d) none of the above
19. The total number of sessions in a pair-wise call graph-based integration testing is _____.
    (a) total edges in the graph + 2
    (b) nodes + leaves
    (c) nodes – leaves + edges
    (d) total edges in the graph
20. Total number of sessions in neighbourhood call graph-based integration testing is _____.
    (a) total edges in the graph + 2
    (b) nodes + sink nodes
    (c) nodes – leaves + edges
    (d) nodes – sink nodes
21. The nodes where the control is being transferred after calling the module, are called _____.
    (a) sink nodes
    (b) source nodes
    (c) message
    (d) none of the above
22. The nodes from which the control is transferred are called _____.
    (a) sink nodes
    (b) source nodes
    (c) message
    (d) none of the above
23. When the control from one unit is transferred to another unit, then the programming language mechanism used to do this is known as
    (a) sink nodes
    (b) source nodes
    (c) message
    (d) none of the above
24. A call graph is a _____.
    (a) undirected graph
    (b) cyclic graph
    (c) directed graph
    (d) none of the above

Review Questions

1. Consider Problem 5.12. (a) Suppose main() module is not ready for the testing of CheckPrime() module. Design driver module for main(). (b) Module CheckPrime() is not ready when called in main(). Design stubs for this module.
2. Perform top-down and bottom-up integration procedure from the following system hierarchy.

3. Calculate the number of test sessions for the decomposition tree shown in Problem 2.
4. Consider Problem 5.12 again. Perform path-based integration calculating source nodes, sink nodes, MEPs and MM-path graph.
5. What are the factors that guide sandwich integration testing?
6. 'Design the system test by analysing the objectives.' Comment on this statement.
7. What is recovery testing? Give some examples.
8. What is risk-based security testing?
9. What are the elements of security testing?
10. What is the importance of realistic size databases in performance testing?
11. What is the difference between alpha and beta testing?
12. What is the entry and exit criteria for alpha and beta testing?
13. Calculate the number of test sessions for the pair-wise and neighbourhood call graph-based integration testing for the call graph shown below.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter

8

Regression Testing

OBJECTIVES

After reading this chapter, you should be able to understand:

- Regression testing is necessary to maintain software whenever it is updated
- Regression testing increases the quality of software
- Objectives of regression testing
- Types of regression testing
- Regression testability
- Problems of regression testing
- Regression testing techniques

In Chapter 7, integration testing was discussed as one of the validation activity. However, when a new module is added as a part of integration testing, the software changes. New data flow paths are established, new I/O may occur, etc. These changes, due to addition of new modules, may affect the functioning of earlier integrated modules which otherwise were functioning flawlessly. Moreover, the software may also change whenever a new bug in the working system appears and it needs some changes. The new modifications may affect other parts of the software too. It means that whenever the software configuration changes, there is a probability of other modules getting affected due to bug fixing. In this case, it becomes important to test the whole software again with all the test cases so that a new modification in a certain part of the software does not affect other parts of the software.

8.1 Progressive vs. Regressive Testing

Whatever test case design methods or testing techniques, discussed until now, have been referred to as progressive testing or development testing. From verification to validation, the testing process progresses towards the release of the product. However, to maintain the software, bug fixing may be required during any stage of development and therefore, there is a need to check the software again to validate that there has been no adverse effect on the already working software. A system under test (SUT) is said to *regress* if

- a modified component fails, or
- a new component, when used with unchanged components, causes failures in the unchanged components by generating side-effects or feature interactions.

Therefore, now the following versions will be there in the system:

Baseline version The version of a component (system) that has passed a test suite.

Delta version A changed version that has not passed a regression test.

Delta build An executable configuration of the SUT that contains all the delta and baseline components.

Thus, it can be said that most test cases begin as progressive test cases and eventually become regression test cases. Regression testing is not another testing activity. Rather, it is the re-execution of some or all of the already developed test cases.

Definition

The purpose of regression testing is to ensure that bug-fixes and new functionalities introduced in a new version of the software do not adversely affect the correct functionality inherited from the previous version. IEEE software glossary defines regression testing as follows [44]:

Regression testing is the selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.

After a program has been modified, we must ensure that the modifications work correctly and check that the unmodified parts of the program have not been adversely affected by these modifications. It may be possible that small changes in one part of a program may have subtle undesired effects on other unrelated modules of the software. It is not necessary that if you are getting the desired outputs on the screen, then there is no problem. It may produce incorrect outputs on other test cases on which the original software produced correct outputs earlier. Thus, during regression testing, the modified software is executed on all tests to validate that it still behaves the same as it did in the original software, except where a change is expected.

Thus, regression testing can be defined as the *software maintenance task performed on a modified program to instill confidence that changes are correct and have not adversely affected the unchanged portions of the program.*

8.2 Regression Testing produces Quality Software

As discussed before, regression testing is performed in case of bug-fixing or whenever there is a need to incorporate any new requirements in the software. After the first release, whenever there is a need to fix the bugs that have

been reported or if there is any updation in the requirement itself, the developer fixes the bug or incorporates the new requirement by changing the code somewhere. This changed software requires to be fully tested again so as to ensure: (a) bug-fixing has been carried out successfully, (b) the modifications have not affected other unchanged modules, and (c) the new requirements have been incorporated correctly. (see Fig. 8.1). It means that every time you make a change in the software for whatever reasons, you have to perform a set of regression tests to validate the software. This process of executing the whole set of test again and again may be time-consuming and frustrating, but it increases the quality of software. Therefore, the creation, maintenance, and execution of a regression test suite helps to retain the quality of the software. Industry experiences noted by Onoma *et al.* [45] indicate that regression testing often has a strong positive influence on software quality. Indeed, the importance of regression testing is well-understood for the following reasons:

- It validates the parts of the software where changes occur.
- It validates the parts of the software which may be affected by some changes, but otherwise unrelated.
- It ensures proper functioning of the software, as it was before changes occurred.
- It enhances the quality of software, as it reduces the risk of high-risk bugs.

Figure 8.1 Regression testing produces Quality Software

8.3 Regression Testability

Regression testability refers to the property of a program, modification, or test suite that lets it be effectively and efficiently regression-tested. Leung and White [47] classify a program as regression testable if most single statement

modifications to the program entail rerunning a small proportion of the current test suite. Under this definition, regression testability is a function of both the design of the program and the test suite.

To consider regression testability, a regression number is computed. It is the average number of affected test cases in the test suite that are affected by any modification to a single instruction. This number is computed using information about the test suite coverage of the program.

If regression testability is considered at an early stage of development, it can provide significant savings in the cost of development and maintenance of the software.

8.4 Objectives of Regression Testing

It tests to check that the bug has been addressed The first objective in bug-fix testing is to check whether the bug-fixing has worked or not. Therefore, you run exactly the same test that was executed when the problem was first found. If the program fails on this testing, it means the bug has not been fixed correctly and there is no need to do any regression testing further.

If finds other related bugs It may be possible that the developer has fixed only the symptoms of the reported bugs without fixing the underlying bug. Moreover, there may be various ways to produce that bug. Therefore, regression tests are necessary to validate that the system does not have any related bugs.

It tests to check the effect on other parts of the program It may be possible that bug-fixing has unwanted consequences on other parts of a program. Therefore, regression tests are necessary to check the influence of changes in one part on other parts of the program.

8.5 When is Regression Testing Done?

Software Maintenance

Corrective maintenance Changes made to correct a system after a failure has been observed (usually after general release).

Adaptive maintenance Changes made to achieve continuing compatibility with the target environment or other systems.

Perfective maintenance Changes designed to improve or add capabilities.

Preventive maintenance Changes made to increase robustness, maintainability, portability, and other features.

Rapid Iterative Development

The *extreme programming* approach requires that a test be developed for each class and that this test be re-run every time the class changes.

First Step of Integration

Re-running accumulated test suites, as new components are added to successive test configurations, builds the regression suite incrementally and reveals regression bugs.

Compatibility Assessment and Benchmarking

Some test suites are designed to be run on a wide range of platforms and applications to establish conformance with a standard or to evaluate time and space performance. These test suites are meant for regression testing, but not intended to reveal regression bugs.

8.6 Regression Testing Types

Bug-Fix regression This testing is performed after a bug has been reported and fixed. Its goal is to repeat the test cases that expose the problem in the first place.

Side-Effect regression/Stability regression It involves retesting a substantial part of the product. The goal is to prove that the change has no detrimental effect on something that was earlier in order. It tests the overall integrity of the program, not the success of software fixes.

8.7 Defining Regression Test Problem

Let us first define the notations used in regression testing before defining the regression test problem.

P denotes a program or procedure,

P' denotes a modified version of P,

S denotes the specification for program P,

S' denotes the specification for program P',

$P(i)$ refer to the output of P on input i,

$P'(i)$ refer to the output of P' on input i, and

$T = \{t_1, \ldots\ldots t_n\}$ denotes a test suite or test set for P.

8.7.1 Is Regression Testing a Problem?

Regression testing is considered a problem, as the existing test suite with probable additional test cases needs to be tested again and again whenever there is a modification. The following difficulties occur in retesting:

- Large systems can take a long time to retest.
- It can be difficult and time-consuming to create the tests.
- It can be difficult and time-consuming to evaluate the tests. Sometimes, it requires a person in the loop to create and evaluate the results.
- Cost of testing can reduce resources available for software improvements.

8.7.2 Regression Testing Problem

Given a program P, its modified version P', and a test set T that was used earlier to test P; find a way to utilize T to gain sufficient confidence in the correctness of P'.

8.8 Regression Testing Techniques

There are three different techniques for regression testing. They are discussed below.

Regression test selection technique This technique attempt to reduce the time required to retest a modified program by selecting some subset of the existing test suite.

Test case prioritization technique Regression test prioritization attempts to reorder a regression test suite so that those tests with the highest priority, according to some established criteria, are executed earlier in the regression testing process rather than those with lower priority. There are two types of prioritization:

(a) **General Test Case Prioritization** For a given program P and test suite T, we prioritize the test cases in T that will be useful over a succession of subsequent modified versions of P, without any knowledge of the modified version.

(b) **Version-Specific Test Case Prioritization** We prioritize the test cases in T, when P is modified to P', with the knowledge of the changes made in P.

Test suite reduction technique It reduces testing costs by permanently eliminating redundant test cases from test suites in terms of codes or functionalities exercised.

8.8.1 Selective Retest Technique

Software maintenance includes more than 60% of development costs. In that case, testing costs dominate because many systems require labour-intensive manual testing. Selective retest techniques attempt to reduce the cost of testing by identifying the portions of P' that must be exercised by the regression test suite. Selective retesting is distinctly different from a retest-all approach that always executes every test in an existing *regression test suite* (RTS). Thus, the objective of selective retest technique is *cost reduction.* It is the process of selecting a subset of the regression test suite that tests the changes.

Following are the characteristic features of the selective retest technique:

- It minimizes the resources required to regression test a new version.
- It is achieved by minimizing the number of test cases applied to the new version.
- It is needed because a regression test suite grows with each version, resulting in broken, obsolete, uncontrollable, redundant test cases.
- It analyses the relationship between the test cases and the software elements they cover.
- It uses the information about changes to select test cases.

Rothermel and Harrold [50,51,52] have also characterized the typical steps taken by this technique in the following manner (see Fig. 8.2):

Figure 8.2 Selective Retest Technique

1. Select T' subset of T, a set of test cases to execute on P'.
2. Test P' with T', establishing correctness of P' with respect to T'.
3. If necessary, create T'', a set of new functional or structural test cases for P'.
4. Test P' with T'', establishing correctness of P' with respect to T''.
5. Create T''', a new test suite and test execution profile for P', from T, T', and T''.

Each of these steps involves the following important problems:

Regression test selection problem Step 1 involves this problem. The problem is to select a subset T' of T with which P' will be tested.

Coverage identification problem Step 3 addresses this problem. The problem is to identify portions of P' or its specifications that requires additional testing.

Test suite execution problem Steps 2 and 4 address the test suite execution problem. The problem is to execute test suites efficiently and checking test results for correctness.

Test suite maintenance problem Step 5 addresses this problem. The problem is to update and store test information.

Strategy for Test Case Selection

Effective testing requires strategies to trade-off between the two opposing needs of amplifying testing thoroughness on one side (for which a high number of test cases would be desirable) and reducing time and costs on the other (for which the fewer the test cases, the better). Given that test resources are limited, how the test cases are selected is of crucial importance. Indeed, the problem of test cases selection has been the largely dominating topic in software testing research to the extent that in the literature 'software testing' is often taken as synonymous for 'test case selection'. A decision procedure for selecting the test cases is provided by a *test criterion.*

Selection Criteria Based on Code

The motivation for code-based testing is that potential failures can only be detected if the parts of code that can cause faults are executed. All the code-based regression test selection techniques attempt to select a subset T' of T that will be helpful in establishing confidence that $P's$ functionality has been preserved where required. In this sense, all code-based test selection techniques are concerned with locating tests in T that expose faults in P'. The following tests are based on these criteria.

Fault-revealing test cases A test case t detects a fault in P' if it causes P' to fail. Hence t is fault-revealing for P'. There is no effective procedure to find the tests in T that are fault-revealing for P'. Under certain conditions, however, a regression test selection technique can select a superset of the tests in T that are fault-revealing for P'. Under these conditions, such a technique omits no tests in T that can reveal faults in P'.

Modification-revealing test cases A test case t is modification-revealing for P and P' if and only if it causes the outputs of P and P' to differ.

Modification-traversing test cases A test case t is modification-traversing if and only if it executes new or modified code in P'. These tests are useful to consider because a non-obsolete test t in T can only be modification-revealing for P and P' if it is modification-traversing for P and P'.

Regression Test Selection Techniques

A variety of regression test selection techniques have been described in the research literature. We consider the following selective retest techniques:

Minimization techniques Minimization-based regression test selection techniques attempt to select minimal sets of test cases from T that yield coverage of modified or affected portions of P. For example, the technique of Fischer *et al.* [53] uses systems of linear equations to express relationships between test cases and basic blocks (single-entry, single-exit, sequences of statements in a procedure). The technique uses a 0-1 integer programming algorithm to identify a subset T' of T that ensures that every segment that is statically reachable from a modified segment is exercised by at least one test case in T' that exercises the modified segment.

Dataflow techniques Dataflow coverage-based regression test selection techniques select test cases that exercise data interactions that have been affected by modifications. For example, the technique of Harrold and Soffa [54] requires that every definition-use pair that is deleted from P, new in P', or modified for P', should be tested. The technique selects every test case in T that, when executed on P, exercised, deleted, or modified definition-use pairs, or executed a statement containing a modified predicate.

Safe techniques Most regression test selection techniques, minimization and dataflow techniques among them, are not designed to be safe. Techniques that are not safe can fail to select a test case that would have revealed a fault in the modified program. In contrast, when an explicit set of safety conditions can be satisfied, safe regression test selection techniques guarantee that the selected subset T' contains all test cases in the original test suite T that can reveal faults in P'.

For example, the technique of Rothermel and Harrold [55] uses control-flow graph representations of P and P' and test execution profiles gathered on P, to select every test case in T that, when executed on P, exercised at least one statement that has been deleted from P, or that, when executed on P', will exercise at least one statement that is new or modified in P' (when P is executed, a statement that does not exist in P cannot be exercised).

Ad hoc/Random techniques When time constraints prohibit the use of a retest-all approach, but no test selection tool is available, developers often select test cases based on 'intuitions' or loose associations of test cases with functionality. Another simple approach is to randomly select a predetermined number of test cases from T.

Retest-all technique The retest-all technique simply reuses all existing test cases. To test P', the technique effectively selects all test cases in T.

Evaluating Regression Test Selection Techniques

To choose a technique for practical application, Rothermel *et al.* [51] have recognized the following categories for evaluating the regression test selection techniques:

Inclusiveness It measures the extent to which M chooses modification-revealing test from T for inclusion in T', where M is a regression test selection technique.

Suppose, T contains n tests that are modification-revealing for P and P', and suppose M selects m of these tests. The inclusiveness of M relative to P and P' and T is:

1. $\text{INCL}(M) = (100 \times (m/n)\%$, if $n \neq 0$
2. $\text{INCL}(M)\% = 100\%$, if $n = 0$

For example, if T contains 100 tests of which 20 are modification-revealing for P and P', and M selects 4 of these 20 tests, then M is 20% inclusive relative to P, P', and T. If T contains no modification-revealing tests, then every test selection technique is 100% inclusive relative to P, P', and T.

If for all P, P', and T, M is 100% inclusive relative to P, P', and T, then M is safe.

Precision It measures the extent to which M omits tests that are non-modification-revealing. Suppose T contains n tests that are non-modification-revealing for P and P', and suppose M omits m of these tests. The precision of M relative to P, P', and T is given by

$$\text{Precision} = 100 \times (m/n)\ \% \text{ if } n \neq 0$$

$$\text{Precision} = 100\ \% \text{ if } n = 0$$

For example, if T contains 50 tests of which 22 are non-modification-revealing for P and P', and M omits 11 of these 22 tests, then M is 50% precise relative to P, P', and T. If T contains no non-modification-revealing tests, then every test selection technique is 100% precise relative to P, P', and T.

Efficiency The efficiency of regression test selection techniques is measured in terms of their space and time requirements. Where time is concerned, a test selection technique is more economical than the retest-all technique, if the cost of selecting T' is less than the cost of running the tests in $T–T'$. Space efficiency primarily depends on the test history and program analysis information a technique must store. Thus, both space and time efficiency depends on the size of the test suite that a technique selects, and on the computational cost of that technique.

8.8.2 Regression Test Prioritization

The regression test prioritization approach is different as compared to selective retest techniques. The prioritization methods order the execution of RTS with the intention that a fault is detected earlier. In other words, regression test prioritization attempts to reorder a regression test suite so that those tests with the highest priority, according to some established criterion, are executed earlier in the regression testing process than those with a lower priority. By prioritizing the execution of a regression test suite, these methods reveal important defects in a software system earlier in the regression testing process.

This approach can be used along with the previous selective retest technique. The steps for this approach are given below:

1. Select T' from T, a set of test to execute on P'.
2. Produce T'_p, a permutation of T', such that T'_p will have a better rate of fault detection than T'.
3. Test P' with T'_p in order to establish the correctness of P' with respect to T'_p.
4. If necessary, create T'', a set of new functional or structural tests for P''.
5. Test P' with T'' in order to establish the correctness of P' with respect to T''.
6. Create T''', a new test suite for P', from T, T'_p, and T''.

Step 1 allows the tester to select T' such that it is a proper subset of T or that it actually contains every test that T contains. Furthermore, step 2 could produce T'_p so that it contains an execution ordering for the regression test that

is the same or different than the order provided by T'. The intention behind prioritizing the execution of a regression test suite is to ensure that the defects normally introduced by competent programmers are discovered earlier in the testing process.

Test prioritization will be discussed in detail in Chapter 12.

SUMMARY

This chapter discusses the importance of regression testing. It validates the changes after the code is modified or the requirements are updated. However, regression testing is considered a problem. The general strategy to validate the system is to execute all the test cases. But it is nearly impossible to test all of them due to time and budget constraints. In this case, we need to decide which parts or modules are unaffected by the changes. So the problem of regression testing lies in deciding the subset of full test cases which will validate the modifications in the software. This chapter discusses the issues of defining regression testing problem, regression testing techniques, and their evaluation.

Let us review the important concepts described in this chapter:

- Regression testing often has a strong positive influence on software quality.
- Regression number is the average number of affected test cases in the test suite that are affected by any modification to a single instruction.
- Regression testing is of two types: bug-fix regression and side-effect regression
- Regression testing is considered a problem, as the existing test suite with probable additional test cases needs to be tested again and again whenever there is a modification.
- Regression testing techniques are: regression test selection technique, test case prioritization technique, and test suite reduction technique.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. A changed version that has not passed a regression test is called _____.
   (a) baseline version
   (b) delta version
   (c) delta build
   (d) none of the above
2. Regression testability is a function of _____.
   (a) design of the program and the test suite
   (b) design of the program
   (c) test suite
   (d) none of the above

3. Regression number is computed using information about the _____.
   (a) design coverage of the program
   (b) test suite coverage of the program
   (c) number of faults
   (d) all of the above
4. Changes made to correct a system after a failure has been observed is called _____.
   (a) perfective maintenance
   (b) adaptive maintenance
   (c) corrective maintenance
   (d) all of the above
5. Changes made to achieve continuing compatibility with the target environment or other systems is called _____.
   (a) perfective maintenance
   (b) adaptive maintenance
   (c) corrective maintenance
   (d) all of the above
6. Changes designed to improve or add capabilities is called _____.
   (a) perfective maintenance
   (b) adaptive maintenance
   (c) corrective maintenance
   (d) all of the above
7. Changes made to increase robustness, maintainability, portability, and other features is called _____.
   (a) perfective maintenance
   (b) preventive maintenance
   (c) adaptive maintenance
   (d) corrective maintenance
8. Which statement best characterizes the regression test selection?
   (a) Minimize the resources required to regression test a new version
   (b) Typically achieved by minimizing the number of test cases applied to the new version
   (c) Select a test case which has caused the problem
   (d) None of the above
9. Problem to select a subset T' of T with which P' will be tested is called _____.
   (a) coverage identification problem
   (b) test suite execution problem
   (c) regression test selection problem
   (d) test suite maintenance problem

10. The problem to identify portions of *P'* or its specification that requires additional testing is called _____.
    (a) coverage identification problem
    (b) test suite execution problem
    (c) regression test selection problem
    (d) test suite maintenance problem
11. The problem to execute test suites efficiently and checking test results for correctness is called _____.
    (a) coverage identification problem
    (b) test suite execution problem
    (c) regression test selection problem
    (d) test suite maintenance problem
12. The problem to update and store test information is called _____.
    (a) coverage identification problem
    (b) test suite execution problem
    (c) regression test selection problem
    (d) test suite maintenance problem
13. A test case t is modification-revealing for *P* and *P'* if _____.
    (a) it executes new or modified code in *P'*
    (b) it detect a fault in *P'* if it causes *P'* to fail
    (c) it causes the outputs of *P* and *P'* to differ
    (d) all of the above
14. A test case *t* is modification-traversing if _____.
    (a) it executes new or modified code in *P'*
    (b) it detect a fault in *P'* if it causes *P'* to fail
    (c) it causes the outputs of *P* and *P'* to differ
    (d) all of the above
15. A test case *t* is fault-revealing if _____.
    (a) it executes new or modified code in *P'*
    (b) it detect a fault in *P'* if it causes *P'* to fail
    (c) it causes the outputs of *P* and *P'* to differ
    (d) all of the above
16. Regression testing is helpful in _____.
    (a) detecting bugs
    (b) detecting undesirable side effects by changing the operating environment
    (c) integration testing
    (d) all of the above

Review Questions

1. When is a system said to regress?
2. What is the difference between baseline version and delta version?
3. How do you classify a program as regression testable?
4. What is regression number?
5. How does regression testing help in producing a quality software?
6. What are the situations to perform regression testing?
7. Why is regression testing a problem?
8. Define regression testing problem with an example.
9. What is the aim of selective retest technique?
10. Define the following:
    (a) Regression test selection problem
    (b) Coverage identification problem
    (c) Test suite execution problem
    (d) Test suite maintenance problem
11. What is regression test prioritization test technique?
12. What is the difference between fault-revealing test cases, modification-revealing test cases, and modification-traversing test cases?
13. Explain the following regression test selection techniques:
    (a) Minimization
    (b) Dataflow
    (c) Safe
    (d) Ad hoc
14. What are the different parameters for evaluating regression test selection techniques?
15. *T* contains 90 tests of which 20 are modification-revealing for *P* and *P'* and *M* selects 12 of these 20 tests, then calculate the inclusiveness of *M* relative to *P*, *P'*, and *T*.
16. *T* contains 100 tests of which 64 are non-modification-revealing for *P* and *P'* and *M* omits 43 of these 64 tests, then calculate the precision of *M* relative to *P*, *P'*, and *T*.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Part 3 Managing the Test Process

CHAPTERS

Chapter 9:
Test Management

Chapter 10:
Software Metrics

Chapter 11:
Testing Metrics for Monitoring and Controlling the Testing Process

Chapter 12:
Efficient Test Suite Management

Testing process, still in its infancy, needs to be managed from various perspectives. In many industries, no test organization hierarchy is followed. Rather, there is a single development team performing every activity including testing. To perform effective testing, we need a testing hierarchy of people with various roles. To manage a testing process, thus, the first step is to get a test organization in place. Next, we must have a master test plan which guides us when and how to perform various testing activities.

In this part, we discuss various test plans including verification and validation test plans. Various specifications such as test design, test case, test result specifications, have been explained in order to carry out systematic documented testing.

To manage the testing process, an organization is not sufficient; we also need to monitor and control various testing activities in order to meet the testing and project milestones. But we cannot monitor and control the test activities unless we have some methods of measurements.

This part discusses software and testing metrics. These metrics help in understanding, monitoring, and controlling various testing activities.

There is one major problem while managing the test process. The test suite prepared to test the system and to perform regression testing is too large to test. We are not able to execute all the test cases. In such a situation, if we leave out some of the test cases, we are unsure if the left-out test cases are critical test cases to be executed or not. This part discusses various techniques adopted to reduce the number of test cases in order to perform effective test cases and manage the test suite.

This part will make ground for the following concepts:

- Structure of test organization
- Test planning
- Test design and report specifications
- Testing metrics
- Test suite reduction methods

Chapter

9

Test Management

Test management is concerned with both test resource and test environment management. It is the role of test management to ensure that new or modified service products meet the business requirements for which they have been developed or enhanced. The key elements of test management are discussed below.

OBJECTIVES

After reading this chapter, you should be able to understand:

- Elements of test management
- Capabilities of a tester
- Structure of testing group
- Test plan components
- Master test plan
- Verification test plan
- Unit test plan
- Integration test plan
- Function test plan
- System test plan
- Acceptance test plan
- Test design specifications
- Test case specifications
- Test result specifications: test log, test incident report, and test summary report

Test organization It is the process of setting up and managing a suitable test organizational structure and defining explicit roles. The project framework under which the testing activities will be carried out is reviewed, high-level test phase plans are prepared, and resource schedules are considered. Test organization also involves the determination of configuration standards and defining the test environment.

Test planning The requirements definition and design specifications facilitate the identification of major test items and these may necessitate updating the test strategy. A detailed test plan and schedule is prepared with key test responsibilities being indicated.

Detailed test design and test specifications A detailed design is the process of designing a meaningful and useful structure for the tests as a whole. It specifies the details of the test approach for a software functionality or feature and identifying the associated test cases.

Test monitoring and assessment It is the ongoing monitoring and assessment to check the integrity of development and construction. The status of configuration items should be reviewed against the phase plans and the test progress reports prepared, the ensure the verification and validation activities are correct.

Product quality assurance The decision to negotiate the acceptance testing program and the release and commissioning of the service product is subject to the 'product assurance' role being satisfied with the outcome of the verification and validation activities. Product assurance may oversee some of the test activity and may participate in process reviews.

9.1 Test Organization

Since testing is viewed as a process, it must have an organization such that a testing group works for better testing and high quality software. The testing group is responsible for the following activities:

- Maintenance and application of test policies
- Development and application of testing standards
- Participation in requirement, design, and code reviews
- Test planning
- Test execution
- Test measurement
- Test monitoring
- Defect tracking
- Acquisition of testing tools
- Test reporting

The staff members of such a testing group are called *test specialists* or *test engineers* or simply *testers*. A tester is not a developer or an analyst. He does not debug the code or repair it. He is responsible for ensuring that testing is effective and quality issues are being addressed. The skills a tester should possess are discussed below.

Personal and Managerial Skills

- Testers must be able to contribute in policy-making and planning the testing activities.
- Testers must be able to work in a team.
- Testers must be able to organize and monitor information, tasks, and people.
- Testers must be able to interact with other engineering professionals, software quality assurance staff, and clients.
- Testers should be capable of training and mentoring new testers.

- Testers should be creative, imaginative, and experiment-oriented.
- Testers must have written and oral communication skills.

Technical Skills

- Testers must be technically sound, capable of understanding software engineering principles and practices.
- Testers must be good in programming skills.
- Testers must have an understanding of testing basics, principles, and practices.
- Testers must have a good understanding and practice of testing strategies and methods to develop test cases.
- Testers must have the ability to plan, design, and execute test cases with the goal of logic coverage.
- Testers must have technical knowledge of networks, databases, operating systems, etc. needed to work in a the project environment.
- Testers must have the knowledge of configuration management.
- Testers must have the knowledge of testware and the role of each document in the testing process.
- Testers must have know about quality issues and standards.

9.2 Structure of Testing Group

Testing is an important part of any software project. One or two testers are not sufficient to perform testing, especially if the project is too complex and large. Therefore, many testers are required at various levels. Figure 9.1 shows different types of testers in a hierarchy.

Test Manager A test manager occupies the top level in the hierarchy. He has the following responsibilities:

(i) He is the key person in the testing group who will interact with project management, quality assurance, and marketing staff.

(ii) Takes responsibility for making test strategies with detailed master planning and schedule.

(iii) Interacts with customers regarding quality issues.

(iv) Acquires all the testing resources including tools.

(v) Monitors the progress of testing and controls the events.

(vi) Participates in all static verification meetings.

(vii) Hires, fires, and evaluates the test team members.

Figure 9.1 Testing Group Hierarchy

Test Leader The next tester in the hierarchy is the test leader who assists the test manager in meeting testing and quality goals. The prime responsibility of a test leader is to lead a team of test engineers who are working at the leaf-level of the hierarchy. The following are his responsibilities:

(i) Planning the testing tasks given by the test manager.
(ii) Assigning testing tasks to test engineers who are working under him.
(iii) Supervising test engineers.
(iv) Helping the test engineers in test case design, execution, and reporting.
(v) Providing tool training, if required.
(vi) Interacting with customers.

Test Engineers Test engineers are highly experienced testers. They work under the leadership of the test leader. They are responsible for the following tasks:

(i) Designing test cases.
(ii) Developing test harness.
(iii) Set-up test laboratories and environment.
(iv) Maintain the test and defect repositories.

Junior Test Engineers Junior test engineers are newly hired testers. They usually are trained about the test strategy, test process, and testing tools. They participate in test design and execution with experienced test engineers.

9.3 Test Planning

There is a general human tendency to 'get on with the next thing', especially under pressure [1]. This is true for the testers who are always short of time.

However, if resources are to be utilized intelligently and efficiently during the earlier testing phases and later phases, these are repaid many times over. The time spent on planning the testing activities early is never wasted and usually the total time cycle is significantly shorter.

According to the test process as discussed in STLC, testing also needs planning as is needed in SDLC. Since software projects become uncontrolled if not planned properly, the testing process is also not effective if not planned earlier. Moreover, if testing is not effective in a software project, it also affects the final software product. Therefore, for a quality software, testing activities must be planned as soon as the project planning starts.

A *test plan* is defined as a document that describes the scope, approach, resources, and schedule of intended testing activities. Test plan is driven with the business goals of the product. In order to meet a set of goals, the test plan identifies the following:

- Test items
- Features to be tested
- Testing tasks
- Tools selection
- Time and effort estimate
- Who will do each task
- Any risks
- Milestones

9.3.1 Test Plan Components

IEEE Std 829–1983 [56] has described the test plan components. These components (see Fig. 9.2) must appear for every test item. These components are described below.

Test Plan Identifier

Each test plan is tagged with a unique identifier so that it is associated with a project.

Introduction

The test planner gives an overall description of the project with:

- Summary of the items and features to be tested.
- The requirement and the history of each item (optional).
- High-level description of testing goals.

- References to related documents, such as project authorization, project plan, QA plan, configuration management plan, relevant policies, and relevant standards.

Test Plan Components

- Test Plan Identifier
- Introduction
- Test-Item to be tested
- Features to be tested
- Features not to be tested
- Approach
- Item Pass/Fail Criteria
- Suspension criteria and resumption requirements
- Test deliverables
- Testing tasks
- Environmental needs
- Responsibilities
- Staffing and training needs
- Scheduling
- Risks and contingencies
- Testing costs
- Approvals

Figure 9.2 Test plan components

Test-Item to be Tested

- Name, identifier, and version of test items.
- Characteristics of their transmitting media where the items are stored, for example, disk, CD, etc.
- References to related documents, such as requirements specification, design specification, users' guide, operations guide, installation guide.
- References to bug reports related to test items.
- Items which are specifically not going to be tested (optional).

Features to be Tested

This is a list of what needs to be tested from the user's viewpoint. The features may be interpreted in terms of functional and quality requirements.

- All software features and combinations of features are to be tested.
- References to test-design specifications associated with each feature and combination of features.

Features Not to be Tested

This is a list of what should 'not' be tested from both the user's viewpoint and the configuration management/version control view:

- All the features and the significant combinations of features which will not be tested.
- Identify *why* the feature is not to be tested. There can be many reasons:
  - (i) Not to be included in this release of the software.
  - (ii) Low-risk, has been used before, and was considered stable.
  - (iii) Will be released but not tested or documented as a functional part of the release of this version of the software.

Approach

- Overall approach to testing.
- For each major group of features or combinations of features, specify the approach.
- Specify major activities, techniques, and tools which are to be used to test the groups.
- Specify the metrics to be collected.
- Specify the number of configurations to be tested.
- Specify a minimum degree of comprehensiveness required.
- Identify which techniques will be used to judge comprehensiveness.
- Specify any additional completion criteria.
- Specify techniques which are to be used to trace requirements.
- Identify significant constraints on testing, such as test-item availability, testing-resource availability, and deadline.

Item Pass/Fail Criteria

This component defines a set of criteria based on which a test case is passed or failed. The failure criteria is based on the severity levels of the defect. Thus, an acceptable severity level for the failures revealed by each test case is specified and used by the tester. If the severity level is beyond an acceptable limit, the software fails.

Suspension Criteria and Resumption Requirements

Suspension criteria specify the criteria to be used to suspend all or a portion of the testing activities, while resumption criteria specify when the testing can resume after it has been suspended.

For example, system integration testing in the integration environment can be suspended in the following circumstances:

- Unavailability of external dependent systems during execution.
- When a tester submits a 'critical' or 'major' defect, the testing team will call for a break in testing while an impact assessment is done.

System integration testing in the integration environment may be resumed under the following circumstances:

- When the 'critical' or 'major' defect is resolved.
- When a fix is successfully implemented and the testing team is notified to continue testing.

Test Deliverables

- Identify deliverable documents: test plan, test design specifications, test case specifications, test item transmittal reports, test logs, test incident reports, test summary reports, and test harness (stubs and drivers).
- Identify test input and output data.

Testing Tasks

- Identify the tasks necessary to prepare for and perform testing.
- Identify all the task interdependencies.
- Identify any special skills required.

All testing-related tasks and their interdependencies can be shown through a *work breakdown structure* (WBS). WBS is a hierarchical or tree-like representation of all testing tasks that need to be completed in a project.

Environmental Needs

- Specify necessary and desired properties of the test environment: physical characteristics of the facilities including hardware, communications and system software, the mode of usage (i.e. stand-alone), and any other software or supplies needed.
- Specify the level of security required.
- Identify any special test tools needed.
- Identify any other testing needs.
- Identify the source for all needs which are not currently available.

Responsibilities

- Identify the groups responsible for managing, designing, preparing, executing, checking, and resolving.

- Identify the groups responsible for providing the test items identified in the test items section.
- Identify the groups responsible for providing the environmental needs identified in the environmental needs section.

Staffing and Training Needs

- Specify staffing needs by skill level.
- Identify training options for providing necessary skills.

Scheduling

- Specify test milestones.
- Specify all item transmittal events.
- Estimate the time required to perform each testing task.
- Schedule all testing tasks and test milestones.
- For each testing resource, specify a period of use.

Risks and Contingencies

Specify the following overall risks to the project with an emphasis on the testing process:

- Lack of personnel resources when testing is to begin.
- Lack of availability of required hardware, software, data, or tools.
- Late delivery of the software, hardware, or tools.
- Delays in training on the application and/or tools.
- Changes to the original requirements or designs.
- Complexities involved in testing the applications.

Specify the actions to be taken for various events. An example is given below.

Requirements definition will be complete by January 1, 20XX and, if the requirements change after that date, the following actions will be taken:

The test schedule and the development schedule will move out an appropriate number of days. This rarely occurs, as most projects tend to have fixed delivery dates.

- The number of tests performed will be reduced.
- The number of acceptable defects will increase.
- These two items may lower the overall quality of the delivered product.
- Resources will be added to the test team.
- The test team will work overtime.

- The scope of the plan may be changed.
- There may be some optimization of resources. This should be avoided, if possible, for obvious reasons.

The management team is usually reluctant to accept scenarios such as the one mentioned above even though they have seen it happen in the past. The important thing to remember is that, if you do nothing at all, testing is cut back or omitted completely, neither of which should be an acceptable option.

Testing Costs

The IEEE standard has not included this component in its specification. However, it is a usual component of any test plan, as test costs are allocated in the total project plan. To estimate the costs, testers will need tools and techniques. The following is a list of costs to be included:

- Cost of planning and designing the tests
- Cost of acquiring the hardware and software required for the tests
- Cost to support the environment
- Cost of executing the tests
- Cost of recording and analysing the test results
- Cost of training the testers, if any
- Cost of maintaining the test database

Approvals

- Specify the names and titles of all the people who must approve the plan.
- Provide space for signatures and dates.

9.3.2 Test Plan Hierarchy

Test plans can be organized in several ways depending on the organizational policy. There is often a hierarchy of plans that includes several levels of quality assurance and test plans. At the top of the plan hierarchy is a master plan which gives an overview of all verification and validation activities for the project, as well as details related to other quality issues such as audits, standards, and configuration control. Below the master test plan, there is individual planning for each activity in verification and validation, as shown in Fig. 9.3.

The test plan at each level of testing must account for information that is specific to that level, e.g. risks and assumptions, resource requirements, schedule, testing strategy, and required skills. The test plans according to each level are written in an order such that the plan prepared first is executed last,

i.e. the acceptance test plan is the first plan to be formalized but is executed last. The reason for its early preparation is that the things required for its completion are available first.

Figure 9.3 Test Plan Hierarchy

9.3.3 Master Test Plan

The master test plan provides the highest level description of verification and validation efforts and drives the testing at various levels. General project information is used to develop the master test plan.

The following topics must be addressed before planning:

- Project identification
- Plan goals
- Summary of verification and validation efforts
- Responsibilities conveyed with the plan
- Software to be verified and validated
- Identification of changes to organization standards

Verification and validation planning may be broken down into the following steps:

- Identify the V&V scope
- Establish specific objectives from the general project scope
- Analyse the project input prior to selecting V&V tools and techniques
- Select tools and techniques
- Develop the software verification and validation plan (SVVP).

Discussed below are the major activities of V&V planning.

Master Schedule

- The master schedule summarizes various V&V tasks and their relationship to the overall project.
- Describes the project life cycle and project milestones including completion dates.
- Summarizes the schedule of V&V tasks and how verification and validation results provide feedback to the development process to support overall project management functions.
- Defines an orderly flow of material between project activities and V&V tasks. Use reference to PERT, CPM, and Gantt charts to define the relationship between various activities.

Resource Summary

This activity summarizes the resources needed to perform V&V tasks, including staffing, facilities, tools, finances, and special procedural requirements such as security, access rights, and documentation control. In this activity,

- Use graphs and tables to present resource utilization.
- Include equipment and laboratory resources required.
- Summarize the purpose and cost of hardware and software tools to be employed.
- Take all resources into account and allow for additional time and money to cope with contingencies.

Responsibilities

Identify the organization responsible for performing V&V tasks. There are two levels of responsibilities—general responsibilities assigned to different organizations and specific responsibilities for the V&V tasks to be performed, assigned to individuals. General responsibilities should be supplemented with specific responsibility for each task in the V&V plan.

Tools, Techniques, and Methodology

Identify the special software tools, techniques, and methodologies to be employed by the V&V team. The purpose of each should be defined and plans for the acquisition, training, support, and qualification of each should be described. This section may be in a narrative or graphic format. A separate tool plan may be developed for software tool acquisition, development, or modification. In this case, a separate tool plan section may be added to the plan.

9.3.4 Verification Test Plan

Verification test planning includes the following tasks:

- The item on which verification is to be performed.
- The method to be used for verification: review, inspection, walkthrough.
- The specific areas of the work product that will be verified.
- The specific areas of the work product that will not be verified.
- Risks associated.
- Prioritizing the areas of work product to be verified.
- Resources, schedule, facilities, tools, and responsibilities.

9.3.5 Validation Test Plan

Validation test planning includes the following tasks:

- Testing techniques
- Testing tools
- Support software and documents
- Configuration management
- Risks associated, such as budget, resources, schedule, and training

Unit Test Plan

Generally, unit tests are performed by the module developer informally. In this case, bugs are not recorded and do not become a part of the unit's history. It is a bad practice because these bugs' history will not be re-used, which otherwise becomes important especially in real-time systems. Therefore, to implement unit testing formally and to make it an effective testing for other projects, it is necessary to plan for unit test. To prepare for unit test, the developer must do the following tasks:

- Prepare the unit test plan in a document form.
- Design the test cases.
- Define the relationship between tests.
- Prepare the test harness, i.e. stubs and drivers, for unit testing.

Discussed below are the tasks contained in a test plan document.

Module overview Briefly define the purpose of this module. This may require only a single phrase, i.e. calculates overtime pay amount, calculates equipment depreciation, performs date edit validation, or determines sick pay eligibility, etc.

Inputs to module Provide a brief description of the inputs to the module under test.

Outputs from module Provide a brief description of the outputs from the module under test.

Logic flow diagram Provide a logic flow diagram, if additional clarity is required. For example, DFD, STD, and ERD.

Test approach Decide the method to test the module.

Features to be tested Identify the features to be tested for the module.

Features not to be tested Identify the features not be tested due to some constraints. Also, assess the risks of not testing these features.

Test constraints Indicate anticipated limitations on the test due to test conditions, such as interfaces, equipment, personnel, and databases.

Test harness Describe the requirements for test harness that interfaces with the units to be tested.

Interface modules Identify the modules that interface with this module indicating the nature of the interface: outputs data to, receives input data from, internal

program interface, external program interface, etc. Identify the sequencing required for subsequent string tests or sub-component integration tests.

Test tools Identify any tools employed to conduct unit testing. Specify any stubs or utility programs developed or used to invoke tests. Identify names and locations of these aids for future regression testing. If data is supplied from the unit test of a coupled module, specify module relationship.

Archive plan Specify how and where the data is archived for use in subsequent unit tests. Define any procedures required to obtain access to data or tools used in the testing effort. The unit test plans are normally archived with the corresponding module specifications.

Updates Define how updates to the plan will be identified. Updates may be required due to enhancements, requirement changes, etc. The same unit test plan should be re-used with revised or appended test cases identified in the update section.

Milestones List the milestone events and dates for unit testing.

Budget List the funds allocated to test this module. A checklist can also be maintained, as shown below:

- Does the test plan completely test the functionality of the unit as defined in the design specification and in functional specification?
- Are the variable (including global variables) inputs to the unit included in the features to be tested in the test plan?
- Are the variable outputs from the unit result included in the features to be tested?
- Is every procedure in the design exercised by the test?
- Does the test plan include critical procedures to demonstrate that performance meets stated requirements?
- For units that interface with other units, are the invalid features tested to test for defensive coding?

Integration Test Plan

An integration test plan entails the development of a document, which instructs a tester what tests to perform in order to integrate and test the already existing individual code modules. The objective of the integration test plan activity is to develop a plan which specifies the necessary steps needed to integrate individual modules and test the integration. It also helps the technical team to think through the logical sequence of integration activities, so

that their individual detailed development plans are well-synchronized, and integration happens in a reasonable manner.

Create a draft of the plan very early in the execution phase. By this time, you will know from the high-level design work in the initiation phase, what major pieces of hardware and/or software will have to be integrated. An early look at integration sequencing will help you sanity-check the amount of time you have put into the schedule for integration testing.

Discussed below are the tasks to be performed under integration test plan.

Documents to be used All the documents needed should be arranged, e.g. SRS, SDD, user manual, usage scenarios containing DFDs, data dictionaries, state charts, etc.

Modules for integration It should be planned which modules will be integrated and tested. There can be many criteria for choosing the modules for integrating. Some are discussed below.

- **Availability** All the modules may not be ready at one time. But we cannot delay the integration and testing of modules which are ready. Therefore, the modules which are ready for integration must be integrated and tested first.
- **Subsystems** The goal of integration testing is also to have some working subsystems and combine them into the working system as a whole. While planning for integration, the subsystems are selected based on the requirements, user needs, and availability of the modules. The subsystems may also be prioritized based on key or critical features. Sometimes, developers need to show the clients some subsystems that are working at the time of integration testing.

Strategy for integration The strategy selection for integration testing should also be well-planned. Generally, sandwich integration testing is adopted.

Test harness A factor to be considered while developing an integration test plan is the amount of 'dead code' or test harness that will have to be developed. The *drivers* and *stub* code that is developed is thrown away and never used again, so a minimum amount of development of this code is desired. Thus, if the program uses a lot of low-level routines such that it is not practical to write a driver for every low-level routine, then perhaps the level above should be developed first, with drivers for it.

List of tasks to be tested The functionalities and interfaces to be tested through the integration of modules must also be planned.

List of tasks not to be tested Plan the functionalities and interfaces not to be tested due to whatever reasons.

Function Test Plan

During planning, the test lead with assistance from the test team defines the scope, schedule, and deliverables for the function test cycle. The test lead delivers a test plan (document) and a test schedule (work plan)—these often undergo several revisions during the testing cycle. The following are the tasks under function test planning:

- **List the objectives of function testing** A list of the overall objectives for performing function testing is prepared.
- **Partitioning/Functional decomposition** Functional decomposition of a system (or partitioning) is the breakdown of a system into its functional components or functional areas. Another group in the organization may take responsibility for the functional decomposition (or model) of the system, but the testing organization should still review this deliverable for completeness before accepting it into the test organization. If the functional decomposition or partitions have not been defined or are deemed insufficient, then the testing organization will have to take responsibility for creating and maintaining the partitions.
- **Traceability matrix formation** Test cases need to be traced/mapped back to the appropriate requirement. A function coverage matrix is prepared. This matrix is a table, listing specific functions to be tested, the priority for testing each function, and test cases that contain tests for each function. Once all the aspects of a function have been tested by one or more test cases, the test design activity for that function can be considered complete. This approach gives a more accurate picture of the application when coverage analysis is done.
- **List the functions to be tested** A list of all the functions mentioned in the traceability matrix with their appropriate details needed for planning, is prepared.

System Test Plan

One of the most important parts of software development is testing. Before you implement a new software into a system, you must be sure it won't crash and that proper bug checking has been done. Testing ensures that your system runs properly and efficiently and meets all the requirements. To fully ensure that your system is free of defects, testing should follow a system test plan and must be thorough and complete.

Often people rush to implement a new program and then spend hours repairing the system, losing many hours of productivity. The experts prevent this by using a system test plan. Basically, a system test plan is a systematic approach to testing a system. The plan contains a detailed understanding of what the eventual workflow will be. This will help keep the project on schedule and make the integration of the new system as smooth as possible. We should not build a house without blueprints, nor should we run a new system without having created a proper system test plan.

System testing is also difficult, as there are no design methodologies for test cases because requirements and objectives do not describe the functions precisely. However, requirements are general. Therefore, a system test plan first requires that requirements are formulated specifically. For this system, test cases are divided into some categories, according to which the system test plan is described. For example, system testing is described in terms of performance testing, then the test plans are prepared according to performance checking. Similarly, test plans can be prepared for stress testing, compatibility testing, security testing, etc.

The following steps describe how a system test plan is implemented.

Partition the requirements Partition the requirements into logical categories and for each category, develop a list of detailed requirements and plan.

System description Provide a chart and briefly describe the inputs, outputs, and functions of the software being tested as a frame of reference for the test descriptions.

Features to be tested The set of functionalities of the test items to be tested must be recognized, as there may be several criteria for system testing.

Strategy and reporting format The strategy to be followed in system testing should also be planned. The type of tests to be performed, standards for documentation, mechanism for reporting, and special considerations for the type of project must also be planned, e.g. a test strategy might say that black-box testing techniques will be used for functional testing.

Develop a requirement coverage matrix Develop a requirement coverage matrix which is a table in which an entry describes a specific subtest, priority of the subtest, the specific test cases in which the subtest appears. This matrix specifies the functions that are to be tested, defining the objectives of the test.

Smoke test set The system test plan also includes a group of test cases that establish that the system is stable and all major functionalities are working. This group of test cases is referred to as *smoke tests* or *testability assessment criteria.*

Entry/Exit criteria The entry criteria will contain some exit criteria from the previous level as well as establishment of test environment, installation of tools, gathering of data, and recruitment of testers, if necessary. The exit criteria also need to be established, e.g. all test cases have been executed and no major identified bugs are open.

Suspension criteria It identifies the conditions that suspend testing, e.g. critical bugs preventing further testing.

Resource requirements State the resource requirements including documents, equipment, software, and personnel.

Test team State the members who are in the test team and enlist their assignments.

Participating organizations System testing may not be done by a single organization, as the software to be developed may be large with many modules being developed by various organizations. Thus, integrated system testing needs people from all these organizations. Identify the participating organizations and fix the date and time with them.

Extent and constraints Since the system testing of a large software system needs many resources, some might not be available. Therefore, indicate the extent of testing, e.g. total or partial. Include the rationale for the partial one. Similarly, indicate anticipated limitations on the test due to test conditions, such as interfaces, equipment, personnel, and database.

Schedule estimation The time schedule for accomplishing testing milestones should also be planned. But this plan should be in tune with the time allocation in the project plan for testing.

Acceptance Test Plan

It is important to have the acceptance criteria defined so that acceptance testing is performed against those criteria. It means that in the acceptance test plan, we must have all the acceptance criteria defined in one document. If these are not available, then prepare them and plan the acceptance test accordingly. Acceptance criteria are broadly defined for functionality requirements, performance requirements, interface quality requirements, and overall software quality requirements.

Another point in acceptance testing plan is to decide the criticality of acceptance features defined. It is necessary to define the criticality. If the system fails the high critical acceptance requirement, then it should not pass the acceptance testing.

9.4 Detailed Test Design and Test specifications

The ultimate goal of test management is to get the test cases executed. Till now, test planning has not provided the test cases to be executed. Detailed test designing for each validation activity maps the requirements or features to the actual test cases to be executed. One way to map the features to their test cases is to analyse the following:

- Requirement traceability
- Design traceability
- Code traceability

The analyses can be maintained in the form of a traceability matrix (see Table 9.1) such that every requirement or feature is mapped to a function in the functional design. This function is then mapped to a module (internal design and code) in which the function is being implemented. This in turn is linked to the test case to be executed. This matrix helps in ensuring that all requirements have been covered and tested. Priority can also be set in this table to prioritize the test cases.

Table 9.1 Traceability matrix

Requirement/Feature	Functional Design	Internal Design/Code	Test cases
R1	F1, F4,F5	abc.cpp, abc.h	T5, T8,T12,T14

9.4.1 Test Design Specification

A test design specification should have the following components according to IEEE recommendation [56]:

Identifier A unique identifier is assigned to each test design specification with a reference to its associated test plan.

Features to be tested The features or requirements to be tested are listed with reference to the items mentioned in SRS/SDD.

Approach refinements In the test plan, an approach to overall testing was discussed. Here, further details are added to the approach. For example, special testing techniques to be used for generating test cases are explained.

Test case identification The test cases to be executed for a particular feature/function are identified here, as shown in Table 9.1. Moreover, test procedures are also identified. The test cases contain input/output information and the test procedures contain the necessary steps to execute the tests. Each test design specification is associated with test cases and test procedures. A test case may be associated with more than one test design specifications.

Feature pass/fail criteria The criteria for determining whether the test for a feature has passed or failed, is described.

9.4.2 Test Case Specifications

Since the test design specifications have recognized the test cases to be executed, there is a need to define the test cases with complete specifications. The test case specification document provides the actual values for input with expected outputs. One test case can be used for many design specifications and may be re-used in other situations. A test case specification should have the following components according to IEEE recommendation [56]:

Test case specification identifier A unique identifier is assigned to each test case specification with a reference to its associated test plan.

Purpose The purpose of designing and executing the test case should be mentioned here. It refers to the functionality you want to check with this test case.

Test items needed List the references to related documents that describe the items and features, e.g. SRS, SDD, and user manual.

Special environmental needs In this component, any special requirement in the form of hardware or software is recognized. Any requirement of tool may also be specified.

Special procedural requirements Describe any special condition or constraint to run the test case, if any.

Inter-case dependencies There may be a situation that some test cases are dependent on each other. Therefore, previous test cases which should be run prior to the current test case must be specified.

Input specifications This component specifies the actual inputs to be given while executing a test case. The important thing while specifying the input values is not to generalize the values, rather specific values should be provided. For example, if the input is in angle, then the angle should not be specified as a range between 0 and 360, but a specific value like 233 should be specified. If there is any relationship between two or more input values, it should also be specified.

Test procedure The step-wise procedure for executing the test case is described here.

Output specifications Whether a test case is successful or not is decided after comparing the output specifications with the actual outputs achieved. Therefore, the output should be mentioned complete in all respects. As in the case of input specifications, output specifications should also be provided in specific values.

Example 9.1

There is a system for railway reservation system. There are many functionalities in the system, as given below:

S. No.	Functionality	Function ID in SRS	Test cases
1	Login the system	F3.4	T1
2	View reservation status	F3.5	T2
3	View train schedule	F3.6	T3
4	Reserve seat	F3.7	T4
5	Cancel seat	F3.8	T5
6	Exit the system	F3.9	T6

Suppose we want to check the functionality corresponding to 'view reservation status'. Its test specification is given in Fig. 9.4.

- **Test case Specification Identifier**
  T2
- **Purpose**
  To check the functionality of 'View Reservation Status'
- **Test Items Needed**
  Refer function F3.5 in SRS of the system.
- **Special Environmental Requirements**
  Internet should be in working condition. Database software through which the data will be retrieved should be in running condition.
- **Special Procedural Requirements**
  The function 'Login' must be executed prior to the current test case.
- **Inter-case Dependencies**
  T1 test case must be executed prior to the current test case execution.
- **Input Specifications**
  Enter PNR number in 10 digits between 0–9 as given below:
  4102645876
  21A2345672
  234
  asdgggggggg
- **Test Procedure**
  Press 'View Reservation status' button.
  Enter PNR number and press ENTER.
- **Output Specifications**
  The reservation status against the entered PNR number is displayed as S12 or RAC13 or WL123 as applicable.

Figure 9.4 Test specifications

9.4.3 Test Procedure Specifications

A test procedure is a sequence of steps required to carry out a test case or a specific task. This can be a separate document or merged with a test case specification.

9.4.4 Test Result Specifications

There is a need to record and report the testing events during or after the test execution, as shown in Fig. 9.5.

Figure 9.5 Test result specifications

Test Log

Test log is a record of the testing events that take place during the test. Test log is helpful for bug repair or regression testing. The developer gets valuable clues from this test log, as it provides snapshots of failures. The format for preparing a test log according to IEEE [56] is given below:

- **Test log identifier**
- **Description** Mention the items to be tested, their version number, and the environment in which testing is being performed.
- **Activity and event entries** Mention the following:
  - (i) Date
  - (ii) Author of the test
  - (iii) Test case ID
  - (iv) Name the personnel involved in testing

(v) For each execution, record the results and mention pass/fail status

(vi) Report any anomalous unexpected event, before or after the execution

The test log corresponding to Example 9.1 is given in Fig. 9.6.

- **Test Log Identifier**
  TL2
- **Description**
  Function 3.5 in SRS v 2.1. The function tested in Online environment with Internet.
- **Activity and Event Entries**
  Mention the following:
  (i) Date: 22/03/2009
  (ii) Author of test: XYZ
  (iii) Test case ID: T2
  (iv) Name of the personnel involved in testing: ABC, XYZ
  (v) For each execution, record the results and mention pass/fail status
  The function was tested with the following inputs:

Inputs	Results	Status
4102645876	S12	Pass
21A2345672	S14	Fail
234	Enter correct 10 digit PNR number	Pass
asdgggggggg	RAC12	Fail

  (vi) Report any anomalous unexpected event before or after the execution.
  Nil

Figure 9.6 Sample test log for Example 9.1

Test Incident Report

This is a form of bug report. It is the report about any unexpected event during testing which needs further investigation to resolve the bug. Therefore, this report completely describes the execution of the event. It not only reports the problem that has occurred but also compares the expected output with the actual results. Listed below are the components of a test incident report [56]:

- **Test incident report identifier**
- **Summary** Identify the test items involved, test cases/procedures, and the test log associated with this test.
- **Incident description** It describes the following:
  (i) Date and time
  (ii) Testing personnel names

(iii) Environment
(iv) Testing inputs
(v) Expected outputs
(vi) Actual outputs
(vii) Anomalies detected during the test
(viii) Attempts to repeat the same test

- **Impact** The originator of this report will assign a severity value/rank to this incident so that the developer may know the impact of this problem and debug the critical problem first.

The test log corresponding to Example 9.1 is given in Fig. 9.7.

- **Test Incident Report Identifier**
  TI2
- **Summary**
  Function 3.5 in SRS v 2.1. Test Case T2 and Test Log TL2.
- **Incident Description**
  It describes the following:
  (i) Date and time: 23/03/2009, 2.00pm
  (ii) Testing personnel names: ABC, XYZ
  (iii) Environment: Online environment with X database
  (iv) Testing inputs
  (v) Expected outputs
  (vi) Actual outputs
  (vii) Anomalies detected during the test
  (viii) Attempts to repeat the same test

Testing Inputs	Expected outputs	Actual outputs	Anomalies detected	Attempts to repeat the same test
4102645876	S12	S12	nil	–
21A2345672	Enter correct 10 digit PNR number	S14	Alphabet is being accepted in the input.	3
234	Enter correct 10 digit PNR number	Enter correct 10 digit PNR number	nil	–
asdgggggggg	Enter correct 10 digit PNR number	RAC12	Alphabet is being accepted in the input.	5

- **Impact**
  The severity value is 1(Highest).

Figure 9.7 Sample test incident report for Example 9.1

Test Summary Report

It is basically an evaluation report prepared when the testing is over. It is the summary of all the tests executed for a specific test design specification. It can provide the measurement of how much testing efforts have been applied for the test. It also becomes a historical database for future projects, as it provides information about the particular type of bugs observed.

The test summary report corresponding to Example 9.1 is given in Fig. 9.8.

- **Test Summary Report Identifier**
  TS2
- **Description**
  SRS v 2.1

S. No.	Functionality	Function ID in SRS	Test cases
1	Login the system	F3.4	T1
2	View reservation status	F3.5	T2
3	View train schedule	F3.6	T3
4	Reserve seat	F3.7	T4
5	Cancel seat	F3.8	T5
6	Exit the system	F3.9	T6

- **Variances**
  Nil
- **Comprehensive Statement**
  All the test cases were tested except F3.7, F3.8, F3.9 according to the test plan.
- **Summary of Results**

S. No.	Functionality	Function ID in SRS	Test cases	Status
1	Login the system	F3.4	T1	Pass
2	View reservation status	F3.5	T2	Bug Found. Could not be resolved.
3	View train schedule	F3.6	T3	Pass

- **Evaluation**
  The functions F3.4, F3.6 have been tested successfully. Function F3.5 couldn't be tested as the bug has been found. The bug is that the PNR number entry has also accepted alphabetical entries as wrong

Figure 9.8 Test summary report

A test summary report contains the following components [56]:

- **Test summary report identifier**
- **Description** Identify the test items being reported in this report with the test case/procedure ID.
- **Variances** Mention any deviation from the test plans, test procedures, if any.

- **Comprehensive statement** The originator of this report compares the comprehensiveness of the testing efforts made with the test plans. It describes what has been tested according to the plan and what has not been covered.
- **Summary of results** All the results are mentioned here with the resolved incidents and their solutions. Unresolved incidents are also reported.
- **Evaluation** Mention the status of the test results. If the status is fail, then mention its impact and severity level.
- **Summary of activities** All testing execution activities and events are mentioned with resource consumption, actual task durations, etc.
- **Approvals** List the names of the persons who approve this document with their signatures and dates.

SUMMARY

Testing is not an intuitive process. It is a systematic well-defined process. Therefore, it needs complete management. There should be a hierarchy of testing persons in the test organization with well-defined roles. Testing activities start with proper planning and continue with detailed test design specifications to result specifications. The idea is to plan and document the steps of STLC according to which the tester works. The tester plans, design, executes the test cases, and reports the test results.

This chapter discusses the test organization with the hierarchy of every testing person. A general test plan's components have been described. A master plan including verification and validation plan is also needed for testing a software. The verification plan and validation test plan at every stage – unit test plan, integration test plan, and system test plan, have also been discussed in this chapter.

The test case specifications along with test design specifications are discussed for designing the test cases. After the test case execution, the test results should also be reported. Test reporting exists in three main documents namely – test log, test incident report, and test summary report. All these testware have been explained in detail with their specifications.

Let us review the important concepts described in this chapter:

- Project manager is a key person in the testing group who interacts with project management, quality assurance, and marketing staff.
- Test leader leads a team of test engineers who work at the leaf-level of the hierarchy.
- Test engineers work under the lead of the test leader and are responsible for designing, developing, and maintaining test cases.
- Junior test engineers are newly hired testers. They usually go for training to learn the test strategy, test process, and testing tools.
- A test plan is defined as a document that describes the scope, approach, resources, and schedule of intended testing activities.

- Master test plan provides the highest level description of verification and validation efforts and drives the testing at various levels.
- Unit test is provided by the module developer. He prepares a test harness to identify the interfaces between the unit to be tested and other units.
- Integration test plan specifies the necessary steps needed to integrate individual modules and test the integration. It helps the technical team to think through the logical sequence of integration activities.
- Function test plan specifies the requirements for a bare-minimum functioning of the system. The plan must be ready with a traceability matrix that maps every function to its requirement and a list of functions to be tested.
- A system test plan is a systematic approach for testing a system containing a detailed understanding of what the eventual workflow will be. For this system, test cases are divided into some categories (recovery, security, performance, compatibility, etc.), according to which the system test plan is described.
- Acceptance test plan must have all the acceptance criteria defined in one document. If they are not available, then prepare them and plan the acceptance test accordingly. Another point in acceptance testing plan is to decide the criticality of acceptance feature defined. It is necessary to define the criticality so as to ensure that the system does not pass the accepted test, if it has failed in high critical acceptance requirement.
- Test log is a record of the testing events that take place during a test.
- Test incident report keeps a log of any unexpected event that has occurred during the test, which needs further investigation to resolve the bug.
- Test summary report is about the tests executed for a specific test design specification. It can measure how much testing efforts have been applied for the test.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. The key elements of test management are _____.
   (a) test organization
   (b) test harness
   (c) test planning
   (d) test monitoring and assessment
2. _____ is at the top-level in the test group hierarchy.
   (a) test engineer
   (b) test manager
   (c) test leader
   (d) junior engineer
3. Verification plan includes _____.
   (a) unit test plan

(b) integration test plan
(c) function design plan
(d) none of the above

4. Validation plan includes _____.
   (a) unit test plan
   (b) integration test plan
   (c) function design plan
   (d) none of the above
5. Master schedule in V&V planning _____.
   (a) summarizes the V&V tasks and their relationship with the overall project
   (b) summarizes the resources needed to perform V&V tasks
   (c) identifies the organization responsible for performing V&V tasks
   (d) none of the above
6. Test log is _____.
   (a) an evaluation report prepared when the testing is over
   (b) a form of bug report
   (c) a record of the testing events that take place during test
   (d) none of the above
7. Test incident report is _____.
   (a) an evaluation report prepared when the testing is over
   (b) a form of bug report
   (c) a record of the testing events that take place during test
   (d) none of the above
8. Test summary report is _____.
   (a) an evaluation report prepared when the testing is over
   (b) a form of bug report
   (c) a record of the testing events that take place during test
   (d) none of the above
9. Test incident report is _____.
   (a) test procedure specification document
   (b) test result specification document
   (c) test design specification document
   (d) none of the above
10. _____ integration testing is preferred while planning integration test.
   (a) top-down
   (b) bottom-up
   (c) sandwich
   (d) none of the above

11. Test engineer is responsible for _____.
    (a) test planning the given tasks by the test manager
    (b) providing tool training, if needed
    (c) interacting with customer
    (d) designing the test cases
12. Junior test engineer _____.
    (a) participates in test design and execution with experienced test engineers
    (b) assigns testing tasks to test engineers who are working under him
    (c) supervises test engineers
    (d) none of the above
13. Test leader is _____.
    (a) responsible for making test strategies with detailed master planning and schedule
    (b) interacts with customer regarding quality issues
    (c) acquires all the testing resources including tools
    (d) assigns testing tasks to test engineers who are working under him
14. Test manager _____.
    (a) interacts with customer regarding quality issues
    (b) acquires all the testing resources including tools
    (c) designs test cases
    (d) develops test harness

Review Questions

1. Discuss the key components of test management.
2. What are the major activities of a testing group?
3. What is meant by testing group hierarchy? Explain the role of each member in this hierarchy.
4. Suppose you are working in the testing group of a company. Identify your role in the testing group hierarchy. What are the major duties performed by you there?
5. What type of test planning will you plan for a real-time critical software? Design a test plan using test plan components.
6. How can the users/clients help in preparing the test plans?
7. What are the major activities in V&V planning?
8. Acquire SRS and SDD of any project and develop the following:
    (a) Unit test plan
    (b) Integration test plan
    (c) System test plan

9. Explain the importance of test harness in the integration test plan.
10. What is the difference between system test plan and acceptance test plan?
11. Discuss the role of traceability matrix in designing the test cases.
12. Discuss the role of test log, test incident report, and test summary report in validation testing.
13. Take three modules of your choice in a project and prepare the following for each module:
    (a) Test design specifications
    (b) Test case specifications
    (c) Test procedure specifications
    (d) Test result specifications
14. The modules taken in Question 13 need to be integrated. Which test planning will you choose for integrating the modules?
15. What is functional decomposition?
16. List all the testing deliverables (documents) and describe the purpose of each, in the organization where you are working.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter

10

Software Metrics

OBJECTIVES

After reading this chapter, you should be able to understand:

- Software projects need measurement to quantify things for better monitoring and control over software development
- Software metrics are quantitative measurements
- Various types of software metrics
- Entities to be measured for the software: process, product, and resource
- Recognition of attributes before measurement, as they are important for designing software metrics
- Line-of-code metrics
- Halstead metrics
- Function point analysis metrics

Today, every technical process demands measurement. We cannot describe a product by simply saying that it should be of high quality or that it should be reliable. Today, the demand is to quantify the terms 'good or high quality', 'more reliable', 'efficient', etc. It is a world of quantification. This becomes more important in terms of software, as its development is becoming more and more complex.

With the increase in complexity, it becomes important to control the software development process to monitor its progress on factors like time, budget, and resource constraints, and measure the quality of the end-product. Thus, if we want to control all these parameters in software, we need software metrics.

DeMarco rightly said, *You cannot control what you cannot measure*. Sometimes, we don't know what can derail a project, therefore it is essential to measure and record the characteristics of good projects as well as bad ones. Therefore, quantification has become a necessity for the effective management of the software process.

Software metrics are used by the software industry to quantify the development, operation, and maintenance of software. Metrics give us information regarding the status of an attribute of the software and help us to evaluate it in an objective way. The practice of applying software metrics to a software process and to a software product is a complex task. By evaluating an attribute of the software, we can know its status. From there, we can identify and classify what its situation is; which helps us to find opportunities of improvements in the software. This also helps us to make plans for modifications that need to be implemented in the future.

Software metrics play an important role in measuring the attributes that are critical for the success of a software project. Measurement of these attributes helps to make the characteristics and relationships between the attributes clearer. This in turn supports informed decision-making. Measuring the attributes of a development process enables the management to have a better insight into the process. Thus, measurement is a tool through which the management identifies important events and trends, enabling them to make informed decisions. Moreover, measurements help in predicting outcomes and evaluating risks, which in turn decreases the probability of unanticipated surprises in different processes.

10.1 Need of Software Measurement

Measurements are a key element for controlling software engineering processes. By controlling, it is meant that one can assess the status of the process, observe the trends to predict what is likely to happen, and take corrective action for modifying our practices. Measurements also play their part in increasing our understanding of the process by making visible relationships among process activities and entities involved. Lastly, measurements improve our processes by modifying the activities based on different measures.

Figure 10.1 Need for software metrics

On the basis of this discussion, software measurement is needed for the following activities (see Fig. 10.1):

Understanding Metrics help in making the aspects of a process more visible, thereby giving a better understanding of the relationships among the activities and entities they affect.

Control Using baselines, goals, and an understanding of the relationships, we can predict what is likely to happen and correspondingly, make appropriate changes in the process to help meet the goals.

Improvement By taking corrective actions and making appropriate changes, we can improve a product. Similarly, based on the analysis of a project, a process can also be improved.

10.2 Definition of Software Metrics

Software metrics can be defined as 'the continuous application of measurement-based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products.'

The IEEE Standard Glossary of Software Engineering Terms [21] defines a metric as 'a quantitative measure of the degree to which a system component or process possesses a given attribute.'

10.3 Classification of Software Metrics

10.3.1 Product vs. Process Metrics

Software metrics may be broadly classified as either *product metrics* or *process metrics*. Product metrics are measures of the software product at any stage of its development, from requirements to installed system. Product metrics may measure the complexity of the software design, the size of the final program, or the number of pages of documentation produced.

Process metrics, on the other hand, are measures of the software development process, such as the overall development time, type of methodology used, or the average level of experience of the programming staff.

10.3.2 Objective vs. Subjective Metrics

Objective measures should always result in identical values for a given metric, as measured by two or more qualified observers. For subjective measures, even qualified observers may measure different values for a given metric. For example, for product metrics, the size of product measured in line of code (LOC) is an objective measure. In process metrics, the development time is an example of objective measure, while the level of a programmer's experience is likely to be a subjective measure.

10.3.3 Primitive vs. Computed Metrics

Primitive metrics are those metrics that can be directly observed, such as the program size in LOC, the number of defects observed in unit testing, or the

total development time for the project. Computed metrics are those that cannot be directly observed but are computed in some way from other metrics. For example, productivity metrics like LOC produced per person-month or product quality like defects per thousand lines of code.

10.3.4 Private vs. Public Metrics

This classification is based on the use of different types to process data. Because it is natural that individual software engineers might be sensitive to the use of metrics collected on an individual basis, these data should be private to individuals and serve as an indicator for individuals only. Examples of private metrics include defect rates (by individual and by module) and errors found during development.

Public metrics assimilate information that originally was private to individuals and teams. Project-level defect rates, effort, calendar times, and related data are collected and evaluated in an attempt to uncover indicators that can improve organizational process performance.

10.4 Entities to be Measured

In order to measure, it is needed to identify an entity and a specific attribute of it. It is very important to define clearly what is being measured, otherwise, the measures cannot be performed or the measures obtained can have different meaning for different people.

The entities considered in software measurement are:

- ***Processes*** Any activity related to software development.
- ***Product*** Any artifact produced during software development.
- ***Resource*** People, hardware, or software needed for a process.

The attributes of an entity can be internal or external.

- ***Internal attributes*** of any entity can be measured only based on the entity and therefore, measured directly. For example, size is an internal attribute of any software measurement.
- ***External attributes*** of any entity can be measured only with respect to how the entity is related with the environment and therefore, can only be measured indirectly. For example, reliability, an external attribute of a program, does not depend only on the program itself but also on the compiler, machine, and user. Productivity, an external attribute of a person, clearly depends on many factors such as the kind of process and the quality of the software delivered.

10.5 Size Metrics

The software size is an important metric to be used for various purposes. At the same time, it is difficult to measure because, unlike other physical products, a software cannot be measured directly with conventional units. Various approaches used for its measurement are given below.

10.5.1 Line of Code (LOC)

This metric is based on the number of lines of code present in the program. The lines of code are counted to measure the size of a program. The comments and blank lines are ignored during this measurement. The LOC metric is often presented on thousands of lines of code (KLOC). It is often used during the testing and maintenance phases, not only to specify the size of the software product, but also it is used in conjunction with other metrics to analyse other aspects of its quality and cost.

10.5.2 Token Count (Halstead Product Metrics)

The problem with LOC is that it is not consistent, because all lines of code are not at the same level. Some lines are more difficult to code than others. Another metric set has been given by Halstead. He stated that any software program could be measured by counting the number of operators and operands. From these set of operators and operands, he defined a number of formulae to calculate the vocabulary, the length, and the volume of the software program. Halstead extended this analysis to determine the effort and time. Some Halstead metrics are given below.

Program Vocabulary

It is the number of unique operators plus the number of unique operands as given below:

$$n = n1 + n2$$

where n = program vocabulary

$n1$ = number of unique operators

$n2$ = number of unique operands

Program Length

It is the total usage of all the operators and operands appearing in the implementation. It is given as,

$$N = N1 + N2$$

where N = program length

$N1$ = all operators appearing in the implementation
$N2$ = all operands appearing in the implementation

Program Volume

The volume refers to the size of the program and it is defined as the program length times the logarithmic base 2 of the program vocabulary. It is given as,

$$V = N \log_2 n$$

where V = program volume
N = program length
n = program vocabulary

10.5.3 Function Point Analysis (FPA)

It is based on the idea that the software size should be measured according to the functionalities specified by the user. Therefore, FPA is a standardized methodology for measuring various functions of a software from the user's point of view.

The size of an application is measured in function points. The process of counting the functions using FPA has been standardized by the International Function Point Users Group (IFPUG). IFPUG [123] has defined the rules and standards for calculating function points and it also promotes their use and evolution.

Process to Calculate Function Points

The process used to calculate the function points is given below [122]:

1. Determine the type of project for which the function point count is to be calculated. For example, development project (a new project) or enhancement project.
2. Identify the counting scope and the application boundary.
3. Identify data functions (internal logical functions and external interface files) and their complexity.
4. Identify transactional functions (external inputs, external outputs, and external queries) and their complexity.
5. Determine the unadjusted function point count (UFP).
6. Determine the value adjustment factor, which is based on 14 general system characteristics (GSCs).
7. Calculate the adjusted function point count (AFP).

Function point counting has been depicted in Fig. 10.2. This figure shows all the data functions and transactional functions. The details of all these functions are described in the following sections.

Figure 10.2 Functionality recognized in function point counting

Sizing Data Functions

Data functions are those functions in the project which relate to the logical data stored and available for update, reference, and retrieval. These functions are categorized in two parts: internal logical functions and external interface files. The categories are logical groupings of logically related data, and not physical representations. These are discussed below.

- ***Internal logical files (ILF)*** An internal logical file is a user-identifiable group of logically related data or control information maintained within the boundary of the application.
- ***External interface files (EIF)*** An external interface file is a user-identifiable group of logically related data or control information referenced by the application but maintained within the boundary of a different application. An EIF counted for an application must be in another application.

The physical count of ILFs and EIFs, together with the relative functional complexity of each, determines the contribution of the data function types to the unadjusted function point. Each identified data function must be assigned a functional complexity based on the number of data element types (DETs) and record element types (RETs) associated with the ILF and EIF.

DETs are unique user-recognizable, non-repeatable fields or attributes. RETs are user-recognizable subgroups (optional or mandatory) of data elements contained within an ILF or EIF. Subgroups are typically represented in an entity relationship diagram as entity subtypes or attribute entities.

Functional complexity is the rating assigned to each data function. The rating is based on record types and DETs being counted, as shown in Table 10.1.

Table 10.1 Complexity matrix for ILFs and EIFs

RETs	DETs		
	1–19	20–50	>=51
1	Low	Low	Average
2–5	Low	Average	High
>5	Average	High	High

Sizing Transactional Functions

Information systems are usually developed to mechanize and automate manual tasks. The tasks that have been automated are identified as transactional functions which represent the functionality provided to the user for processing the data by an application. The categorization of these functions is given below.

External inputs (EIs) are incoming data or control information to alter the behaviour of a system, e.g. adding, changing, deleting, etc.

External inquiries (EQs) send data outside the application through retrievals of data or control information from ILFs and/or EIFs, e.g. retrieval and display of a current description from our file.

External outputs (EOs) send data outside the application with processing logic other than or in addition to retrieval of data or control information, e.g. report of monthly sales with calculation by category.

Complexity and Contribution

The physical count of EIs, EOs, and EQs together with the relative functional complexity for each, determines the contribution of external inputs, outputs, and queries to the unadjusted function point count. Each identified EI, EO, and EQ must be assigned a functional complexity based on the number of DETs and FTRs (file type referenced) associated with them. Complexity matrices for EIs, EOs, and EQs are shown in Tables 10.2, 10.3, and 10.4.

Table 10.2 Complexity matrix for EIs

FTRs	DETs		
	1–4	5–15	>=16
<2	Low	Low	Average
2	Low	Average	High
>2	Average	High	High

Table 10.3 Complexity matrix for EOs

FTRs	DETs		
	1–5	6–19	>=20
<2	Low	Low	Average
2–3	Low	Average	High
>3	Average	High	High

Table 10.4 Complexity matrix for EQs

FTRs	DETs		
	1–5	6–19	>=20
<2	Low	Low	Average
2–3	Low	Average	High
>3	Average	High	High

Calculating Unadjusted Function Point (UFP)

The steps for calculating UFP are given below:

- Count all DETs/FTRs for all five components, i.e. ILF, EIF, EI, EO, and EQ of an application and determine the level of the component as low, average, or high, based on the individual complexity matrix, as described above.
- Count the number of all five components. The sum of each count is multiplied by an appropriate weight using Table 10.5, as shown below.

Table 10.5 IFPUGs unadjusted function point table

Components	Function Levels		
	Low	Average	High
ILF	X7	X10	X15
EIF	X5	X7	X10
EI	X3	X4	X6
EO	X4	X5	X7
EQ	X3	X4	X6

- Add all the five results calculated in the previous step. This is the final UFP.

Calculating Adjusted Function Point

There are some characteristics regarding every project which must be considered for the effort being measured for the project. Therefore, the function point calculated above is unadjusted, as the project dependent characteristics have not been considered in the function count. Therefore, IFPUG defines a

value adjustment factor (VAF) which is used as a multiplier of the unadjusted function point count in order to calculate the adjusted function point (AFP) count of an application.

Each GSC is evaluated in terms of its degree of influence (DI) on a scale of 0 to 5, as given below in Table 10.6.

Table 10.6 Degree of influence scaling

DI	Meaning
0	Not present, or no influence
1	Incidental influence
2	Moderate influence
3	Average influence
4	Significant influence throughout

Table 10.7 shows the GSCs to calculate the VAF.

Table 10.7 Components of VAF

Factor	Meaning
F1	Data communications
F2	Performance
F3	Transaction rate
F4	End user efficiency
F5	Complex processing
F6	Installation ease
F7	Multiple sites
F8	Distributed data processing
F9	Heavily used configuration
F10	Online data entry
F11	Online update
F12	Reusability
F13	Operational ease
F14	Facilitate change

The 14 GSCs shown in the table are summarized in the VAF. After applying the VAF, the UFP adjusts by 35% to determine the adjusted function point. The following steps determine the AFP.

- Evaluate the 14 GSCs on a scale of 0–5 to determine the DI for each GSC description.
- Add the DIs for all 14 GSCs to produce the total degree of influence (TDI).

- Use the TDI in the following equation to compute VAF.

$$VAF = (TDI \times 0.01) + 0.065$$

- The final adjusted function point is calculated as,

$$AFP = UFP \times VAF$$

Example 10.1

Consider a project with the following parameters: EI = 50, EO = 40, EQ = 35, ILF = 06, and ELF = 04. Assume all weighing factors are average. In addition, the system requires critical performance, average end-user efficiency, moderate distributed data processing, and critical data communication. Other GSCs are incidental. Compute the function points using FPA.

Solution

$$UFP = 50 \times 4 + 40 \times 5 + 35 \times 4 + 6 \times 10 + 4 \times 7 = 628$$

$$TDI = (4 + 3 + 2 + 4) \times 10 = 130$$

$$VAF = 130 \times 0.01 + 0.065 = 1.365$$

$$AFP = 628 \times 1.365 = 857.22$$

SUMMARY

Software measurement is not only useful but necessary. After all, how do you decide that a project is under control, if there is no measure for this? Software metrics play an important role today in understanding, controlling, and improving the software process as well as the product.

This chapter discusses the need for software measurement with its various types of metrics. Function point analysis is the major size metric used to measure the size of software. The technique has been adopted for measuring the development and testing effort. The chapter discusses the whole process of calculating the function points for a project to be developed. This chapter makes ground for the next chapter which is about software testing metrics.

Let us review the important concepts described in this chapter:

- Software metrics play an important role in measuring attributes that are critical to the success of a software project. Measurement of these attributes helps to make the characteristics and relationships between the attributes clearer.
- Software metric can be defined as a quantitative measure of the degree to which a system component or process possesses a given attribute.
- LOC (line of code) metric is based on the number of lines of code present in the program. The lines of code are counted for measuring the size of the program.
- Token count metric is based on counting the number of operators and operands in a program. From this set of operators and operands, a number of formulae are defined to calculate the vocabulary, the length, and the volume of the software program.

- Function point analysis (FPA) is a standardized methodology for measuring the various functions of a software from the user's point of view. The size of an application is measured in function points.
- Data functions are those functions in the project which relate to logical data stored and available for update, reference, and retrieval.
- An internal logical file (ILF) is a user-identifiable group of logically related data or control information, maintained within the boundary of the application.
- An external logical file (EIF) is a user-identifiable group of logically related data or control information, referenced by the application, but maintained within the boundary of a different application. An EIF counted for an application must be in another application.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Metrics give us information regarding the status of _____ in a software.
   (a) attributes
   (b) processes
   (c) costs
   (d) none of the above
2. _____ activity does not require software measurement.
   (a) Development
   (b) Understanding
   (c) Control
   (d) Improvement
3. Primitive metrics are those metrics that can be _____observed.
   (a) directly
   (b) indirectly
   (c) user
   (d) none of the above
4. Data functions are those functions in the project which relate to _____ data stored.
   (a) physical
   (b) logical
   (c) complex
   (d) none of the above
5. VAF = (TDI × 0.01) + _____
   (a) 0.076
   (b) 0.075
   (c) 0.065
   (d) none of the above

6. VAF adjusts the UFP by _____ to determine the adjusted function point.
   (a) 34%
   (b) 35%
   (c) 43%
   (d) none of the above
7. AFP = UFP __ VAF
   (a) +
   (b) *
   (c) –
   (d) /

Review Questions

1. What is the need for software measurement?
2. Discuss the various types of software metrics.
3. What is the disadvantage of LOC metrics?
4. What is the basis of Halstead metrics to calculate the size of a software?
5. What type of projects can be best counted with function point analysis?
6. Explain the process of calculating function points for a project which you are going to build.
7. Can you adopt the FPA for calculating the function points of a real-time software?
8. What is the difference between UFP and AFP?
9. Consider a project with the following parameters: EI= 60, EO= 40, EQ = 45, ILF = 06, ELF = 08. Assume all weighing factors are average. In addition, the system requires significant data communications, performance is very critical, designed code may be moderately reusable, and other GSCs are average. Compute the function points using FPA.
10. Consider a project with the following components: EI (simple) = 30, EO (average) = 20, EQ (average) = 35, ILF (complex) = 08, ELF (complex) = 05. In addition, the system requires significant end-user efficiency, moderate distributed data processing, critical data communications, and other GSCs are incidental. Compute the function points for this system using FPA.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter

11

Testing Metrics for Monitoring and Controlling the Testing Process

OBJECTIVES

After reading this chapter, you should be able to understand:

- Testing metrics are important to monitor and control all testing activities
- Attributes for software testing on the basis of which testing metrics are designed
- Progress metrics
- Quality metrics
- Cost metrics
- Size metrics
- Testing effort estimation models: Halstead model, development ratio method, project-staff ratio method, test procedure method, task planning method, function point method, test-point method

Software testing is the most time-consuming part of SDLC. Therefore, managing the software testing process is a real challenge. Everyone in the project team has some questions regarding the testing process.

- How efficient is our testing?
- Is it worth doing verification as opposed to validation?
- What is the thoroughness of validation testing?
- What kind of errors are we going to detect and how many?
- How many errors remain in the product, even after testing?
- When should we stop testing?

Measurement answers all these questions. Test execution can be tracked effectively using software metrics. Software metrics play a vital role in understanding, controlling, and improving the test process. The organizations that can control their software testing processes are able to predict costs and schedules and increase the effectiveness, efficiency, and profitability of their business [113]. Therefore, knowing and measuring what is being done is important for an effective testing effort [112].

Thus, assessment of the effectiveness of a software testing process has to rely on appropriate measures. If these measures are embedded in the organizational-level testing strategy, they make the underlying testing process activities visible. It helps the managers and engineers to better acknowledge the connections among various process activities. Moreover, measurement helps in improving the software testing procedures, methods, tools, and activities, by providing objective evidence for evaluations and comparisons.

The benefits of measurement in software testing are [35]:

- Identifying the strengths and weaknesses of testing
- Providing insights into the current state of the testing process
- Evaluating testing risks
- Benchmarking
- Improving the planning process
- Improving the effectiveness of testing
- Evaluating and improving the product quality
- Measuring the productivity
- Determining the level of customer involvement and satisfaction
- Supporting, controlling, and monitoring the testing process
- Comparing the processes and products with those inside and outside the organization

11.1 Measurement Objectives for Testing

The objectives for assessing a test process should be well-defined. A number of metrics programs begin by measuring what is convenient or easy to measure, rather than by measuring what is needed. Such programs often fail because the resulting data are not useful to anyone [32]. A measurement program will be more successful, if it is designed with the goals of the project in mind. For this purpose, we can take the help of a *goal question metric* (GQM), first suggested by Basili and Rombach.

The GQM approach is based on the fact that the objectives of measurement should be clear, much before the data collection begins. According to this approach, the objectives of a measurement process should be identified first. The GQM approach, with reference to test process measurement, provides the following framework:

- Lists the major goals of the test process.
- Derives from each goal, the questions that must be answered to determine if the goals are being met.
- Decides what must be measured in order to answer the questions adequately.

In this way, we generate only those measures that are related to the goals. In many cases, several measurements may be required to answer a single question. Likewise, a single measurement may apply to more than one question.

Software measurement is effective only when the metrics are used and analysed in conjunction with one another. We will now discuss the metrics for testing.

11.2 Attributes and Corresponding Metrics in Software Testing

An organization needs to have a reference set of measurable attributes and corresponding metrics that can be applied at various stages during the course of execution of an organization-wide testing strategy. The attributes to be measured depend on the following factors:

- Time and phase in the software testing lifecycle.
- New business needs.
- Ultimate goal of the project.

Therefore, we discuss measurable attributes for software testing and the corresponding metrics based on the attribute categorization done by Wasif Afzal and Richard Torkar [121], as shown in Table 11.1. These attributes are discussed in the subsequent sections.

Table 11.1 Attribute categories

Category	Attributes to be Measured
Progress	• Scope of testing • Test progress • Defect backlog • Staff productivity • Suspension criteria • Exit criteria
Cost	• Testing cost estimation • Duration of testing • Resource requirements • Training needs of testing group and tool requirement • Cost-effectiveness of automated tool
Quality	• Effectiveness of test cases • Effectiveness of smoke tests • Quality of test plan • Test completeness
Size	• Estimation of test cases • Number of regression tests • Tests to automate

11.3 Attributes

11.3.1 Progress

The following attributes are defined under the progress category of metrics:

- Scope of testing
- Test progress
- Defect backlog
- Staff productivity
- Suspension criteria
- Exit criteria

Scope of testing It helps in estimating the overall amount of work involved, by documenting which parts of the software are to be tested [114], thereby estimating the overall testing effort required. It also helps in identifying the testing types that are to be used for covering the features under test.

Tracking test progress The progress of testing should also be measured to keep in line with the schedule, budget, and resources. If we monitor the progress with these metrics by comparing the actual work done with the planning, then the corrective measures can be taken in advance, thereby controlling the project.

For measuring the test progress, well-planned test plans are prepared wherein testing milestones are planned. Each milestone is scheduled for completion during a certain time period in the test plan. These milestones can be used by the test manager to monitor how the testing efforts are progressing. For example, executing all planned unit tests is one example of a testing milestone. Measurements need to be available for comparing the planned and actual progress towards achieving the testing milestones. To execute the testing milestones of all planned unit tests, the data related to the number of planned unit tests currently available and the number of executed unit tests on this date should be available.

The testing activity being monitored can be projected using graphs that show the trends over a selected period of time. The test progress *S* curve compares the test progress with the plan to indicate corrective actions, in case the testing activity is falling behind schedule. The advantage of this curve is that schedule slippage is difficult to ignore. The test progress *S* curve shows the following set of information on one graph [42]:

- Planned number of test cases to be completed successfully by a week.
- Number of test cases attempted in a week.
- Number of test cases completed successfully by a week.

Defect backlog It is the number of defects that are outstanding and unresolved at one point of time with respect to the number of defects occurring. The defect backlog metric is to be measured for each release of a software product for release-to-release comparisons [42]. One way of tracking it is to use *defect removal percentage* [112] for a prior release so that decisions regarding additional testing can be wisely taken in the current release. If the backlog increases, the bugs should be prioritized according to their criticality.

Staff productivity Measurement of testing staff productivity helps to improve the contributions made by the testing professionals towards a quality testing process. For example, as test cases are developed, it is interesting to measure the productivity of the staff in developing these test cases. This estimation is useful for the managers to estimate the cost and duration for testing activities.

The basic and useful measures of a tester's productivity are [60]:

- Time spent in test planning.
- Time spent in test case design.
- Number of test cases developed.
- Number of test cases developed per unit time.

Suspension criteria These are the metrics that establish conditions to suspend testing. It describes the circumstances under which testing would stop temporarily. The suspension criteria of testing describes in advance that the occurrence of certain events/conditions will cause the testing to stop temporarily. With the help of suspension metrics, we can save precious testing time by raising the conditions that suspend testing. Moreover, it indicates the shortcomings of the application and the environmental setup.

The following conditions indicate the suspension of testing:

- There are incomplete tasks on the critical path.
- Large volumes of bugs are occurring in the project.
- Critical bugs.
- Incomplete test environments (including resource shortages).

Exit criteria The exit criteria are established for all the levels of a test plan. It indicates the conditions that move the testing activities forward from one level to the next. If we are clear when to exit from one level, the next level can start. On the other hand, if the integration testing exit criteria are not strict, then system testing may start before the completion of integration testing.

The exit criteria determine the termination of testing effort which must be communicated to the development team prior to the approval of the test plan.

The exit criteria should also be standardized. If there are any open faults, then the software development manager, along with the project manager and the members of the change control board, decides whether to fix the faults, or defer them to the next release, or take the risk of shipping it to the customer with the faults.

The following metrics are studied for the exit criteria of testing:

- Rate of fault discovery in regression tests.
- Frequency of failing fault fixes.
- Fault detection rate.

11.3.2 Cost

The following attributes are defined under the cost category of metrics:

- Testing cost estimation
- Duration of testing
- Resource requirements
- Training needs of testing groups and tool requirements
- Cost effectiveness of automated tools

Testing cost estimation The metrics supporting the budget estimation of testing need to be established early. It also includes the cost of test planning itself.

Duration of testing There is also a need to estimate the testing schedule during test planning. As a part of the testing schedule, the time required to develop a test plan is also estimated. A testing schedule contains the timelines of all testing milestones. The testing milestones are the major testing activities carried out according to a particular level of testing. A work break-down structure is used to divide the testing efforts into tasks and activities. The timelines of activities in a testing schedule matches the time allocated for testing in the overall project plan.

Resource requirements Test planning activities have to estimate the number of testers required for all testing activities with their assigned duties planned in the test plan.

Training needs of testing groups and tool requirements Since the test planning also identifies the training needs of testing groups and their tool requirements, we need to have metrics for training and tool requirements.

Cost-effectiveness of automated tools When a tool is selected to be used for testing, it is beneficial to evaluate its cost-effectiveness. The cost-effectiveness

is measured by taking into account the cost of tool evaluation, tool training, tool acquisition, and tool update and maintenance.

11.3.3 Quality

The following attributes are defined under this category of metrics:

- Effectiveness of test cases
- Effectiveness of smoke tests
- Quality of test plan
- Test completeness

Effectiveness of test cases The test cases produced should be effective so that they uncover more and more bugs. The measurement of their effectiveness starts from the test case specifications. The specifications must be verified at the end of the test design phase for conformance with the requirements. While verifying the specifications, emphasis must be given to those factors that affect the effectiveness of test cases including the test cases with incomplete functional specifications, poor test designs, and wrong interpretation of test specifications by the testers [110].

Common methods available for verifying test case specifications include inspections and traceability of test case specifications to functional specifications. These methods seldom help to improve the fault-detecting ability of test case specifications.

There are several measures based on faults to check the effectiveness of test cases. Some of them are discussed here:

1. *Number of faults* found in testing.
2. *Number of failures* observed by the customer which can be used as a reflection of the effectiveness of test cases.
3. *Defect-removal efficiency* is another powerful metric for test-effectiveness, which is defined as the ratio of the number of faults actually found in testing and the number of faults that could have been found in testing. There are potential issues that must be taken into account while measuring the defect-removal efficiency. For example, the severity of bugs and an estimate of time by which the customers would have discovered most of the failures are to be established. This metric is more helpful in establishing the test effectiveness in the long run as compared to the current project.
4. *Defect age* is another metric that can be used to measure the test effectiveness, which assigns a numerical value to the fault, depending on the phase

in which it is discovered. Defect age is used in another metric called *defect spoilage* to measure the effectiveness of defect-removal activities. Defect spoilage is calculated as [117]:

$$\text{Spoilage} = \frac{\text{Sum of (Number of defects} \times \text{Defect age)}}{\text{Total number of defects}}$$

Spoilage is more suitable for measuring the long-term trend of test-effectiveness. Generally, low values of defect spoilage mean more effective defect discovery processes. The effectiveness of a test case can also be judged on the basis of the coverage provided. It is a powerful metric in the sense that it is not dependent on the quality of software and also, it is an in-process metric that is applicable when the testing is actually done.

Example 11.1

Consider a project with the following distribution of data and calculate its defect spoilage.

SDLC phase	No. of defects	Defect age
Requirement Specs.	34	2
HLD	25	4
LLD	17	5
Coding	10	6

Solution

$$\text{Spoilage} = (34 \times 2 + 25 \times 4 + 17 \times 5 + 10 \times 6)/86 = 3.64$$

Effectiveness of smoke tests Smoke tests are required to ensure that the application is stable enough for testing, thereby assuring that the system has all the functionality and is working under normal conditions. This testing does not identify faults, but establishes confidence over the stability of a system. The tests that are included in smoke testing cover the basic operations that are most frequently used, e.g. logging in, addition, and deletion of records. Smoke tests need to be a subset of the regression testing suite. It is time-consuming and human-intensive to run the smoke tests manually, each time a new build is received for testing. Automated smoke tests are executed against the software, each time the build is received, ensuring that the major functionality is working.

Quality of test plan The quality of the test plan produced is also a candidate attribute to be measured, as it may be useful in comparing different plans for different products, noting down the changes observed, and improving the test plans for the future. The test plan should also be effective in giving a high

number of errors. Thus, the quality of a test plan is measured in concern with the probable number of errors.

To evaluate a test plan, Berger describes a multi-dimensional qualitative method using *rubrics* [118]. Rubrics take the form of a table, where each row represents a particular dimension and each column represents a ranking of the dimension. There are ten dimensions that contribute to the philosophy of test planning. These ten dimensions are:

1. Theory of objective
2. Theory of scope
3. Theory of coverage
4. Theory of risk
5. Theory of data
6. Theory of originality
7. Theory of communication
8. Theory of usefulness
9. Theory of completeness
10. Theory of insightfulness

Against each of these theories, there is a rating of excellent, average, and poor, depending on certain criteria, as shown in Table 11.2.

Table 11.2 Measuring test plan through rubrics

Dimension	Rating		
	Excellent	**Average**	**Poor**
Theory of Objective	Identification of realistic test objectives to have the most efficient test cases.	Describes somewhat credible test objectives.	No objectives or irrelevant objectives.
Theory of Scope	Specific and unambiguous test scope.	Some scope of testing understood by only some people.	Wrong scope assumptions.
Theory of Coverage	The test coverage is completely relative to the test scope.	The test coverage is somewhat related to the test scope.	Coverage is not related to test scope or objective.
Theory of Risk	Identifies possible testing risks.	Risks have inappropriate priorities.	No understanding of project risk.
Theory of Data	Efficient method to generate enough valid and invalid test data.	Some method to generate test data.	There is no method of capturing data.

Theory of Originality	High information in test plan.	A test plan template has some original content.	Template has not been filled in.
Theory of Communication	Multiple modes to communicate test plan to, and receive feedback from appropriate stakeholders.	Less effective feedback.	No distribution, no opportunity for feedback
Theory of Usefulness	Effective test plan such that it discovers critical bugs early.	Critical bugs are not discovered earlier.	Test plans are not useful to the organization.
Theory of Completeness	Enough testing through the plan.	Most test items have been identified.	Enough testing criteria has not been defined.
Theory of Insightfulness	Understanding of what is interesting and challenging in testing this specific project.	The test plan is effective, but weak.	No insight through the plan.

Measuring test completeness This attribute refers to how much of code and requirements are covered by the test set. The advantages of measuring test coverage are that it provides the ability to design new test cases and improve existing ones. There are two issues: (i) whether the test cases cover all possible output states and (ii) the adequate number of test cases to achieve test coverage. The relationship between code coverage and the number of test cases is described by the following expression [119]:

$$C(x) = 1 - e^{-(p/N) * x}$$

where $C(x)$ is the coverage after executing x number of test cases, N is the number of blocks in the program, and p is the average number of blocks covered by a test case during the function test. The function indicates that test cases cover some blocks of code more than others, while increasing test coverage beyond a threshold is not cost-effective.

Test coverage for control flow and data flow can be measured using spanning sets of entities in a program flow graph. A spanning set is a minimum subset of the set of entities of the program flow graph such that a test suite covering the entities in this subset is guaranteed to cover every entity in the set [120].

At the system testing level, we should measure whether all the features of the product are being tested or not. Common requirements coverage metric is the percentage of requirements covered by at least one test [112]. A requirements traceability matrix can be used for this purpose.

11.3.4 Size

The following attributes are defined under the size category of metrics:

- Estimation of test cases
- Number of regression tests
- Tests to automate

Estimation of test cases To fully exercise a system and to estimate its resources, an initial estimate of the number of test cases is required. Therefore, metrics estimating the required number of test cases should be developed.

Number of regression tests Regression testing is performed on a modified program that establishes confidence that the changes and fixes against reported faults are correct and have not affected the unchanged portions of the program [51]. However, the number of test cases in regression testing becomes too large to test. Therefore, careful measures are required to select the test cases effectively.

Some of the measurements to monitor regression testing are [60]:

- Number of test cases re-used
- Number of test cases added to the tool repository or test database
- Number of test cases rerun when changes are made to the software
- Number of planned regression tests executed
- Number of planned regression tests executed and passed

Tests to automate Tasks that are repetitive in nature and tedious to perform manually are prime candidates for an automated tool. The categories of tests that come under repetitive tasks are:

- Regression tests
- Smoke tests
- Load tests
- Performance tests

If automation tools have a direct impact on the project, it must be measured. There must be benefits attached with automation including speed, efficiency, accuracy and precision, resource reduction, and repetitiveness. All these factors should be measured.

11.4 Estimation Models for Estimating Testing Efforts

After discussing the attributes needed to measure the testing efforts, we will see some estimation models and metrics that are being used.

11.4.1 Halstead Metrics

The metrics derived from Halstead measures described in Chapter 10 can be used to estimate testing efforts, as given by Pressman [116]. Halstead developed expressions for program volume V and program level PL which can be used to estimate testing efforts. The program volume describes the number of volume of information in bits required to specify a program. The program level is a measure of software complexity. Using these definitions, Halstead effort e can be computed as [124]:

$$PL = 1/[(n1/2) \times (N2/n2)]$$

$$e = V/PL$$

The percentage of overall testing effort to be allocated to a module k can be estimated using the following relationship:

$$\textbf{Percentage of testing effort } (k) = e(k)/\sum e(i)$$

where $e(k)$ is the effort required for module k and $\sum e(i)$ is the sum of Halstead effort across all modules of the system.

Example 11.2

In a module implementation of a project, there are 17 unique operators, 12 unique operands, 45 total operators, and 32 total operands appearing in the module implementation. Calculate its Halstead effort.

Solution

$$n = n1 + n2 = 17 + 12 = 92$$

$$N = N1 + N2 = 45 + 32 = 77$$

$$V = N \log_2 n = 77 \times \log_2 92 = 502.3175$$

$$\text{PL} = 1/[(n1/2) \times (N2/n2)] = 3/68 = 0.044$$

$$e = V/PL = 11416.30$$

11.4.2 Development Ratio Method

This model is based on the estimation of development efforts. The number of testing personnel required is estimated on the basis of the developer-tester ratio [12]. The results of applying this method is dependent on numerous factors including the type and complexity of the software being developed, testing level, scope of testing, test-effectiveness during testing, error tolerance level for testing, and available budget.

Another method of estimating tester-to-developer ratios, based on heuristics, is proposed by K. Iberle and S. Bartlett [115]. This method selects a baseline project(s), gathers testers-to-developers ratios, and collects data on various effects like developer-efficiency at removing defects before testing, developer-efficiency at inserting defects, defects found per person, and the value of defects found. After that, an initial estimate is made to calculate the number of testers based upon the ratio of the baseline project. The initial estimate is adjusted using professional experience to show how the above-mentioned effects affect the current project and the baseline project.

11.4.3 Project-Staff Ratio Method

Project-staff ratio method makes use of historical metrics by calculating the percentage of testing personnel from the overall allocated resources planned for the project [12]. The percentage of a test team size may differ according to the type of project. The template can be seen in Table 11.3.

Table 11.3 Project-staff ratio

Project type	Total number of project staff	Test team size %	Number of testers
Embedded system	100	23	23
Application development	100	8	8

11.4.4 Test Procedure Method

This model is based on the number of test procedures planned. The number of test procedures decides the number of testers required and the testing time [12]. Thus, the baseline of estimation here is the quantity of test procedures. But you have to do some preparation before actually estimating the resources. It includes developing a historical record of projects including the data related to size (e.g. number of function points, number of test procedures used) and test effort measured in terms of personnel hours. Based on the estimates of historical development size, the number of test procedures required for the new project is estimated.

The only thing to be careful about in this model is that the projects to be compared should be similar in terms of nature, technology, required expertise, and problems solved. The template for this model can be seen in Table 11.4. For a historical project, person-hours consumed for testing test procedures were observed and a factor in the form of number hours per test procedure was calculated. This factor is used for a new project where the number of test procedures are given. Using this factor, we will calculate the number of person-hours for the new project. Then, with the knowledge of the total expended period for this new project, we are able to calculate the number of testers required.

Table 11.4 Test procedure method

	Number of test procedures (NTP)	Number of person-hours consumed for testing (PH)	Number of hours per test procedure = PH/NTP	Total period in which testing is to be done (TP)	Number of testers = PH/TP
Historical Average Record	840	6000	7.14	10 months (1600 hrs)	3.7
New Project Estimate	1000	7140	7.14	1856 hrs	3.8

11.4.5 Task Planning Method

In this model, the baseline for estimation is the historical records of the number of personnel hours expended to perform testing tasks. The historical records collect data related to the work break-down structure and the time required for each task so that the records match the testing tasks [12]. First, the number of person-hours for the full project is calculated as in the test procedure method (see Table 11.5).

Table 11.5 Calculating number of person-hours

Number of Test Procedures (NTP)	Person-hours consumed for testing (PH)	Hours per test procedure = PH/NTP
840	6000	7.14
1000 (New project)	7140	7.14

The historical data is observed to see how much time has been consumed on individual test activities. After estimating the average idea on historical data, the total person-hours are distributed, which is later adjusted according to some conditions in the new project (see Table 11.6).

Table 11.6 Task planning method

Testing activity	Historical value	% time of the project consumed on the test activity	Preliminary estimate of person hours	Adjusted estimate of person-hours
Test planning	210	3.5	249	
Test design	150	2.5	178	
Test execution	180	3	214	
Project total	6000	100%	7140	6900

Next, the test team size is calculated using the total adjusted estimate of person-hours for the project, as given in Table 11.7.

Table 11.7 Test team size using task planning method

	NTP	PH (Adjusted estimate)	Number of hours per test procedure = PH/NTP	TP	Number of testers = PH/TP
New project estimate	1000	6900	6.9	1856 hrs	3.7

After getting an idea about the model for effort estimation, let us discuss some important metrics being used in various phases of testing.

11.5 Architectural Design Metric used for Testing

Card and Glass [57] introduced three types of software design complexity that can also be used in testing. These are discussed below.

Structural Complexity

It is defined as

$$S(m) = f^2_{out}(m)$$

where S is the structural complexity and $f_{out}(m)$ is the fan-out of module m.

This metric gives us the number of stubs required for unit testing of the module m. Thus, it can be used in unit testing.

Data Complexity

This metric measures the complexity in the internal interface for a module m and is defined as

$$D(m) = v(m) \ / \ [f_{\text{out}}(m) + 1]$$

where $v(m)$ is the number of input and output variables that are passed to and from module m.

This metric indicates the probability of errors in module m. As the data complexity increases, the probability of errors in module m also increases. For example, module X has 20 input parameters, 30 internal data items, and 20 output parameters. Similarly, module Y has 10 input parameters, 20 internal data items, and 5 output parameters. Then, the data complexity of module X is more as compared to Y, therefore X is more prone to errors. Therefore, testers should be careful while testing module X.

System Complexity

It is defined as the sum of structural and data complexity:

$$SC(m) = S(m) + D(m)$$

Since the testing effort of a module is directly proportional to its system complexity, it will be difficult to unit test a module with higher system complexity. Similarly, the overall architectural complexity of the system (which is the sum total of system complexities of all the modules) increases with the increase in each module's complexity. Consequently, the efforts required for integration testing increase with the architectural complexity of the system.

11.6 Information Flow Metrics used for Testing

Researchers have used information flow metrics between modules. For understanding the measurement, let us understand the way the data moves through a system:

- **Local direct flow** exists if
  - (i) a module invokes a second module and passes information to it.
  - (ii) the invoked module returns a result to the caller.
- **Local indirect flow** exists if the invoked module returns information that is subsequently passed to a second invoked module.
- **Global flow** exists if information flows from one module to another via a global data structure.

The two particular attributes of the information flow can be described as follows:

(i) **Fan-in** of a module *m* is the number of local flows that terminates at *m*, plus the number of data structures from which information is retrieved by *m*.

(ii) **Fan-out** of a module *m* is the number of local flows that emanate from *m*, plus the number of data structures that are updated by *m*.

11.6.1 Henry and Kafura Design Metric

Henry and Kafura's information flow metric is a well-known approach for measuring the total level of information flow between individual modules and the rest of the system. They measure the information flow complexity as

$$IFC(m) = length(m) \times ((fan\text{-}in(m) \times fan\text{-}out(m))^2$$

Higher the *IF* complexity of *m*, greater is the effort in integration and integration testing, thereby increasing the probability of errors in the module.

11.7 Cyclomatic Complexity measures for Testing

McCabe's cyclomatic complexity was discussed in the previous chapter. This measure can be used in software testing in the following ways:

- Since the cyclomatic number measures the number of linearly independent paths through flow graphs, it can be used as the set of minimum number of test cases. If the number of test cases is lesser than the cyclomatic number, then you have to search for the missing test cases. In this way, it becomes a thumb rule that cyclomatic number provides the minimum number of test cases for effective branch coverage.
- McCabe has suggested that ideally, cyclomatic number should be less than or equal to 10. This number provides a quantitative measure of testing difficulty. If the cyclomatic number is more than 10, then the testing effort increases due to the following reasons:
  1. Number of errors increase.
  2. Time required to detect and correct the errors increase.

Thus, the cyclomatic number is an important indication of the ultimate reliability. The amount of test design and test effort is better approximated by the cyclomatic number as compared to the lines of code (LOC) measure. A project in USA that applied these rules did achieve zero defects for at least 12 months after its delivery [34].

11.8 Function Point Metrics for Testing

The function point (FP) metric is used effectively for measuring the size of a software system. Function-based metrics can be used as a predictor for the overall testing effort. Various project-level characteristics (e.g. testing effort and time, errors uncovered, number of test cases produced) of past projects can be collected and correlated with the number of FP produced by a project team. The team can then project the expected values of these characteristics for the current project.

Listed below are a few FP measures:

1. Number of hours required for testing per FP.
2. Number of FPs tested per person-month.
3. Total cost of testing per FP.
4. Defect density measures the number of defects identified across one or more phases of the development project lifecycle and compares that value with the total size of the system. It can be used to compare the

density levels across different lifecycle phases or across different development efforts. It is calculated as

Number of defects (by phase or in total) / Total number of FPs

5. Test case coverage measures the number of test cases that are necessary to adequately support thorough testing of a development project. This measure does not indicate the effectiveness of test cases, nor does it guarantee that all conditions have been tested. However, it can be an effective comparative measure to forecast anticipated requirements for testing that may be required on a development system of a particular size. This measure is calculated as

Number of test cases / Total number of FPs

Capers Jones [146] estimates that the number of test cases in a system can be determined by the function points estimate for the corresponding effort. The formula is

$$\text{Number of test cases} = (\text{function points})^{1.2}$$

Function points can also be used to measure the acceptance test cases. The formula is

$$\text{Number of test cases} = (\text{function points}) \times 1.2$$

The above relationships show that test cases grow at a faster rate than function points. This is intuitive because, as an application grows, the number of inter-relationships within the applications becomes more complex. For example, if a development application has 1,000 function points, there should be approximately 4,000 total test cases and 1,200 acceptance test cases.

Example 11.3

In a project, the estimated function points are 760. Calculate the total number of test cases in the system and the number of test cases in acceptance testing. Also, calculate the defect density (number of total defects is 456) and test case coverage.

Solution

Total number of test cases $= (760)^{1.2} = 2864$

Number of test cases for acceptance testing $= (760) \times 1.2 = 912$

Defect density $= 456/760 = 0.6$

Test case coverage $= 2864/760 = 3.768$

11.9 Test Point Analysis (TPA)

Test point analysis is a technique to measure the black-box test effort estimation, proposed by Drs Eric P W M Van Veenendaal CISA and Ton Dekkers [111]. The estimation is for system and acceptance testing. The technique uses function point analysis in the estimation. TPA calculates the test effort estimation in test points for highly important functions, according to the user and also for the whole system. As the test points of the functions are measured, the tester can test the important functionalities first, thereby predicting the risk in testing. This technique also considers the quality characteristics, such as functionality, security, usability, efficiency, etc. with proper weightings.

11.9.1 Procedure for Calculating TPA

For TPA calculation, first the dynamic and static test points are calculated (see Fig. 11.1). *Dynamic test points* are the number of test points which are based on dynamic measurable quality characteristics of functions in the system. Dynamic test points are calculated for every function. To calculate a dynamic function point, we need the following:

- Function points (FPs) assigned to the function.
- Function dependent factors (FDC), such as complexity, interfacing, function-importance, etc.
- Quality characteristics (QC).

The dynamic test points for individual functions are added and the resulting number is the dynamic test point for the system.

Similarly, *static test points* are the number of test points which are based on static quality characteristics of the system. It is calculated based on the following:

- Function points (FPs) assigned to the system.
- Quality requirements or test strategy for static quality characteristics (QC).

After this, the dynamic test point is added to the static test point to get the total test points (TTP) for the system. This total test point is used for calculating *primary test hours* (PTH). PTH is the effort estimation for primary testing activities, such as preparation, specification, and execution. PTH is calculated based on the environmental factors and productivity factors. Environmental factors include development environment, testing environment, testing tools,

etc. Productivity factor is the measure of experience, knowledge, and skills of the testing team.

Secondary testing activities include management activities like controlling the testing activities. *Total test hours* (TTH) is calculated by adding some allowances to secondary activities and PTH. Thus, TTH is the final effort estimation for the testing activities.

Figure 11.1 Procedure for test point analysis

11.9.2 Calculating Dynamic Test Points

The number of dynamic test points for each function in the system is calculated as

$$DTP = FP \times FDC_w \times QC_{dw}$$

where DTP = number of dynamic test points

FP = function point assigned to function.

FDC_w = weight-assigned function-dependent factors

QC_{dw} = quality characteristic factor, wherein weights are assigned to dynamic quality characteristics

FDC_w is calculated as

$$FDC_w = ((FI_w + UIN_w + I + C) \div 20) \times U$$

where FI_w = function importance rated by users

UIN_w = weights given to usage intensity of the function, i.e. how frequently the function is being used

I = weights given to the function for interfacing with other functions, i.e. if there is a change in function, how many functions in the system will be affected

C = weights given to the complexity of function, i.e. how many conditions are in the algorithm of function

U = uniformity factor

The ratings done for providing weights to each factor of FDC_w can be seen in Table 11.8.

Table 11.8 Ratings for FDC_w factors

Factor/ Rating	Function Importance (FI)	Function usage Intensity (UIN)	Interfacing (I)	Complexity (C)	Uniformity Factor
Low	3	2	2	3	0.6 for the function, wherein test specifications are largely re-used such as in clone function or dummy function. Otherwise, it is 1.
Normal	6	4	4	6	
High	12	12	8	12	

QC_{dw} is calculated based on four dynamic quality characteristics, namely suitability, security, usability, and efficiency. First, the rating to every quality characteristic is given and then, a weight to every QC is provided. Based on the rating and weights, QC_{dw} is calculated.

$$QC_{dw} = \sum (\text{rating of } QC / 4) \times \text{weight factor of } QC$$

Rating and weights are provided based on the following characteristics, as shown in Table 11.9.

Table 11.9 Ratings and weights for QC_{dw}

Characteristic/ Rating	Not Important (0)	Relatively Unimportant (3)	Medium Importance (4)	Very Important (5)	Extremely Important (6)
Suitability	0.75	0.75	0.75	0.75	0.75
Security	0.05	0.05	0.05	0.05	0.05
Usability	0.10	0.10	0.10	0.10	0.10
Efficiency	0.10	0.10	0.10	0.10	0.10

11.9.3 Calculating Static Test Points

The number of static test points for the system is calculated as

$$STP = FP \times \sum QC_{sw} / 500$$

where STP = static test point
FP = total function point assigned to the system
QC_{sw} = quality characteristic factor, wherein weights are assigned to static quality characteristics

Static quality characteristic refers to what can be tested with a checklist. QC_{sw} is assigned the value 16 for each quality characteristic which can be tested statically using the checklist.

$$Total\ test\ points\ (TTP) = DTP + STP$$

11.9.4 Calculating Primary Test Hours

The total test points calculated above can be used to derive the total testing time in hours for performing testing activities. This testing time is called primary test hours (PTH), as it is an estimate for primary testing activities like preparation, specification, execution, etc. This is calculated as

$$PTH = TTP \times productivity\ factor \times environmental\ factor$$

Productivity factor It is an indication of the number of test hours for one test point. It is measured with factors like experience, knowledge, and skill set of the testing team. Its value ranges between 0.7 to 2.0. But explicit weightage for testing team experience, knowledge, and skills have not been considered in this metric.

Environmental factor Primary test hours also depend on many environmental factors of the project. Depending on these factors, it is calculated as

$$\begin{aligned} Environmental\ factor = {} & weights\ of\ (test\ tools + development\ testing \\ & + test\ basis + development\ environment \\ & + testing\ environment + testware)/21 \end{aligned}$$

These factors and their weights are discussed below.

Test tools It indicates the use of automated test tools in the system. Its rating is given in Table 11.10.

Table 11.10 Test tool ratings

1	Highly automated test tools are used.
2	Normal automated test tools are used.
4	No test tools are used.

Development testing It indicates the earlier efforts made on development testing before system testing or acceptance testing for which the estimate is being done. If development test has been done thoroughly, then there will be less effort and time needed for system and acceptance testing, otherwise, it will increase. Its rating is given in Table 11.11.

Table 11.11 Development testing ratings

2	Development test plan is available and test team is aware about the test cases and their results.
4	Development test plan is available.
8	No development test plan is available.

Test basis It indicates the quality of test documentation being used in the system. Its rating is given in Table 11.12.

Table 11.12 Test basis rating

3	Verification as well as validation documentation are available.
6	Validation documentation is available.
12	Documentation is not developed according to standards.

Development environment It indicates the development platforms, such as operating systems, languages, etc. for the system. Its rating is given in Table 11.13.

Table 11.13 Development environment rating

2	Development using recent platform.
4	Development using recent and old platform.
8	Development using old platform.

Test environment It indicates whether the test platform is a new one or has been used many times on the systems. Its rating is given in Table 11.14.

Table 11.14 Test environment rating

1	Test platform has been used many times.
2	Test platform is new but similar to others already in use.
4	Test platform is new.

Testware It indicates how much testware is available in the system. Its rating is given in Table 11.15.

Table 11.15 Testware rating

1	Testware is available along with detailed test cases.
2	Testware is available without test cases.
4	No testware is available.

11.9.5 Calculating Total Test Hours

Primary test hours is the estimation of primary testing activities. If we also include test planning and control activities, then we can calculate the total test hours.

Total test hours = PTH + Planning and control allowance

Planning and Control allowance is calculated based on the following factors:

Team size

3	≤ 4 team members
6	5–10 team members
12	>10 team members

Planning and control tools

2	Both planning and controlling tools are available.
4	Planning tools are available.
8	No management tools are available.

Planning and control allowance (%)
= weights of (team size + planning and control tools)

Planning and control allowance (hours)
= planning and control allowance (%) × PTH

Thus, the total test hours is the total time taken for the whole system. TTH can also be distributed for various test phases, as given in Table 11.16.

Table 11.16 TTH distribution

Testing phase	% of TTH
Plan	10%
Specification	40%
Execution	45%
Completion	5%

Example 11.4

Calculate the total test points for a module whose specifications are: functions points = 414, ratings for all FDC_w factors are normal, uniformity factor =1, rating for all QC_{dw} are 'very important' and for QC_{sw}, three static qualities are considered.

Solution

$$FDC_w = ((FI_w + UIN_w + I + C)/20) \times U = ((6+4+4+6)/20) \times 1 = 1$$

$$QC_{dw} = \sum (\text{rating of QC} / 4) \times \text{weight factor of QC}$$
$$= (5/4 \times 0.75) + (5/4 \times 0.05) + (5/4 \times 0.10) + (5/4 \times 0.10)$$
$$= 0.9375 + 0.0625 + 0.125 + 0.125 = 1.25$$

$$DTP = FP \times FDC_w \times QC_{dw}$$
$$= 414 \times 1 \times 1.25 = 517.5$$

$$STP = FP \times \sum QC_{sw} / 500$$
$$= 414 \times ((16 \times 3)/500) = 39.744$$

$$\text{Total test points (TTP)} = DTP + STP = 517.5 + 39.744 = 557.244$$

11.10 Some Testing Metrics

Everyone in the testing team wants to know when the testing should stop. To know when the testing is complete, we need to track the execution of testing. This is achieved by collecting data or metrics, showing the progress of testing. Using these progress metrics, the release date of the project can be determined. These metrics are collected iteratively during the stages of test execution cycle. Some metrics [12] are discussed below.

Test Procedure Execution Status

It is defined as:

Test Proc Exec. Status (%) = Number of executed test cases / Total number of test cases

This metric ascertains the number of percentage of test cases remaining to be executed.

Defect Aging

Defect aging is the turnaround time for a defect to be corrected. This metric is defined as

Defect aging = Closing date of bug – Start date when bug was opened

This metric data for various bugs is used for a trend analysis such that it can be known that *n* number of defects per day can be fixed by the test

team. For example, suppose 100 defects are being recorded for a project. If documented, the past experience indicates that the team can fix as many as 20 defects per day, the turnaround time for 100 problem reports may be estimated at one work-week.

Defect Fix Time to Retest

It is defined as

Defect fix time to retest
= Date of fixing the bug and releasing in new build – Date of retesting the bug

This metric provides a measure that the test team is retesting all the modifications corresponding to bugs at an adequate rate. If the rate of fixing and retesting them is not on time, then the project progress will suffer and consequently will increase the cost of the project.

Defect Trend Analysis

It is defined as the trend in the number of defects found as the testing life cycle progresses. For example,

Number of defects of each type detected in unit test per hour

Number of defects of each type detected in integration test per hour

To strengthen the defect information, defects of a type can also be classified according to their severity of impact on the system. It will be useful for testing, if we get many defects of high severity. For example,

Number of defects over severity level X (where X is an integer value for measuring the severity levels)

The trend can be such that the number of bugs increases or decreases as the system approaches its completion. Ideally, the trend should be such that the number of newly opened defects should decrease as the system testing phase ends. If the trend shows that the number of bugs are increasing with every stage of testing:

1. Previous bugs have not been closed/fixed properly.
2. Testing techniques for testing coverage is not adequate for earlier builds.
3. Inability to execute some tests until some of the defects have been fixed, allowing execution of those tests, which then find new defects.

Recurrence Ratio

This metric indicates the quality of bug-fixes. The quality of bug-fixes is good if it does not introduce any new bug in the previous working functionality of the software and the bug does not re-occur. But if the bug-fixing introduces new bugs in the already working software or the same bug re-occurs, then the quality is not good. Moreover, this metric indicates the test team the degree to

which the previously working functionality is being adversely affected by the bug-fixes. When the degree is high, the developers are informed about it. This metric is measured as

Number of bugs remaining per fix

Defect Density

This metric is defined as

$$\textit{Defect density} = \frac{\textit{Total number of defects found for a requirement}}{\textit{Number of test cases executed for that requirement}}$$

It indicates the average number of bugs found per test case in a specific functional area or requirement. If the defect density is high in a specific functionality, then there is some problem in the functionality due to the following reasons:

1. Complex functionality
2. Problem with design or implementation of that functionality
3. Inadequate resources are assigned to the functionality

This metric can also be utilized after the release of the product. The defect data should also be collected after the release of the product. For example, we can use the following metric:

Pre-ship defect density / Post-ship defect density

This metric gives the indication of how many defects remain in the software when it is released. Ideally, the number of pre-release defects found should increase and post-release defects should decrease.

Coverage Measures

Coverage goals can be planned, against which the actual ones can be measured. For white-box testing, a combination of following is recommended: degree of statement, branch, data flow, basis path, etc. coverage (planned, actual).

In this way, testers can use the following metric:

Actual degree of coverage / Planned degree of coverage

Similarly, for black-box testing, the following measures can be determined: (i) number of requirements or features to be tested, and (ii) number of equivalence classes identified.

In this way, testers can use the following metric:

Number of features or equivalence classes actually covered / Total number of features or equivalence classes.

These metrics will help in identifying the work to be done.

Tester Productivity Measures

Testers' productivity in unit time can also be measured and tracked. Managers can use these metrics for their team's productivity and build a confidence level for the on-date delivery of the product.

The following are some useful metrics in measuring the testers' productivity:

1. Time spent in test planning
2. Time spent in test case design
3. Time spent in test execution
4. Time spent in test reporting
5. Number of test cases developed
6. Number of test cases executed

Each of these metrics is planned and actual ones are measured and used for tracking the productivity.

Budget and Resource Monitoring Measures

The budget and other resources involved in a project are well-planned and must be tracked with the actual ones. Otherwise, a project may stop in between due to over-budget or fall behind its schedule.

Earned value tracking [108,109] given by Humphrey is the measure which is used to monitor the use of resources in testing. Test planners first estimate the total number of hours or cost involved in testing. Each testing activity is then assigned a value based on the estimated percentage of the total time or budget. It provides a relative value to each testing activity with respect to the entire testing effort. Then, this metric can be measured for the actual one and compared with the planned earned value.

For the planned earned values, we need the following measurement data:

1. Total estimated time or cost for overall testing effort.
2. Estimated time or cost for each testing activity.
3. Actual time or cost of each testing activity.

Then, the following metric is used to track the cost and time:

Estimated cost or time for testing activity / Actual cost or time of testing activity

Earned values are usually represented in a table form, as shown in Table 11.17.

Table 11.17 Earned values

Planned				Actual		
Testing Activity	Estimated Time (HRS)	Estimated Earned Value	Cumulative Earned Value	Date	Actual Earned value	Cumulative Earned Value

This table has two partitions: *planned values* and *actual values*. Each testing activity as well as their estimated hours for completion should be listed. The total hours for all the tasks are determined and the estimated earned value for each activity is then calculated based on their estimated percentage of the total time. This gives a relative value to each testing activity with respect to the entire testing effort. The estimated earned values are accumulated in the next column. When the testing effort is in progress, the date and actual earned value for each activity as well as their actual accumulated earned values.

Test Case Effectiveness Metric

This metric was defined by Chernak [110]. It shows how to determine whether a set of test cases is sufficiently effective in revealing defects. It is defined as

$$TCE = \frac{Number\ of\ defects\ found\ by\ the\ test\ cases}{Total\ number\ of\ defects} \times 100$$

In fact, a baseline value is selected for the TCE and is assigned for the project. When the TCE value is at or above the baseline, then the test cases have been found effective for testing. Otherwise, the test cases must be redesigned to increase the effectiveness.

SUMMARY

This chapter discusses the importance of testing effort measurements so that the largest phase of SDLC, i.e. testing, does not go beyond its schedule and budget. Therefore, testing metrics are designed to track and control all the testing activities at every phase of STLC. But before designing the testing metrics, we must know the attributes of testing to be measured, as we have seen in Chapter 10 on software measurement that attributes are necessary to design the metrics. We have recognized four major categories of attributes: progress, quality, cost, and size. All the attributes under these categories have been discussed in detail.

Various testing effort estimation models have also been discussed. These models consider various strategies to measure the effort made on testing activities. For example, Halstead model, function-point based model, test-point based model, etc. All these models provide metrics which can be used to measure the testing effort. Some of the models are very well-established in the industry. After covering the testing effort estimation model, finally, we discussed some of the well-known testing metrics.

Let us review the important concepts described in this chapter:

- Defect backlog is the number of defects that are outstanding and unresolved at one point of time with respect to the number of defects occuring.
- Defect-removal efficiency is a powerful metric for test-effectiveness, which is defined as the ratio of number of faults actually found in testing and the number of faults that could have been found in testing.

- Defect age is another metric that can be used to measure the test-effectiveness, which assigns a numerical value to the fault, depending on the phase in which it is discovered. Defect age is used in another metric called defect spoilage to measure the effectiveness of defect removal activities. Defect spoilage is calculated as

  Spoilage = Sum of (number of defects × defect age) / Total number of defects

- Spoilage is more suitable for measuring the long-term trend of test effectiveness. Generally, low values of defect spoilage mean more effective defect discovery processes.
- The relationship between code coverage and the number of test cases is described by the expression $C(x) = 1 - e^{-(p/N)*x}$ where $C(x)$ is the coverage after executing x number of test cases, N is the number of blocks in the program and p is the average number of blocks covered by a test case during the function test.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. The function point metric is used effectively for measuring the _____ of a software system.
   (a) effort
   (b) time
   (c) size
   (d) none of the above
2. As the data complexity increases, the probability of errors in module *m* _____
   (a) remains the same
   (b) increases
   (c) decreases
   (d) none of the above
3. It will be difficult to unit test a module with _____ system complexity.
   (a) higher
   (b) lower
   (c) none of the above
4. Integration and integration testing effort increases with the _____ in architectural complexity of the system.
   (a) decrease
   (b) increase
   (c) none of the above
5. Cyclomatic number should be less than or equal to _____.
   (a) 8
   (b) 9
   (c) 10
   (d) 12

6. Earned value tracking is the measure which is used to monitor the use of _____ in testing.
   (a) resources
   (b) modules
   (c) test cases
   (d) none of the above
7. Test point analysis is a technique to measure the _____ test effort estimation.
   (a) black-box
   (b) white-box
   (c) black-box and white-box
   (d) none of the above
8. Function point analysis is a technique to measure the _____ test effort estimation.
   (a) black-box
   (b) white-box
   (c) black-box and white-box
   (d) none of the above
9. Estimating an overall amount of work involved by documenting the parts of the software to be tested, is known as _____.
   (a) staff productivity
   (b) progress
   (c) scope of testing
   (d) none of the above
10. Which information is not shown on the test progress S curve?
   (a) planned number of test cases to be completed successfully by week
   (b) number of test cases attempted by week
   (c) number of test cases completed successfully by week
   (d) testing defect backlog
11. Test point analysis uses the _____ for calculating test points.
   (a) function points
   (b) use case points
   (c) number of test cases
   (d) number of testers
12. QC_{dw} is assigned the value _____ for each quality characteristic which can be tested statically using the checklist.
   (a) 13
   (b) 16
   (c) 17
   (d) 19

Review Questions

1. Consider the following program for finding the divisors of a positive integer:

```
main()
{
 int number, flag;
 int CheckPrime(int n);

 printf("Enter the value of positive integer");
 scanf("%d", &number);

 if(number == 1)
   printf("Divisor of %d is %d", number, number);
 else
 {
   flag = CheckPrime(number);
   if (flag)
   {
           printf(" %d is a prime number", number);
           printf(" Divisors are 1 and %d", number);
   }
 }
}

int CheckPrime(int n)
{
 void divisors(int n);
 int squareroot, x;

 squareroot = sqrt(n);
 x = 2;
 while(x<=squareroot)
 {
   if(number == 0)
           break;
   x++;
 }
 if(x > squareroot)
   return(1);
 else
 {
   printf("%d is a composite number", number);
   printf("Divisors are");
   divisors(number);
   return(0);
 }
}
```

```
void divisors(int n)
{
 int x, mid;
 mid = n/2;
 for(x = 1; x<=mid; x++)
 {
   if (n % d == 0)
   printf("%d", x);
 }
 printf("%d", n);
}
```

(a) Calculate the following for each module in the program:
   (i) Fan-in
   (ii) Fan-out
   (iii) Structural complexity
   (iv) Data complexity
   (v) System complexity
   (vi) IFC

(b) Which module is difficult to unit test?

2. Consider a project with the following distribution of data and calculate its defect spoilage.

SDLC Phase	No. of defects	Defect age
Requirement Specs.	34	2
HLD	25	4
LLD	17	5
Coding	10	6

3. The module implementation details of a software project are given below:

Module	Unique operators	Unique operands	Total operators	Total operands
A	23	12	43	37
B	34	12	56	34
C	12	23	54	41

Calculate Halstead effort for all the modules and the percentage of overall testing effort to be allocated to each module.

4. There are 1200 estimated function points in a project. Calculate the total number of test cases in the system and the number of test cases in acceptance testing. Also calculate the defect density (number of total defects is 236) and test case coverage.

5. Consider the table in Question 2 and calculate the defect density at each phase considering FP = 980 for the system.

6. What are the consequences if the cyclomatic complexity of a system is greater than 10? As a designer of the system, what strategy will you choose to handle it?
7. In a software project, 2050 test cases were planned, out of which 1980 were executed till date. During the execution of these test cases, bugs were found with the following details:

Bug reference ID	Bug opening date	Bug closing date
M01-01	09-10-2007	10-10-2007
M01-02	12-10-2007	12-10-2007
M01-03	11-10-2007	14-10-2007
M02-01	13-10-2007	14-10-2007
M02-02	13-10-2007	15-10-2007
M02-03	14-10-2007	16-10-2007
M02-04	15-10-2007	15-10-2007
M02-05	17-10-2007	20-10-2007
M02-06	17-10-2007	19-10-2007

Compute the following for this project:

(a) Test procedure execution status
(b) Turnaround time for a defect to be corrected
(c) Bug trend analysis for defects per day that can be fixed
(d) Defect density for requirement number M01 and M02
(e) TCE

8. What is the importance of software attributes in measurement?
9. Explain the different categories of testing attributes.
10. Visit a CMM level 5 software industry. Survey the various testing metrics being used there. List all of them and mark their importance in STLC.
11. Obtain the test plan of a project and evaluate it on the basis of rubrics discussed in this chapter.
12. What is testing defect backlog?
13. What are the problems in measuring the staff productivity attribute?
14. Define and discuss the purpose of defect removal efficiency, defect age, and defect spoilage.
15. The historical record of some similar type of projects are shown below:

Number of test procedures (NTP)	Number of person-hours consumed for testing (PH)
866	4500
870	4512
956	4578
903	4520
790	4460

If a new project of similar types is to be developed for which the number of test procedures are 1245, calculate the number of testers required if the total period of testing scheduled is 2050 hours.

16. Calculate the total test points for the functions of a software. The specifications of all the modules are given in the table below.

Function	Specification
M01	Function points = 234, Ratings for all FDC_w factors are high. Uniformity factor =1, rating for all QC_{dw} are 'important' and for QC_{sw}, two static qualities are considered.
M02	Functions points = 340, Ratings for all FDC_w factors are normal. Uniformity factor = 1, rating for all QC_{dw} are 'very important' and for QC_{sw}, three static qualities are considered.
M03	Functions points = 450, Ratings for all FDC_w factors are high. Uniformity factor = 1, rating for all QC_{dw} are 'important' and for QC_{sw}, two static qualities are considered.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter

12

Efficient Test Suite Management

OBJECTIVES

After reading this chapter, you should be able to understand:

- Why the test suite grows as the software evolves
- Why is it required to minimize the number of test cases
- Factors to be considered while minimizing a test suite
- Definition of test suite minimization problem
- Test suite prioritization as a method to reduce test cases
- Prioritization techniques

Software testing is a continuous process that takes place throughout the life cycle of a project. Test cases in an existing test suite can often be used to test a modified program. However, if the test suite is inadequate for retesting, new test cases may be developed and added to the test suite. Thus, the size of a test suite grows as the software evolves. Due to resource constraints, it is important to prioritize the execution of test cases so as to increase the chances of early detection of faults. A reduction in the size of the test suite decreases both the overhead of maintaining the test suite and the number of test cases that must be rerun after changes are made to the software. This chapter discusses the techniques of minimizing a test suite.

12.1 Why Does a Test Suite Grow?

A testing criterion is a rule or collection of rules that imposes requirements on a set of test cases. Test engineers measure the extent to which a criterion is satisfied in terms of coverage; a test set achieves 100% coverage if it completely satisfies the criterion. *Coverage* is measured in terms of the requirements that are imposed; *partial coverage* is defined as the percentage of requirements that are satisfied.

Test requirements are specific things that must be satisfied or covered; e.g. for statement coverage, each statement is a requirement; in mutation, each mutant is a requirement; and in data flow testing, each DU pair is a requirement.

Coverage criteria are used as a stopping point to decide when a program is sufficiently tested. In this case, additional tests are added until the test suite has achieved a specified coverage level according to a specific adequacy criterion. For example, to achieve statement coverage adequacy for a program, one would add additional test cases to the test suite until each statement in that program is executed by at least one test case.

Test suites can also be reused later as the software evolves. Such test suite reuse, in the form of regression testing, is pervasive in the software industry. These tests account for as much as one half of the cost of software maintenance. Running all the test cases in a test suite, however, can require a lot of effort.

There may be unnecessary test cases in the test suite including both obsolete and redundant test cases. A change in a program causes a test case to become obsolete by removing the reason for the test case's inclusion in the test suite. A test case is redundant if other test cases in the test suite provide the same coverage of the program. Thus, due to obsolete and redundant test cases, the size of a test suite continues to grow unnecessarily.

12.2 Minimizing the Test Suite and its Benefits

A test suite can sometimes grow to an extent that it is nearly impossible to execute. In this case, it becomes necessary to minimize the test cases such that they are executed for maximum coverage of the software. Following are the reasons why minimization is important:

- Release date of the product is near.
- Limited staff to execute all the test cases.
- Limited test equipments or unavailability of testing tools.

When test suites are run repeatedly for every change in the program, it is of enormous advantage to have as small a set of test cases as possible. Minimizing a test suite has the following benefits:

- Sometimes, as the test suite grows, it can become prohibitively expensive to execute on new versions of the program. These test suites will often contain test cases that are no longer needed to satisfy the coverage criteria, because they are now obsolete or redundant. Through minimization, these redundant test cases will be eliminated.
- The sizes of test sets have a direct bearing on the cost of the project. Test suite minimization techniques lower costs by reducing a test suite to a minimal subset.

- Reducing the size of a test suite decreases both the overhead of maintaining the test suite and the number of test cases that must be rerun after changes are made to the software, thereby reducing the cost of regression testing.

Thus, it is of great practical advantage to reduce the size of test cases.

12.3 Defining Test Suite Minimization Problem

Harrold *et al.* [51] have defined the problem of minimizing the test suite as given below.

Given A test suite TS; a set of test case requirements $r_1, r_2, \ldots, r_n$ that must be satisfied to provide the desired testing coverage of the program; and subsets of TS, $T_1, T_2, \ldots, T_n$, one associated with each of the r_i's such that any one of the test cases t_j belonging to T_i can be used to test r_i.

Problem Find a representative set of test cases from *TS* that satisfies all the r_i's.

The r_i's can represent either all the test case requirements of a program or those requirements related to program modifications. A representative set of test cases that satisfies the r_i's must contain at least one test case from each T_i. Such a set is called a *hitting set* of the group of sets, $T_1, T_2, \ldots T_n$. Maximum reduction is achieved by finding the smallest representative of test cases. However, this subset of the test suite is the minimum cardinality hitting set of the T_i's and the problem of finding the minimum cardinality hitting set is *NP*-complete. Thus, minimization techniques resort to heuristics.

12.4 Test Suite Prioritization

The reduction process can be best understood if the cases in a test suite are prioritized in some order. The purpose of prioritization is to reduce the set of test cases based on some rational, non-arbitrary criteria, while aiming to select the most appropriate tests. For example, the following priority categories can be determined for the test cases:

Priority 1 *The test cases must be executed*, otherwise there may be worse consequences after the release of the product. For example, if the test cases for this category are not executed, then critical bugs may appear.

Priority 2 *The test cases may be executed, if time permits.*

Priority 3 *The test case is not important prior to the current release.* It may be tested shortly after the release of the current version of the software.

Priority 4 *The test case is never important, as its impact is nearly negligible.*

In the prioritization scheme, the main guideline is to ensure that low-priority test cases do not cause any severe impact on the software. There may be several goals of prioritization. These goals can become the basis for prioritizing the test cases. Some of them are discussed here:

- Testers or customers may want to get some critical features tested and presented in the first version of the software. Thus, the important features become the criteria for prioritizing the test cases. But the consequences of not testing some low-priority features must be checked. Therefore, risk factor should be analysed for every feature in consideration.
- Prioritization can be on the basis of the functionality advertised in the market. It becomes important to test those functionalities on a priority basis, which the company has promised to its customers.
- The rate of fault detection of a test suite can reveal the likelihood of faults earlier.
- Increase the coverage of coverable code in the system under test at a faster rate, allowing a code coverage criterion to be met earlier in the test process.
- Increase the rate at which high-risk faults are detected by a test suite, thus locating such faults earlier in the testing process.
- Increase the likelihood of revealing faults related to specific code changes, earlier in the regression testing process.

12.5 Types of Test Case Prioritization

General Test Case Prioritization

In this prioritization, we prioritize the test cases that will be useful over a succession of subsequent modified versions of P, without any knowledge of the modified versions. Thus, a general test case prioritization can be performed following the release of a program version during off-peak hours, and the cost of performing the prioritization is amortized over the subsequent releases.

Version-Specific Test Case Prioritization

Here, we prioritize the test cases such that they will be useful on a specific version P' of P. Version-specific prioritization is performed after a set of changes

have been made to P and prior to regression testing of P', with the knowledge of the changes that have been made.

12.6 Prioritization Techniques

Prioritization techniques schedule the execution of test cases in an order that attempts to increase their effectiveness at meeting some performance goal. Therefore, given any prioritization goal, various prioritization techniques may be applied to a test suite with the aim of meeting that goal. For example, in an attempt to increase the rate of fault-detection of test suites, we might prioritize test cases in terms of the extent to which they execute modules that, measured historically, tend to fail. Alternatively, we might prioritize the test cases in terms of their increasing cost-per-coverage of code components, or in terms of their increasing cost-per-coverage of features listed in a requirement specification. In any case, the intent behind the choice of a prioritization technique is to increase the likelihood that the prioritized test suite can better meet the goal than an *ad hoc* or random ordering of test cases.

Prioritization can be done at two levels, as discussed below.

Prioritization for regression test suite This category prioritizes the test suite of regression testing. Since regression testing is performed whenever there is a change in the software, we need to identify the test cases corresponding to modified and affected modules.

Prioritization for system test suite This category prioritizes the test suite of system testing. Here, the consideration is not the change in the modules. The test cases of system testing are prioritized based on several criteria: risk analysis, user feedback, fault-detection rate, etc.

12.6.1 Coverage-based Test Case Prioritization

This type of prioritization [58] is based on the coverage of codes, such as statement coverage, branch coverage, etc. and the fault exposing capability of the test cases. Test cases are ordered based on their coverage. For example, count the number of statements covered by the test cases. The test case that covers the highest number of statements will be executed first. Some of the techniques are discussed below.

Total Statement Coverage Prioritization

This prioritization orders the test cases based on the total number of statements covered. It counts the number of statements covered by the test cases

and orders them in a descending order. If multiple test cases cover the same number of statements, then a random order may be used.

For example, if T_1 covers 5 statements, T_2 covers 3, and T_3 covers 12 statements; then according to this prioritization, the order will be T_3, T_1, T_2.

Additional Statement Coverage Prioritization

Total statement coverage prioritization schedules the test cases based on the total statements covered. However, it will be useful if it can execute those statements as well that have not been covered yet. Additional statement coverage prioritization iteratively selects a test case T_1, that yields the greatest statement coverage, then selects a test case which covers a statement uncovered by T_1. Repeat this process until all statements covered by at least one test case have been covered.

For example, if we consider Table 12.1, according to total statement coverage criteria, the order is 2,1,3. But additional statement coverage selects test case 2 first and next, it selects test case 3, as it covers statement 4 which has not been covered by test case 2. Thus, the order according to addition coverage criteria is 2,3,1.

Table 12.1 Statement coverage

Statement	Statement Coverage		
	Test case 1	Test case 2	Test case 3
1	X	X	X
2	X	X	X
3		X	X
4			X
5			
6		X	
7	X	X	
8	X	X	
9	X	X	

Total Branch Coverage Prioritization

In this prioritization, the criterion to order is to consider condition branches in a program instead of statements. Thus, it is the coverage of each possible outcome of a condition in a predicate. The test case which will cover maximum branch outcomes will be ordered first. For example, see Table 12.2. Here, the order will be 1, 2, 3.

Table 12.2 Branch coverage

Branch Statements	Branch Coverage		
	Test case 1	Test case 2	Test case 3
Entry to while	X	X	X
2-true	X	X	X
2-false	X		
3-true		X	
3-false	X		

Additional Branch Coverage Prioritization

Here, the idea is the same as in additional statement coverage of first selecting the test case with the maximum coverage of branch outcomes and then, selecting the test case which covers the branch outcome not covered by the previous one.

Total Fault-Exposing-Potential (FEP) Prioritization

Statement and branch coverage prioritization ignore a fact about test cases and faults:

- Some bugs/faults are more easily uncovered than other faults.
- Some test cases have the proficiency to uncover particular bugs as compared to other test cases.

Thus, the ability of a test case to expose a fault is called the *fault exposing potential*. It depends not only on whether test cases cover a faulty statement, but also on the probability that a fault in that statement will also cause a failure for that test case.

To obtain an approximation of the FEP of a test case, an approach was adopted using mutation analysis [137,138]. This approach is discussed below.

Given a program P and a test suite T,

- First, create a set of mutants $N = \{n_1, n_2, \ldots n_m\}$ for P, noting which statement s_j in P contains each mutant.
- Next, for each test case $t_i T$, execute each mutant version n_k of P on t_i, noting whether t_i kills that mutant.
- Having collected this information for every test case and mutant, consider each test case t_i, and each statement s_j in P, and calculate $FEP(s,t)$ of t_i on s_j as the ratio:

  Mutants of s_j killed / Total number of mutants of s_j

  If t_i does not execute s_j, this ratio is zero.

To perform total FEP prioritization, given these (FEP)(s,t) values, calculate an *award value* for each test case t_iT, by summing the (FEP)(s_j, t_i) values for all statements s_j in P. Given these award values, we prioritize the test cases by sorting them in order of descending award value.

Table 12.3 shows FEP estimates of a program. Total FEP prioritization outputs the test case order as (2, 3, 1).

Table 12.3 FEP estimates

Statement	FEP(s,t) values		
	Test case 1	Test case 2	Test case 3
1	0.5	0.5	0.3
2	0.4	0.5	0.4
3		0.01	0.4
4		1.3	
5			
6	0.3		
7	0.6		0.1
8		0.8	0.2
9			0.6
Award values	1.8	3.11	2.0

12.6.2 Risk-based Prioritization

Risk-based prioritization [22] is a well-defined process that prioritizes modules for testing. It uses risk analysis to highlight potential problem areas, whose failures have adverse consequences. The testers use this risk analysis to select the most crucial tests. Thus, risk-based technique is to prioritize the test cases based on some potential problems which may occur during the project.

- **Probability of occurrence/fault likelihood** It indicates the probability of occurrence of a problem.
- **Severity of impact/failure impact** If the problem has occurred, how much impact does it have on the software.

  Risk analysis uses these two components by first listing the potential problems and then, assigning a probability and severity value for each identified problem, as shown in Table 12.4. By ranking the results in this table in the form of risk exposure, testers can identify the potential

problems against which the software needs to be tested and executed first. For example, the problems in the given table can be prioritized in the order of P_5, P_4, P_2, P_3, P_1.

A risk analysis table consists of the following columns:

- **Problem ID** A unique identifier to facilitate referring to a risk factor.
- **Potential problem** Brief description of the problem.
- **Uncertainty factor** It is the probability of occurrence of the problem. Probability values are on a scale of 1 (low) to 10 (high).
- **Severity of impact** Severity values on a scale of 1 (low) to 10 (high).
- **Risk exposure** Product of probability of occurrence and severity of impact.

Table 12.4 Risk analysis table

Problem ID	Potential Problem	Uncertainty Factor	Risk Impact	Risk Exposure
P_1	Specification ambiguity	2	3	6
P_2	Interface problems	5	6	30
P_3	File corruption	6	4	24
P_4	Databases not synchronized	8	7	56
P_5	Unavailability of modules for integration	9	10	90

12.6.3 Prioritization Based on Operational Profiles

This is not a prioritization in true sense, but the system is developed in such a way that only useful test cases are designed. So there is no question of prioritization. In this approach, the test planning is done based on the operation profiles [128, 129] of the important functions which are of use to the customer. An operational profile is a set of tasks performed by the system and their probabilities of occurrence. After estimating the operational profiles, testers decide the total number of test cases, keeping in view the costs and resource constraints.

12.6.4 Prioritization using Relevant Slices

During regression testing, the modified program is executed on all existing regression test cases to check that it still works the same way as the original

program, except where a change is expected. But re-running the test suite for every change in the software makes regression testing a time-consuming process. If we can find the portion of the software which has been affected with the change in software, then we can prioritize the test cases based on this information. This is called the *slicing technique* [130,131,132]. The various definitions related to this technique have been defined in literature. Let us discuss them.

Execution Slice

The set of statements executed under a test case is called the *execution slice* of the program. For example, see the following program (Fig. 12.1):

```
Begin
S1: read (basic, empid);
S2: gross = 0;
S3: if (basic > 5000 || empid > 0)
         {
S4:       da = (basic*30)/100;
S5:       gross = basic + da;
         }
S6:  else
         {
S7:       da = (basic*15)/100;
S8:       gross = basic + da;
          }
S9: print (gross, empid);
End
```

Figure 12.1 Example program for execution slice

Table 12.5 shows the test cases for the given program.

Table 12.5 Test cases

Test Case	Basic	Empid	Gross	Empid
T1	8000	100	10400	100
T2	2000	20	2300	20
T3	10000	0	13000	0

$T1$ and $T2$ produce correct results. On the other hand, $T3$ produces an incorrect result. Syntactically, it is correct, but an employee with the empid '0' will not get any salary, even if his basic salary is read as input. So it has to be modified.

Suppose $S3$ is modified as [if (basic>5000 && empid>0)]. It means for $T1$, $T2$, and $T3$, the program would be rerun to validate whether the change in $S3$ has introduced new bugs or not. But if there is a change in $S7$ statement [da = (basic*25)/100; instead of da = (basic*15)/100;], then only $T2$ will be rerun. So in the execution slice, we will have less number of statements. The execution slice is highlighted in the given code segment.

```
Begin
   S1: read (basic, empid);
   S2: gross=0;
   S3: if(basic > 5000 || empid > 0)
          {
   S4:    da = (basic*30)/100;
   S5:    gross = basic + da;
          }
   S6: else
          {
   S7:    da = (basic*15)/100;
   S8:    gross = basic + da;
          }
   S9: print(gross, empid);
End
```

Instead of computing an execution slice at the statement level, we may also compute it at the function or the module level. That is, instead of determining which program statements are executed under each test case, we may simply determine which functions or modules are invoked under each test case. Then, we need to execute the new program on a test case only if a modified function or module was invoked during the original program's execution on that test case.

Dynamic Slice

The set of statements executed under a test case having an effect on the program output is called the *dynamic slice* of the program with respect to the output variables. For example, see the following program (Fig. 12.2).

```
Begin
   S1: read (a,b);
   S2: sum=0;
   S2.1: I=0;
   S3: if (a==0)
```

```
            {
   S4:         print(b);
   S5:         sum+=b;
            }
   S6: else if(b==0)
            {
   S7:         print(a);
   S8:         sum+=a;
            }
   S9: else
            {
   S10:        sum=a+b+sum;
   S10.1       I=25;
   S10.2       print(I);
            }
   S11:endif
   S12:print(sum);
End
```

Figure 12.2 Example program for dynamic slice

Table 12.6 shows the test cases for the given program.

Table 12.6 Test cases

Test Case	a	b	sum
T1	0	4	4
T2	67	0	67
T3	23	23	46

$T1$, $T2$, and $T3$ will run correctly but if some modification is done in $S10.1$ [say I = 50], then this change will not affect the output variable. Therefore, there is no need to rerun any of the test cases. On the other hand, if $S10$ is changed [say, sum=a*b+sum], then this change will affect the output variable 'sum', so there is a need to rerun $T3$. The dynamic slice is highlighted in the code segment.

```
Begin
   S1:     read (a,b);
   S2:     sum = 0;
   S2.1:   I=0;
   S3:     if (a==0)
           {
   S4:     print(b);
   S5:     sum+=b;
           }
```

```
  S6: elseif(b==0)
        {
  S7:     print(a);
  S8:     sum+=a;
            }
  S9: else
        {
  S10:    sum=a+b+sum;
  S10.1   I=25;
  S10.2   print(I);
        }
  S11: endif
  S12: print(sum);
End
```

Relevant Slice

The set of statements that were executed under a test case and did not affect the output, but have the potential to affect the output produced by a test case, is known as the *relevant slice* of the program. It contains the dynamic slice and in addition, includes those statements which, if corrected, may modify the variables at which the program failure has manifested. For example, consider the example of Fig. 12.2, statements $S3$ and $S6$ have the potential to affect the output, if modified.

If there is a change in any statement in the relevant slice, we need to rerun the modified software on only those test cases whose relevant slices contain a modified statement. Thus, on the basis of relevant slices, we can prioritize the test cases. This technique is helpful for prioritizing the regression test suite which saves time and effort for regression testing.

Jeffrey and Gupta [139] enhanced the approach of relevant slicing and stated: 'If a modification in the program has to affect the output of a test case in the regression test suite, it must affect some computation in the relevant slice of the output for that test case'. Thus, they applied the heuristic for prioritizing test cases such that the test case with larger number of statements must get higher weight and will get priority for execution.

12.6.5 Prioritization Based on Requirements

This technique is used for prioritizing the system test cases. The system test cases also become too large in number, as this testing is performed on many grounds. Since system test cases are largely dependent on the requirements, the requirements can be analysed to prioritize the test cases.

This technique does not consider all the requirements on the same level. Some requirements are more important as compared to others. Thus, the test cases corresponding to important and critical requirements are given more weight as compared to others, and these test cases having more weight are executed earlier.

Hema Srikanth *et al.* [136] have applied requirement engineering approach for prioritizing the system test cases. It is known as PORT (prioritization of requirements for test). They have considered the following four factors for analysing and measuring the criticality of requirements:

Customer-assigned priority of requirements Based on priority, the customer assigns a weight (on a scale of 1 to 10) to each requirement. Higher the number, higher is the priority of the requirement.

Requirement volatility This is a rating based on the frequency of change of a requirement. The requirement with a higher change frequency is assigned a higher weight as compared to the stable requirements.

Developer-perceived implementation complexity All the requirements are not equal on a implementation level. The developer gives more weight to a requirement which he thinks is more difficult to implement.

Fault proneness of requirements This factor is identified based on the previous versions of system. If a requirement in an earlier version of the system has more bugs, i.e. it is error-prone, then this requirement in the current version is given more weight. This factor cannot be considered for a new software.

Based on these four factor values, a *prioritization factor value* (PFV) is computed as given below. PFV is then used to produce a prioritized list of system test cases.

$$\mathbf{PFV}_i = \sum(\mathbf{FV}_{ij} \times \mathbf{FW}_j)$$

where FV_{ij} = Factor value is the value of factor j corresponding to requirement i

FW_j = Factor weight is the weight given to factor j

12.7 Measuring the Effectiveness of a Prioritized Test Suite

When a prioritized test suite is prepared, how will we check its effectiveness? We need one metric which can tell us the effectiveness of one prioritized test suite. For this purpose, the rate of fault-detection criterion can be taken. Elbaum *et al.* [133,134] developed APFD (average percentage of faults detected) metric that measures the weighted average of the percentage of faults detected

during the execution of a test suite. Its value ranges from 0 to 100, where a higher value means a faster fault-detection rate. Thus, APFD is a metric to detect how quickly a test suite identifies the faults. If we plot percentage of test suite run on the x-axis and percentage of faults detected on the y-axis, then the area under the curve is the APFD value, as shown in Fig. 12.3.

Figure 12.3 Calculating APFD

APFD is calculated as given below.

$$\text{APFD} = 1 - ((\text{TF}_1 + \text{TF}_2 + \ldots\ldots\ldots + \text{TF}_m)/nm) + 1/2n$$

where TF_i is the position of the first test in test suite T that exposes fault i
m is the total number of faults exposed in the system or module under T
n is the total number of test cases in T

Example 12.1

Consider a program with 10 faults and a test suite of 10 test cases, as shown in Table 12.7.

Table 12.7 Test case fault exposure

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
F1					X			X		
F2		X	X	X		X				
F3	X	X	X	X			X	X		
F4						X				X
F5	X		X		X	X		X		X
F6					X	X	X		X	
F7									X	
F8		X		X		X			X	X
F9	X									X
F10			X	X					X	X

If we consider the order of test suite as (T1, T2, T3, T4, T5, T6, T7, T8, T9, T10), then calculate the APFD for this program.

Solution

$$\text{APFD} = 1 - ((5 + 2 + 1 + 6 + 1 + 5 + 9 + 2 + 1 + 3)/10*10) + \frac{1}{2 \times 10}$$

$$= 0.65 + 0.05$$
$$= 0.7$$

All the bugs detected are not of the same level of severity. One bug may be more critical as compared to others. Moreover, the cost of executing the test cases also differs. One test case may take more time as compared to others. Thus, APFD does not consider the severity level of the bugs and the cost of executing the test cases in a test suite. Elbaum *et al.* [135] modified their APFD metric and considered these two factors to form a new metric which is known as *cost-cognizant* APFD and denoted as APFD_c. In APFD_c, the total cost incurred in all the test cases is represented on x-axis and the total fault severity detected is taken on y-axis. Thus, it measures the unit of fault severity detected per unit test case cost.

SUMMARY

The size of a test suite increases as the software evolves. We have also seen this in regression testing. This chapter separately deals with this problem because test suite size is a general problem, not restricted to regression testing. In this chapter, we discuss the benefits of reducing the number of test cases along with the general definition of test suite minimization problem.

Test case prioritization technique has become very popular and effective today in minimizing the test suite. Prioritization can be done at various levels like statement coverage, branch coverage, or fault exposing potential. The prioritization technique and its various methods have also been discussed here.

Let us review the important concepts described in this chapter:

- The purpose of prioritization is to reduce the set of test cases based on some rational, non-arbitrary criteria, while aiming to select the most appropriate tests.
- General test case prioritization technique prioritizes the test cases that will be useful over a succession of subsequent modified versions of *P,* without any knowledge of modified versions.
- Version-specific test case prioritization prioritizes the test cases such that they will be useful on a specific version *P'* of *P*.
- Risk-based prioritization technique is to prioritize the test cases based on some potential problems which may occur during the project.
- The set of statements executed under a test case is called the execution slice of the program.

- The set of statements executed under a test case having an effect on the program output is called the dynamic slice of the program with respect to the output variables.
- The set of statements that were executed under a test case and did not affect the output, but have the potential to affect the output produced by a test case is known as the relevant slice of a program.
- Prioritization factor value (PFV) is used to produce a prioritized list of system test cases as given below:

$$PFV_i = \sum (FV_{ij} \times FW_j)$$

where FV = Factor value is the value of factor *j* corresponding to requirement *i*

FW = Factor weight is the weight given to factor *j*

- APFD (average percentage of faults detected) is the metric that measures the weighted average of the percentage of faults detected during the execution of a test suite. Its value ranges from 0 to 100, where a higher value means a faster fault-detection rate.

$$APFD = 1 - ((TF_1 + TF_2 + \ldots\ldots\ldots + TF_m) / nm) + 1/2n$$

where TF_i is the position of the first test in test suite *T* that exposes fault *i*

m is the total number of faults exposed in the system or module under *T*

n is the total number of test cases in *T*

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. If the test suite is inadequate for retesting, then _____.
   (a) new test cases may be developed and added to the test suite
   (b) existing test suite should be modified accordingly
   (c) old test suite should be discarded and altogether new test suite should be developed
   (d) none of the above
2. The size of a test suite _____ as the software evolves.
   (a) decreases
   (b) increases
   (c) remains same
   (d) none of the above
3. Coverage is measured in terms of the _____ that are imposed.
   (a) requirements
   (b) design
   (c) test cases
   (d) none of the above

4. In the prioritization scheme, the main guideline is to ensure that _____ priority test cases do not cause any severe impacts on the software.
   (a) high
   (b) low
   (c) medium
   (d) none of the above
5. Automatic test generation often results in _____ test sets.
   (a) larger
   (b) smaller
   (c) medium size
   (d) none of the above
6. The set of statements executed under a test case, having an effect on the program output under that test case is called _____.
   (a) execution slice
   (b) dynamic slice
   (c) relevant slice
   (d) none of the above
7. The set of statements executed under a test case is called _____.
   (a) execution slice
   (b) dynamic slice
   (c) relevant slice
   (d) none of the above
8. The set of statements that were executed under a test case and did not affect the output, but have the potential to affect the output produced by a test case is known as _____.
   (a) execution slice
   (b) dynamic slice
   (c) relevant slice
   (d) none of the above
9. Which one is true?
   (a) $APFD = 1 + ((TF_1+TF_2+\ldots\ldots\ldots+ TF_m) / nm) + 1/2n$
   (b) $APFD = 1- ((TF_1+TF_2+\ldots\ldots\ldots+ TF_m) / nm) + 1/3n$
   (c) $APFD = 1- ((TF_1+TF_2+\ldots\ldots\ldots+ TF_m) / nm) + 1/2n$
   (d) none of the above

Review Questions

1. What is the need for minimizing the test cases in a project?
2. Develop a priority category scheme for the test cases to be executed in a project that deals with all kinds of priorities set in that project.

3. Identify some potential problems in a project. Mark them on a scale of 1 to 10 for uncertainty factor and risk impact. Prepare its risk table.
4. Explain the following with example:
   (a) Total statement coverage prioritization
   (b) Additional statement coverage prioritization
   (c) Total branch coverage prioritization
   (d) Additional branch coverage prioritization
   (e) Total fault-exposing-potential (FEP) prioritization
5. Take a module from a project and its test cases. Identify execution, dynamic, and relevant slices in this module.
6. PORT depends on four factors. Are these factors sufficient for prioritization? If no, identify some more factors.
7. What is the use of PFV?
8. Search the literature on prioritization, based on operational profiles, and implement this technique in a project.
9. What is the difference between APFD and $APFD_c$?
10. Consider Example 12.1 and calculate its APFD with the following order of test suite:
    (a) (T2,T1,T3,T4,T5,T6,T7,T8,T9,T10)
    (b) (T1,T2,T5,T4,T3,T6,T9,T8,T7,T10)

    What is the effect of changing the order of test cases?

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Part 4 Quality Management

CHAPTERS

Chapter 13:
Software Quality Management

Chapter 14:
Testing Process Maturity Models

One of the major testing goals is to ensure software quality. We define testing as a critical element of software quality. This is where the story of software testing begins. We have mentioned this in Part I. In Part IV, we discuss in detail the factors that contribute to software quality, the role of testing in quality, and how to manage quality.

As software testing aims to get better quality software, the quality of the testing process also needs to be measured. There have been various quality models for SDLC, but none of them take the testing process into consideration. Therefore, testing process should also be measured for its maturity level with new models. The various models for testing the process maturity have been discussed in this part.

This part will make ground for the following concepts:

- Quality control
- Quality assurance
- Quality management
- Quality factors
- Methods of quality management
- Test process maturity models

Chapter

13

Software Quality Management

We use the term *quality* very often in our daily life. We expect a certain level of quality from a product. But how do we define quality? Crosby [39] defines quality as *conformance to requirements*. This means that the product should be developed according to the pre-specified requirements. If the requirements are misunderstood and the product is developed with incorrect requirements, then the product lacks quality.

Juran & Gryna [40] defines quality as *fitness for use*. It means whether the product is actually usable by the user or not. If the user is satisfied with the product, it is of good quality.

This shows that quality is not a single idea; rather it is a multi-dimensional concept, as shown in Fig. 13.1.

On the basis of the multi-dimensional concept, quality can be defined as 'the degree to which a product or service possesses a desired combination of attributes'.

OBJECTIVES

After reading this chapter, you should be able to understand:

- Software Quality is a multi-dimensional concept
- Difference between product quality and process quality
- Costs incurred to maintain quality
- Benefits of investment on quality
- Difference between quality control, quality assurance, and quality management
- Difference between quality management and project management
- Various factors of quality
- Types of quality management: procedural and quantitative
- Various quality metrics

Figure 13.1 Quality as multi-dimensional concept

Similarly, Pressman [7] defines quality as *an attribute of an item; quality refers to measurable characteristics—things we are able to compare to known standards such as length, colour, electrical properties, etc.*

13.1 Software Quality

We know that software is not similar to any physical product, so the concept of software quality also differs from physical products. Software quality is not easily definable. Software has many possible quality characteristics. In the narrowest sense, software quality revolves around defects. It is commonly recognized as lack of bugs in the software. This definition is the same as the classical definition of quality–'conformance to requirements'. If a software has functional bugs, then it may be possible that the software is not able to perform the functions. Thus, software quality may be defined in the form of delivered defect density or defect rate, i.e. the number of defects per unit size (e.g. LOC, FP, or other unit).

However, as discussed above, software quality is also a multi-dimensional concept rather than a single concept. In a software project, every member of the project team is responsible for the quality of the product. In fact, quality, along with cost and schedule is an important activity in a software project, which concerns everyone in the team. The ultimate goal is that the final product should have as few defects as possible, resulting in high quality software.

13.2 Broadening the Concept of Quality

Till now, software has been considered a product and in quality sense, we expect all its attributes to improve. But the definition of quality is not complete at this point. The concept of quality can be extended to the process to be adopted for development of the product. In this sense, there are two views of quality.

Product Quality

Quality measures are taken to improve the end-product.

Process Quality

From customers' requirements to the delivery of end-product, the development process to be adopted for software development is complex. If the quality of development process is not good enough, then it will certainly affect the quality of the end-product. If each stage of the development process is developed according to the requirement of its intermediate user, then the final end-product will be of high quality. Therefore, the quality of development process directly affects the quality of delivered product, as shown in Fig. 13.2.

Figure 13.2 Process affects quality

From the customer's point of view, product quality is according to the customer, while process quality is according to the intermediate user of the next stage. For example, the design person will use SRS. But all the intermediate stages of development process are being developed for a high-quality end-product.

Process quality is particularly useful in software development, as it is difficult to measure software attributes, such as maintainability, without using the software for a long period. Quality improvement focuses on identifying good quality products and examining the processes so that they may be applied across a range of projects.

To improve the quality of processes and products, the following activities must be carried out:

- Choose the development process best suited for the project.
- Select and replay specific methods and approaches within the selected process.
- Adopt proper tools and technologies.
- Prepare quality metrics for products based on their desired attributes and implement software metrics program to ensure that the development process is under control and making progress.

13.3 Quality Cost

Quality of a software is maintained with efforts in both products and processes, as shown above. If the efforts in the form of quality evaluation programs are not implemented, then the desired-level quality cannot be achieved. According to John Ruskin, *quality is never by an accident, it is always the result of an intelligent effort.*

But implementation of those quality-related procedures and practices is expensive and incurs cost. According to Pressman and Humphrey [7,41], the cost of quality is decomposed into three major categories:

- **Prevention costs** are related with activities that identify the cause of defects and those actions that are taken to prevent them, e.g. quality planning, formal technical reviews, test and laboratory equipments, training, defect causal analysis and prevention.
- **Appraisal costs** include the costs of evaluating the quality of software products at some level, e.g. testing, in-process and inter-process inspection by implementing software metrics programs (SMP), equipment calibrations, and maintenance.
- **Failure costs** include the costs to analyse and remove the failures. Since failures can occur at the development site as well as at the customer's site, failure costs can be divided into the following parts:
  - *External failure costs* are associated with failures that appear at the customer's site when the product has been released, e.g. regression testing corresponding to any bug, listening and resolving the customers' problems in case of any complaint from the customer, product return and replacement, etc.
  - *Internal failure costs* are those costs that appear on the developer's site prior to the release of the product, e.g. regression testing, isolating the bug and repairing, analysing the criticality of bug, etc.

As discussed earlier, the cost of finding and debugging a bug increases tenfold as we move further in the development stages of SDLC. Similarly, if we are not implementing prevention and appraisal activities in the early stages, then a high cost of failure detection and debugging has to be paid for internal and external failures.

13.4 Benefits of Investment on Quality

Despite the costs incurred in prevention and appraisal activities, benefits of these activities greatly outweigh the costs. If we invest in prevention and appraisal activities at an early stage, then failures and failure-costs reduce (which are high if prevention and appraisal are ignored), as shown in Fig. 13.3.

Figure 13.3 Benefits of investment on quality

Thus, by implementing quality evaluation programs, the following benefits are achieved:

- Customer is satisfied, as the end-product is of high quality.
- Productivity increases due to shorter SDLC cycle.
- Failures and failure costs reduce.
- Rework and cost of quality reduce.

13.5 Quality Control and Quality Assurance

It is clear that for a high-quality software, the removal of defects is necessary. This removal of defects takes place when quality control activities are adopted. Quality control involves a series of static and dynamic testing throughout the process, such that all the stages produce outputs according to the desired requirements.

The term 'quality control' was used in 1960s, basically for manufacturing products. The definition of quality control had a broad view at that time including the measurement of processes. In 1980s, the term 'quality assurance' became popular. Both these terms appear to be synonymous and confusing. But Pressman and Pankaj Jalote help to clarify the difference between the two.

According to Pankaj Jalote [3], *quality control focuses on finding and removing defects, whereas the main purpose of quality assurance is to verify that applicable procedures and standards are being followed.*

According to Pressman [7], *a key concept of quality control is that all work products have defined, measurable specifications to which we may compare the output of each process.*

Quality assurance consists of auditing and reporting functions of management. Based on the above definitions, we can define the two terms again in reference to software.

Quality control is basically related to software product such that there is minimum variation, according to the desired specifications. This variation is checked at each step of development. Quality control may include the following activities: reviews, testing using manual techniques or with automated tools (V&V).

Quality assurance is largely related to the process. In addition, quality assurance activities are in the management zone. Therefore, auditing and reporting of quality, based on quantitative measurements, are also performed. The goal is to inform the management about quality-related issues and supply relevant data. It is the management's responsibility to address these problems. Thus, software quality assurance (SQA) is defined as a planned and systematic approach to evaluate quality and maintain software product standards, processes, and procedures.

SQA includes the process of assuring that standards and procedures are established and are followed throughout the software acquisition life cycle. Compliance with agreed-upon standards and procedures is evaluated through process monitoring, product evaluation, and audits. Software development and control processes should include quality assurance approval points, where an SQA evaluation of the product may be done in relation to the applicable standards.

Quality assurance may include the following activities:

- Measuring the products and processes.
- Auditing the quality of every step in SDLC and the end-product.
- Verifying that the adopted procedures and standards are being followed.

13.6 Quality Management (QM)

We have discussed that quality assurance is a management activity performed by managers. At the managerial level, quality needs to be planned on a larger scale and all controlling and assurance activities are performed according to quality plans. The managerial root has given the term 'quality management' whose scope is larger than quality control and quality assurance. It is now an established way of managing the quality of a product.

The task of quality management is to

- plan suitable quality control and quality assurance activities.
- define procedures and standards which should be used during software development and verify that these are being followed by everyone.
- properly execute and control activities.

If quality at some point is not under control, then it is the responsibility of quality managers to manage the resources such that the required level of quality is achieved.

But there is more to quality management. The major role of QM is to develop a *quality culture* in the organization. Quality culture means that every member of the team is aware and conscious about the quality and is working towards a high-quality end-product. It is the responsibility of quality managers to encourage the team members to take responsibility for their work, not only to function correctly but also for improving the quality (in the sense of defects). Though QM specifies the quality specifications and standards, yet there are some aspects which cannot be standardized, e.g. elegance, readability, etc. are intangible aspects of software quality. Quality managers also encourage the team members to keep an eye on these intagible aspects of quality. Thus, every team member works in unison to achieve the common goal of high-quality software.

QM involves verifying and improving the software process. The quality of intermediate results at each stage and the final end-product is measured, reviewed, and analysed. If the target does not seem to be achievable with the current process, then problems and difficulties are reported to senior management, and finally the current process can be improved based on this feedback.

13.7 QM and Project Management

QM is different from project management (PM), in the sense that QM is not associated with a particular project, rather it is an organization-wide activity for many projects. On the other hand, PM is specific to a particular project and manages the cost schedule, resources, and quality of that project only.

13.8 Quality Factors

We discussed earlier that quality is not a single idea; rather it is a combination of many factors. But these factors are based on the project and the environment in which the software is to be developed. Discussed below are some of these factors.

Functionality The major visible factor for software quality is its functionality such that all the intended features are working.

Correctness Software should be correct in nature largely satisfying specifications and fulfilling user's objectives. Another measure of correctness may be the extent to which a software is fault-free.

Completeness A software product should be complete to the extent that all its parts are present and fully developed. This means that if the code calls a subroutine from an external library, the software package must provide reference to that library and all the required parameters must be passed. All the required input data must be available.

Efficiency The software should be efficient in its working, i.e. utilizing all resources.

Portability A software product should be operated on all well-known platforms and configurations.

Testability A software should be testable such that it is easy to debug and maintain.

Usability The software should be easy to use, operate, and understand in every manner that its user wants. The usability criterion may differ from project to project.

Reliability It is the ultimate factor that a user demands from any product. A software product should also be reliable in the sense that all its desired features must run forever without bugs; it should not stop after working for some years. This implies a time factor in which a reliable product is expected to perform correctly over a period of time. It also encompasses environmental considerations in which the product is required to perform correctly in whichever condition it finds itself—this is sometimes called *robustness*. The reliability factor is more in demand in case of real-time projects, where the system must work continuously without fail.

Reusability It is an important factor, as it can greatly reduce the effort required to engineer a software product. However, it is not easy to achieve.

Integrity It defines the extent to which an unauthorized access or a modification can be controlled in a computer system (either manually or with automated tools).

Maintainability The software should have the scope of maintaining it, if a bug appears or a new requirement needs to be incorporated.

13.9 Methods of Quality Management

We have discussed the meaning of quality management and the activities which are involved in it. There are largely two approaches to perform QM (see Fig. 13.4), which are discussed here.

13.9.1 Procedural Approach to QM

In this approach, quality is controlled by detecting defects by performing reviews, walkthroughs, inspections (static testing), and validation testing methods. For implementing this approach, the plan and procedures are defined for conducting reviews and testing. As discussed earlier that every testing plan is prepared at a certain point and according to the defined procedure in test documents, testing activities are executed again on defined points. For example, acceptance test plan is prepared at the time of requirement gathering and acceptance testing is performed at the time of validation testing. Moreover, verification is conducted after every step of SDLC.

Execution of all V&V activities are performed by software engineers with the software quality assurance (SQA) team, who identify the defects. In this way, quality is being controlled by identifying and removing defects in the software.

Figure 13.4 Methods of quality management

Software Quality Assurance Activities

SQA activities include *product evaluation* and *process monitoring*. Products are monitored for conformance to standards and processes are monitored for conformance to procedures. Audits are a key technique used to perform product evaluation and process monitoring.

Product evaluation assures that standards are being followed. SQA team assures that clear and achievable standards exist and then evaluates the software with the established standards. Product evaluation assures that the software product reflects the requirements of the applicable standard(s). Process monitoring ensures that appropriate steps to carry out the process are being

followed. SQA team monitors processes by comparing the actual steps carried out with those in the documented procedures.

The audit technique used by the SQA team looks at a process and/or a product in depth, comparing them with the established procedures and standards. Audits are used to review management, technical, and assurance processes to provide an indication of the quality and status of the software product. The purpose of an SQA audit is to assure that proper control procedures are being followed, that required documentation is maintained, and that the developer's status reports accurately reflect the status of an activity. After the audit, an audit report is prepared that consists of findings and recommendations to bring the development into conformance with the standards and/or procedures.

SQA Relationships to Other Assurance Activities

Some of the more important relationships of SQA to other management and assurance activities are described below.

Configuration management monitoring SQA assures that software configuration management (CM) activities are performed in accordance with the CM plans, standards, and procedures. It reviews the CM plans for compliance with the software CM policies and requirements, and provides follow-up for nonconformances. It also audits the CM functions for adherence to standards and procedures and prepares a report of its findings.

The CM activities monitored and audited by SQA include baseline control, configuration identification, configuration control, configuration status accounting, and configuration authentication. SQA also monitors and audits the software library. It assures that:

- Baselines are established and consistently maintained for use in subsequent baseline development and control.
- Software configuration identification is consistent and accurate with respect to the numbering or naming of computer programs, software modules, software units, and associated software documents.
- Configuration control is maintained such that the software configuration used in critical phases of testing, acceptance, and delivery is compatible with the associated documentation.
- Configuration status accounting is performed accurately, including the recording and reporting of data, reflecting the software's configuration identification, proposed changes to the configuration identification, and the implementation status of approved changes.

- Software configuration authentication is established by a series of configuration reviews and audits that exhibit the performance required by the SRS and the configuration of the software is accurately reflected in the software design documents.
- Software development libraries provide for proper handling of software code, documentation, media, and related data in their various forms and versions from the time of their initial approval or acceptance until they have been incorporated into the final media.
- Approved changes to baselined software are made properly and consistently in all products, and no unauthorized changes are ma

Verification and validation monitoring SQA assures verification and validation (V&V) activities by monitoring technical reviews, inspections, and walkthroughs. The SQA team's role in reviews, inspections, and walkthroughs is to observe, participate as needed, and verify that they were properly conducted and documented. SQA also ensures that any actions required are assigned, documented, scheduled, and updated.

Formal test monitoring SQA assures that formal software testings like acceptance testing, is done in accordance with plans and procedures. SQA reviews testing documentation for completeness and adherence to standards. The documentation review includes test plans, test specifications, test procedures, and test reports. SQA monitors testing and provides follow-up on non-conformances. By test monitoring, SQA assures software completeness and readiness for delivery.

The objectives of SQA in monitoring formal software testing are to assure that:

- The test procedures are testing the software requirements in accordance with the test plans.
- The test procedures are verifiable.
- The correct or 'advertised' version of the software is being tested (by SQA monitoring of the CM activity).
- Non-conformances occurring during testing (that is, any incident not expected in the test procedures) are noted and recorded.
- Test reports are accurate and complete.
- Regression testing is conducted to assure that all the non-conformances have been corrected.
- Resolution of all non-conformances takes place prior to the delivery of product.

Software Quality Assurance during SDLC

In addition to the general activities described above, there are phase-specific SQA activities that should be conducted during SDLC. The activities for each phase are described here.

Concept and initiation phase SQA should be involved in both writing and reviewing the management plan in order to assure that the processes, procedures, and standards identified in the plan are appropriate, clear, specific, and auditable.

Requirements phase During this phase, SQA assures that software requirements are complete, testable, and properly expressed as functional, performance, and interface requirements.

Architectural design phase SQA activities during the architectural (preliminary) design phase include:

- Assuring adherence to approved design standards as designated in the management plan.
- Assuring all software requirements are allocated to software components.
- Assuring that a testing verification matrix exists and is kept up-to-date.
- Assuring the interface control documents are in agreement with the standard, in form and content.
- Assuring the approved design is placed under configuration management.

Detailed design phase SQA activities during the detailed design phase include:

- Assuring the approved design standards are followed.
- Assuring the allocated modules are included in the detailed design.
- Assuring the results of design inspections are included in the design.

Implementation phase SQA activities during the implementation phase include the audit of:

- Status of all deliverable items.
- Configuration management activities.
- Non-conformance reporting and corrective action system.

Integration and test phase SQA activities during the integration and test phase include:

- Assuring readiness for testing of all deliverable items.
- Assuring that all tests are run according to test plans and procedures and that non-conformances are reported and resolved.
- Assuring that test reports are complete and correct.
- Certifying that testing is complete, and the software and its documentation are ready for delivery.

Acceptance and delivery phase As a minimum, SQA activities during the acceptance and delivery phase include assuring the performance of a final configuration audit to demonstrate that all deliverable items are ready for delivery.

13.9.2 Quantitative Approach to QM

In the procedural approach, V&V activities are performed to control the quality by detecting and removing the bugs, but quality at the end of a project cannot be assessed. It means, we have no confidence in quality management. Therefore, the quantitative approach that measures every activity in the life cycle, analyses the data and then, gives judgement about the level of quality. If there is a need to increase the level of quality, then auditing is done to check whether all the procedures and standards are being followed. Thus, this approach covers all the QA activities.

The overall effect of implementing software metrics for software quality can be increased if the whole process of collecting data and analysing things are put into a framework or systematic program of activities called the software metrics programme (SMP). We discuss a model for SMP given by Paul Goodman [34].

Paul Goodman Model for SMP

This model presents a framework for SMP in the lifecycle stages just like SDLC. The idea is to identify and implement the software metrics in the finer detailed intermediate stages. This model comprises of five stages as discussed here (Fig. 13.5).

Figure 13.5 Paul Goodman model for SMP

Initiation stage There are many triggers that can start a metrics program and there are many individuals within an organization who can recognize such a trigger. Some triggers are:

- Productivity/quality concerns
- Customer complaints
- Customer/management requirement for more information
- A cost-cutting environment

Some of the initiators are:

- IT management
- Senior management (non-IT)
- Quality or R&D function

Requirement definition The SMP cannot be directly implemented in an organization without any reason. Therefore, the first stage is to find and analyse the problems in an organization so that these problems can be removed or controlled. Thus, in this stage, we collect the problems and define the requirements where we need to stress. The requirements may be customer identification, market identification, late discovery of project, overflow of cost, etc.

Component design After analysing the requirements in the previous stage, we will have sub-projects or components which must be designed or worked

on for the organization. Every requirement is linked to business needs. For example, cost estimation, reliability prediction, etc. may be the components which must be worked on for the organization.

Implementation Having completed all the four stages, the last stage is to implement the program in an organization. So after preparing a program, it has to be sold to the customers. To help the customer, training sessions and procedures or written guidance addressing the application of measurement must be provided. Changes or tailoring the original program according to local conditions of the organization can also be done. Moreover, help should be extended to the customers on a day-to-day basis.

Software metrics programs can be implemented for a project as well as for a process, if required.

Major Issues for Quantitative Approach

The major issues for quantitatively managing quality for a project are:

Setting quality goal The quality goals can be set in a project in terms of defects. It may be set as the number of defects to be delivered in the last phase of validation testing, i.e. acceptance testing. One method to set the quality goals is to use the past data if the same process is used in similar kind of projects. Using the past data on similar projects, the number of defects in the current project can be estimated, as given by Jalote [3]:

$$\textit{Estimate for defects}(P) = \textit{defects}(SP) \times \textit{effort estimate}(P) \,/\, \textit{actual effort}(SP)$$

where P is the current project and SP is the similar project

Managing the software development process quantitatively The development process is managed towards achieving the quality goal by setting intermediate goals at every intermediate stage and working towards these intermediate goals.

13.10 Software Quality Metrics

Software quality metrics are a subset of software metrics that focus on the quality aspects of the product, process, and project. In general, software quality metrics are more closely associated with process and product metrics than with project metrics. Nonetheless, the project parameters such as the number of developers and their skill levels, the schedule, the size, and the organization structure certainly affect the quality of product. Software quality metrics can be divided further into end-product quality, and based on the findings, to engineer the improvements in both process and product quality. Moreover,

we should view quality from the entire software life cycle perspective and in this regard, we should include metrics that measure the quality level of the maintenance process as another category of software quality metrics. Thus, software quality metrics can be grouped into the following three categories in accordance with the software lifecycle: (i) product quality metrics, in-process quality metrics, and metrics for software maintenance.

Product Quality Metrics

In product quality, the following metrics are considered [42]:

Mean-time to failure (MTTF) MTTF metric is an estimate of the average or mean time until a product's first failure occurs. It is most often used with safety critical systems such as the airline traffic control system.

It is calculated by dividing the total operating time of the units tested by the total number of failures encountered.

Defect density metrics It measures the defects relative to the software size.

$$\textit{Defect density} = \textit{Number of defects / Size of product}$$

Thus, it is the defect rate metrics or the volume of defects generally used for commercial software systems. It is important for cost and resource estimate of the maintenance phase of the software lifecycle.

Customer problem metrics This metric measures the problems which customers face while using the product. The problems may be valid defects or usability problems, unclear documentation, etc. The problem metrics is usually expressed in terms of problems per user month (PUM), as given below:

$$PUM = \frac{\textit{Total problems reported by the customer for a time period}}{\textit{Total number of licensed months of the software during the period}}$$

where number of licensed months = number of installed licenses of the software × number of months in the calculation period.

PUM is usually calculated for each month after the software is released. Basically, this metric relates with the problem of software usage. The goal is to achieve a low PUM by:

(a) Improving the development process and reducing the product defects.

(b) Reducing the non-defect-oriented problems by improving all the aspects of a product.

(c) Increasing the number of installed licenses of the product.

Customer satisfaction metrics Customer satisfaction is usually measured through various methods of customer surveys via the five-point scale.

1. Very satisfied
2. Satisfied
3. Neutral
4. Dissatisfied
5. Very dissatisfied

Based on this scale, several metrics can be prepared like the percentage of completely satisfied customers or the percentage of satisfied customers.

In-process Quality Metrics

In-process quality metrics are less formally defined than the end-product metrics. Following are some metrics [42]:

Defect-density during testing Higher defect rates found during testing is an indicator that the software has experienced higher error injection during its development process. This metric is a good indicator of quality while the software is still being tested. It is especially useful to monitor subsequent releases of a product in the same development organization.

Defect-arrival pattern during testing The pattern of defect arrivals or the time between consecutive failures gives more information. Different patterns of defect arrivals indicate different quality levels in the field. The objective is to have defect arrivals that stabilize at a very low level or times between failures that are far apart.

Defect-removal efficiency

$$DRE = \frac{Defects\ removed\ during\ the\ month}{Number\ of\ problem\ arrivals\ during\ the\ month} \times 100$$

Metrics for Software Maintenance

In this phase, the goal is to fix the defects as soon as possible and with excellent quality. The following metrics are important in software maintenance:

Fix backlog and backlog management index Fix backlog metrics is the count of reported problems that remain open at the end of a month or a week. This metric can provide meaningful information for managing the maintenance process.

Backlog management index (BMI) is also the metric to manage the backlog of open unresolved problems.

$$BMI = \frac{Number\ of\ problems\ closed\ during\ the\ month}{Number\ of\ problem\ arrivals\ during\ the\ month} \times 100$$

If BMI is larger than 100, it means the backlog is reduced; if BMI is less than 100, the backlog is increased. Therefore, the goal is to have a BMI larger than 100.

Fix response time and fix responsiveness While fixing the problems, time-limit also matters according to the severity of the problems. There may be some critical situations in which the customers feel the risk due to defects in the software product, therefore high-severity defects require the response in terms of fixing them as early as possible. The fix response time metric is used for this purpose and is calculated as mean-time of high severity defects from open to closed. Thus, a short fix response time leads to customer satisfaction.

This metric can be enhanced from the customers' viewpoint. Sometimes, the customer sets a time-limit to fix the defects. This agreed-to fix time is known as *fix responsiveness metric.* The goal is to get high fix responsiveness to meet customer expectations and have highly satisfied customers.

Percent delinquent fixes For each fix, if the turn-around time greatly exceeds the required response time, then it is classified as delinquent.

$$\text{Percent delinquent fixes} = \frac{\text{Number of fixes that exceeded the response time criteria by severity level}}{\text{Number of fixes delivered in a specified time}} \times 100$$

Fix quality It is the metric to measure the number of fixes that turn out to be defective. A fix is defective if it did not fix the reported problems or injected a new defect as a consequence of fixing. For real-time software, this metric is important for customer satisfaction. Fix quality is measured as a percent defective fixes. Percent defective fixes is the percentage of all fixes in a time interval that are defective.

13.11 SQA Models

Organizations must adopt a model for standardizing their SQA activities. Many SQA models have been developed in recent years. Some of them are discussed below.

13.11.1 ISO 9126

This is an international software quality standard from ISO. It aims to eliminate any misunderstanding between customers and developers. There may be a case that the customer is not able to communicate the requirements for the product to be developed. Moreover, the developer should also be able to understand the requirement and assess with confidence whether it is possible

to provide the product with the right level of software quality. Thus, there may be problems with both customers and developers which may affect the delivery and quality of the software project. ISO 9126 targets the common understanding between customers and developers.

ISO 9126 provides a set of six quality characteristics with their definitions and associated quality evaluation process to be used when specifying the requirements, and evaluating the quality of software products throughout their life cycle. The quality characteristics are shown in Fig. 13.6. The quality characteristics are accompanied by guidelines for their use. Each attribute is associated with one or more metrics, which allow a value for that attribute to be determined for a particular system.

Figure 13.6 ISO 9126 quality attributes

ISO 9126 defines the following guidelines for implementation of this model:

Quality requirements definition A quality requirement specification is prepared with implied needs, technical documentation of the project, and the ISO standard.

Evaluation preparation This stage involves the selection of appropriate metrics, a rating-level definition, and the definition of assessment criteria. Metrics are quantifiable measures mapped onto scales. The rating levels definition determines what ranges of values on these scales count as satisfactory or unsatisfactory. The assessment criteria definition takes as input a management criterion specific to the project and prepares a procedure for summarizing the results of the evaluation.

Evaluation procedure The selected metrics are applied to the software product and values are taken on the scales. Subsequently, for each measured value, the rating level is determined. After this, a set of rated levels are summarized, resulting in a summary of the software product quality. This summary is then

compared with other aspects such as time and cost, and the final managerial decision is taken, based on managerial criteria.

13.11.2 Capability Maturity Model (CMM)

It is a framework meant for software development processes. This model is aimed to measure the quality of a process by recognizing it on a maturity scale. In this way, matured organizations can be distinguished from immature ones. The maturity of a process directly affects the progress of a project and its results. Therefore, for a high-quality software product, the process adopted for development must be a matured one.

The development of CMM started in 1986 by the Software Engineering Institute (SEI) with MITRE Corporation. They developed a process maturity framework for improving software processes in organizations. After working for four years on this framework, the SEI developed the CMM for software. Later on, it was revised and is continuing to evolve.

CMM Structure

The structure of CMM consists of five maturity levels. These levels consist of key process areas (KPAs) which are organized by common features. These features in turn consist of key practices, decided in KPAs depending on these common features, as shown in Fig. 13.7. The maturity levels indicate the process capability. Key practices describe the activities to be done. Common features address implementations. In this way, by working on the activities described in key practices under some common feature, we achieve a goal under a KPA. In other words, each KPA identifies a cluster of related activities that, when performed collectively, achieve a set of goals considered important for enhancing the process capability.

Figure 13.7(a) CMM Structure

Maturity levels	Indicate	Process capability
Key process areas	Achieve	Goals
Common features	Address	Implementation/Institutionalization
Key practices	Describe	Infrastructure/Activities

Figure 13.7(b) CMM Structure

The structure of CMM is hierarchical in nature such that if one organization wishes to be on maturity level 3, then it must first achieve maturity level 2, and then work according to the goals of the next higher level. It means that we need to attain maturity in a sequenced order of all levels, as shown in Fig. 13.8. The process capability at each maturity level is shown in Table 13.1.

Figure 13.8 Maturity levels

Table 13.1 Process capability at each maturity level

Maturity level	Process Capability
1	—
2	Disciplined process
3	Standard consistent process
4	Predictable process
5	Continuously improving process

Maturity Levels

A brief description of the five maturity levels is given below.

Initial At this level, the process is chaotic or ad-hoc, where there is no control on development progress and success is purely dependent on individual efforts. There is no stable environment for development. The team members may not know the components or goals of development processes. The projects exceed the budget and schedule.

Repeatable Earlier project successes are used here and the lessons learned from past projects are incorporated in the current project. Basic project management processes are established so that cost, schedule, and other parameters can be tracked. However, organization-wide processes still do not exist.

Defined The management scope is widened to the organization here. Project management as well as process management starts. The standard processes are defined with documentations.

Managed A quantitative understanding of the processes is established in order to monitor and control them. Quantitative quality goals are also set for the processes. It means the processes are predictable and controlled.

Optimizing The process is continually improved through incremental and innovative technological changes or improvements. In this way, quantitative process improvement objectives for the organization is identified, evaluated, and deployed.

Key Process Areas

The KPAs and their corresponding goals at each maturity level are discussed in Tables 13.2, 13.3, 13.4, and 13.5.

Table 13.2 KPAs for level 2

KPAs, at this level, focus on the project's concerns related to establishing basic project management controls.	
KPA	***Goals***
Requirement Management	Document the requirements properly. Manage the requirement changes properly.
Software Project Planning	Ensure proper project plan including estimation and listing of activities to be done.
Software Project Tracking and Oversight	Evaluate the actual performance of the project against the plans during project execution. Action, if there is deviation from plan.
Software Sub-contract Management	Selects qualified software sub-contractors. Maintain ongoing communications between prime contractor and the software sub-contractor. Track the software sub-contractor's actual results and performance against its commitments.
Software Quality Assurance	Plan the SQA activities. Ensure that there are proper processes by conducting review and audits. Take proper actions, if projects fail.
Software Configuration Management	Identify work products and documents to be controlled in the project. Control the changes in the software.

Table 13.3 KPAs for level 3

KPAs, at this level, address both project and organizational issues, as the organization establishes an infrastructure that institutionalizes effective software engineering and management processes across all projects.	
KPA	***Goals***
Organization Process Focus	Coordinate process development and improvement activities across the organization. Compare software processes with a process standard. Plan organization-level process development and improvement activities.
Organization Process Definition	Standard software processes are defined and documented.
Training Program	Identify training needs of various team members and implementation of training programs.
Integrated Software Management	Tailor the project from the standard defined process. Manage the project according to defined process.
Software Product Engineering	Define, integrate, and perform software engineering tasks. Keep the work products consistent especially if there are changes.
Inter-group Coordination	Ensure coordination between different groups.
Peer Reviews	Plan peer review activities. Identify and remove defects in the software work products.

Table 13.4 KPAs for level 4

KPAs, at this level, focus on establishing a quantitative understanding of both the software process and the software work products being built.	
KPA	***Goals***
Quantitative Process Management	Plan the quantitative process management activities. Control the performance of the project's defined software process quantitatively. The process capability of the organization's standard software process is known in quantitative terms.
Software Quality Management	Set and plan quantitative quality goals for the project. Measure the actual performance of the project with quality goals and compare them with plans.

Table 13.5 KPAs for level 5

KPAs, at this level, cover the issues that both the organization and the projects must address to implement continuous and measurable software process improvement.	
KPA	***Goals***
Defect Prevention	Plan the defect prevention activities. Identify, prioritize, and eliminate the common causes of bugs.
Technology Change Management	Plan the technology changes to be incorporated in the project, if any. Evaluate the effect of new technologies on quality and productivity.
Process Change Management	Plan the process improvement activities such that an organization-wide participation is there. Measure the change of improvement in the process.

Common Features

As discussed above, all KPAs are organized by common features. The common features are attributes that indicate what to implement according to a KPA goal. The common features have been listed in Table 13.6.

Table 13.6 List of common features

Common Feature	What to Implement/Institutionalize
Commitment to Perform	Organization must take action to ensure that the process is established and will carry on.
Ability to Perform	Define the pre-conditions that must exist in the project or organization to implement the software process competently. Estimate resources, organizational structures, and training.
Activities Performed	Assign the team members and identify procedures to implement a key process area. Perform the work, track it, and take corrective actions as necessary.
Measurement and Analysis	Measure the process and analyse the results.
Verifying Implementation	Ensure that the activities are performed in compliance with the process that has been established. Implement reviews and audits by management and software quality assurance team.

Assessment of Process Maturity

The assessment is done by an assessment team consisting of some experienced team members. These members start the process of assessment by collecting information about the process being assessed. The information about the process can be collected in the form of maturity questionnaires, documentation, or interviews. For assessment, some representative projects of the organization are selected.

13.11.3 Software Total Quality Management (STQM)

Total quality management (TQM) is a term that was originally coined in 1985 by the Naval Air Systems Command to describe its Japanese-style management approach to quality improvement. TQM is a philosophical management strategy to have quality culture in the entire organization. TQM means that the organization's culture is defined by the constant attainment of satisfaction through an integrated system of tools, techniques, and training. Thus, TQM is defined as a quality-centered, customer-focused, fact-based, team-driven, senior-management-led process to achieve an organization's strategic imperative through continuous process improvement.

T = Total = *everyone* in the organization

Q = Quality = *customer satisfaction*

M = Management = *people* and *processes*

Total quality here means to provide *quality product* and *quality services* to the customer, i.e. quality in all aspects. TQM is based on the participation of all the members of an organization to improve processes, products, services, and the culture they work in. TQM is a customer-driven approach, wherein the quality is defined and judged by the customer. It focuses on continuous process improvement to achieve high quality of product (or service). Moreover, the emphasis is to achieve total quality throughout the entire business, not just in the product. Thus, quality is a part of every function in all the phases of a product's lifecycle.

The elements of a TQM system can be summarized as follows:

1. **Customer focus** The goal is to attain total customer satisfaction. It includes studying customers' needs, gathering customer requirements, and measuring and managing customer satisfaction.
2. **Process** The goal is to achieve continuous process improvement. Process includes both the business process and the product development process. Through process improvement, product quality will be enhanced.
3. **Human-side of quality** The goal is to create a company-wide quality culture. Focus areas include management commitment, total participation, employee empowerment, and other social, psychological, and human factors.
4. **Measurement and analysis** The goal is to drive continuous improvement in all quality parameters by the goal-oriented measurement system.

TQM has a few short-term advantages. Most of its benefits are long-term and come into effect only after running smoothly for some time. In large organizations, it may take several years before long-term benefits are realized.

Long-term benefits that may be expected from TQM are: higher productivity, increased morale, reduced costs, and greater customer commitment.

TQM originated in the manufacturing sector, but it has been successfully adopted by almost every type of organization. So it can also be applied to software development because we need to improve our software processes. Remember, we can't test quality in our software, we design in it. And the only way we can design quality in it is by continuously monitoring and improving our processes. As we studied in the previous sections, the software quality is maintained by the SQA team. However, SQA is not enough to achieve the quality standards demanded by the customer. It becomes important to adopt the TQM method to the entire software development organization. This is known as *software total quality management* (STQM). The STQM is also a fundamental cultural change that incorporates quality improvements in every aspect of the software organization. Therefore, SQA provides the methodology to assure quality, while STQM provides the framework to continually improve it. STQM fits well in the components of TQM discussed above.

Customer focus in software development As we know, requirement bugs constitute a large portion of the software bugs. Therefore, gathering customers' needs has become increasingly important in the software industry.

Process, technology, and development quality Given the customers' requirements, the central question is how to develop the software effectively so that it can meet the criterion of 'conformance to customers' requirements'. For this, a matured process and updated technologies are needed to achieve product quality.

Human-side of software quality In software development, at the operational level, TQM can be viewed as the integration of project, process, and quality management with the support of every development team member. During software development, managers must manage in-process quality in the way they manage schedule for quality improvement. Quality, process, and schedule management must be totally integrated for software development to be effective.

Measurement and analysis Software development must be controlled in order to have product delivery on time and within budget, which are the prime concerns for every software development. Software metrics help in measuring various factors during development and thereby, in monitoring and controlling them.

13.11.4 Six Sigma

Six sigma is a quality model originally developed for manufacturing processes. It was developed by Motorola. Six sigma derives its meaning from the field of statistics. Sigma is the standard deviation for a statistical population. Six sigma means, if one has six standard deviations between the mean of a process and the nearest specification limit, there will be practically no items that fail to meet the specifications. Therefore, the goal of this model is to have a process quality of that level. Eventually, six sigma evolved and was applied to other non-manufacturing processes. Today, you can apply six sigma to many fields like services, medical, and insurance procedures, call centers including software.

Six sigma is a way to achieve strategic business results emphasizing on lower costs with less number of defects. Six sigma processes will produce less than 3.4 defects per million opportunities. To achieve this target, it uses a methodology known as DMAIC with the following steps:

- **D**efine opportunities
- **M**easure performance
- **A**nalyse opportunity
- **I**mprove performance
- **C**ontrol performance

This methodology improves any existing business process by constantly reviewing and improving the process.

Six sigma is not just all about statistics. It can be applied to software problems which affects its quality. Six sigma can be applied to analyse the customer requirements and define business goals correspondingly. Its approach to customer requirements, if applied to software development projects, is fundamentally different than those typically practiced in software deployment efforts. It does not start by asking the customers about the requirements first. But it begins by analysing what we need to learn. There are six sigma tools which help in identifying the prioritization of functionalities to be delivered.

Summary

Quality is the concept from where the story of software testing begins. In simple words, we want quality in the software product, which is why we perform testing on it so that a high-quality product can be made. This chapter discusses the concept of quality, its benefits, its various factors, and the costs incurred in it.

In literature, three terms have been in use: quality control, quality assurance, and quality management; all these terms have been distinguished. Quality management is the term which

we use today in the broader sense. The two approaches for quality management, namely procedural approach and quantitative approach, have also been considered in detail. But for a successful high-quality product, we have to pass the project through both approaches. Some quality metrics to monitor and control the quality attributes have also been defined.

Let us review the important concepts described in this chapter:

- Quality is the degree to which a product or service possesses a desired combination of attributes.
- Software quality may be defined in the form of delivered defect density or defect rate, i.e. the number of defects per unit size (e.g. LOC, FP or other unit).
- Quality costs include: prevention costs, appraisal costs, and failure costs.
- Prevention costs are related with activities that identify the cause of defects and actions that are taken to prevent them.
- Appraisal costs include the costs of evaluating the quality of software products at some level, e.g. testing, in-process and inter-process inspection by implementing software metrics program (SMP), equipment calibrations, and maintenance.
- Failure costs include the costs to analyse and remove failures.
- Quality control focuses on finding and removing defects, whereas the main purpose of quality assurance is to verify that applicable procedures and standards are being followed.
- Software quality assurance (SQA) is defined as a planned and systematic approach to evaluate quality and maintain software product standards, processes, and procedures. SQA includes the process of assuring that standards and procedures are established and are followed throughout the software acquisition lifecycle.
- Quality management is an established way of managing the quality of a product wherein quality measures are planned on a larger scale and all controlling and assurance activities are performed according to quality plans.
- The overall effect of implementing software metrics for software quality can be increased if the whole process of collecting data and analysing them are put into a framework or systematic program of activities called software metrics program (SMP).
- Mean-time to failure (MTTF) metric is an estimate of the average or mean-time until a product's first failure occurs. It is calculated by dividing the total operating time of the units tested by the total number of failures encountered.
- Backlog management index (BMI) is the metric to manage the backlog of open and unresolved problems.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Quality is a _____ dimension concept.
   (a) single
   (b) two

(c) three
(d) multi

2. External failure costs are associated with failures that appear on the _____.
   (a) developer's site
   (b) customer's site
   (c) installation time
   (d) none of the above
3. Internal failure costs are those costs that appear on the _____prior to release.
   (a) developer's site
   (b) customer's site
   (c) installation time
   (d) none of the above
4. Quality control is basically related to software _____.
   (a) process
   (b) product
   (c) project
   (d) none of the above
5. Quality assurance is basically related to software _____.
   (a) process
   (b) product
   (c) project
   (d) none of the above
6. Scope of quality management is _____ than control or assurance.
   (a) smaller
   (b) same
   (c) larger
   (d) not related
7. Project management is related to _____ project.
   (a) one
   (b) two
   (c) three
   (d) many
8. Quality management is related to _____ project.
   (a) one
   (b) two
   (c) three
   (d) many

9. Defect density = _____ / size of product
   (a) Number of defects
   (b) Number of test cases
   (c) Number of attributes
   (d) none of the above
10. PUM should be _____.
   (a) high
   (b) low
   (c) average
   (d) none of the above
11. Customer satisfaction is usually measured through various methods of customer surveys via the _____ point scale.
   (a) 3
   (b) 4
   (c) 5
   (d) 6
12. BMI = (Number of problems _____ during the month / Number of problem arrivals during the month) × 100
   (a) closed
   (b) opened
   (c) none of the above
13. If BMI is larger than 100, it means the backlog is _____.
   (a) increased
   (b) reduced
   (c) remains same
   (d) none of the above

Review Questions

1. How does a process affect the quality of a product?
2. Take a project and prepare a list of all the categories of cost of quality in it.
3. What is the difference between quality management and project management? List the various activities under both.
4. How does failure costs reduce if we invest in prevention and appraisal activities? Explain with an example.
5. Differentiate quality control, quality assurance, and quality management.
6. Define the following metrics:
   (a) Defect density
   (b) PUM

(c) BMI
(d) Fix responsiveness
(e) Fix quality

7. Explain the various activities performed in the procedural approach for quality management.
8. Explain the various activities performed in the quantitative approach for quality management.
9. What is CMMI? What is the difference between CMMI and CMM?
10. Search the literature on SPICE quality model and compare it with CMM.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter

14

Testing Process Maturity Models

OBJECTIVES

After reading this chapter, you should be able to understand:

- Testing process should be measured for its maturity and improvement
- Various test maturity models for assessment and improvement of test process
- Test Improvement Model (TIM)
- Test Process Improvement (TPI) Model
- Test Organization Model (TOM)
- Test Maturity Model (TMM)

We have already discussed that software testing is more a systematic process than an intuitive one. Therefore, there should be ways or models for measuring its maturity. The world is working on measurements of the software development processes like CMM, but little attention has been given to measuring the maturity level of a testing process. Many organizations strive hard to get better results out of their testing process. But in the absence of a standardized testing process, they are not able to implement the best practices for software testing.

There are many questions regarding the implementation of a testing process. How should you organize and implement the various testing activities? How should these activities correlated with organizational goals? How should one measure the performance of one testing process? What are the key process areas to improve the current testing process? Whenever an organization plans to make investments in testing practices, the problem of test process assessment arises. We are dependent on conventional SQA models like CMM. However, CMM does not adequately address the testing process or testing-related issues. Thus, it is evident that while we have a standard for measuring the software development process, we also need to have a testing process maturity measurement model.

Some work has been done in the direction of developing the models of testing maturity. The prime purpose of these test maturity models was to provide a foundation for testing process improvement. Many models came up around 1995–96 for assessing the testing process maturity. Some of them have been developed based on the concepts already established in CMM. One of the

major models that has emerged is TMM (Testing Maturity Model), developed by Ilene Burnstein of Illinois Institute of Technology. Another model is TPI (Test Process Improvement), developed by Martin Pol and Koomen. These models provide a list of key process areas according to the test process activities and provide a formal basis for assessing any given system of software testing.

This chapter provides the details of all the major test process maturity models developed so far. The key process areas, levels, scope, and other details are discussed for these models.

14.1 Need for Test Process Maturity

The basic underlying goal for every process is to have continuous process improvement. The testing process should also be designed such that there is scope of its continuous improvement. It merges with our general goals of software development using software engineering, i.e. the software should be tested such that it is released on time, within budget, and is of high quality. But, test process improvement or test process maturity should be understood in the sense of testing terminology, also leading to the goals mentioned above.

There are some key process areas under which we should recognize the testing activities and concentrate on them so that these goals are achieved. However, there should be a mechanism or model which informs us whether the current practices of test process are sufficient or not. If they are not able to give the desired results, then we need to understand and work according to the key process areas and reach a desired level of that test process. The following are some of the reasons why test process maturity models are important:

- The progress and quality of the whole test process should be measured and the results should be used as input for further test process improvement.
- The testing process steps, i.e. test planning, design, execution, etc. are in place. There is a systematic flow of these activities in the organization.
- Testing maturity model provides important information to the testers and management, enabling them to improve the software quality.
- The model should be able to improve the process such that it provides information about the bug's source, i.e. the exact location of their existence, leading to minimization of correction costs.
- The model should be able to improve the process such that it detects high-priority or criticality bugs early.

14.2 Measurement and Improvement of a Test Process

To measure and improve an existing test process, the following steps are used [145]:

Determine target focus areas of test process Why do you want to measure or improve your current testing process? This question should be asked to realize what your long-term goal is. Without realizing the goals, you cannot improve the test process in reality. For example, you may have the target focus on shortening the time-frame of the test process. It may be possible that testing activities are not performed in order, so you want to have a proper test organization in place.

Analyse the testing practices in the current testing process Look for the strong and weak points of the current test process and analyse the aspects which are lacking.

Compare the current practices with a test maturity model Every test activity should be compared with a test maturity model in order to get the best test practices in a process.

Determine key process areas Based on the analysis done in the previous step, list the key process areas where the current process needs attention.

Develop a plan Depending on the key process areas recognized, develop a plan and suggest changes to nullify all the weak points of a process. Based on the suggestions, a modified test process is developed.

Implement the plan Implement the developed plan to improve the existing test process.

14.3 Test Process Maturity Models

Around 1996, many models were developed to measure the maturity of a test process. Some of the models are discussed here.

14.3.1 Testing Improvement Model

The testing improvement model (TIM) was developed by Ericson, Subotic, and Ursing [144]. TIM is used to identify the current state of practice in key areas, and it further suggests ways in which the weakness of a key area can be removed. TIM consists of a

(i) maturity model, and an (ii) Assessment procedure.

Maturity Model

TIM consists of four levels, each consisting of key areas. These are discussed below:

- *Baselining* A baseline work for all the testing activities is established. The test personnel are assigned responsibilities and every work is documented.
- *Cost-effectiveness* After establishing a baseline, efforts are made to improve the cost-effectiveness of testing activities. The focus is on early detection of bugs, training, and re-use.
- *Risk-lowering* Risk management is to have a plan that addresses how to deal with the undesired effects. For this purpose, it is required that the testing team communicates and cooperates with the development team. Ideally, the testing function should participate in all development work and meetings. At the risk-lowering level, metrics are put to use. Metrics are used for determining the fulfillment of test objectives, analysing the cost/benefits of test activities in order to justify expenses, and detecting fault-prone test objects.
- *Optimizing* Optimizing means to move towards continuous improvement. The test process should be able to measure the quality and have a base for improvement.

Key areas of TIM

The five key areas of TIM are:

- Organization
- Planning and tracking
- Testware
- Test cases
- Reviews

Each key area consists of related activities distributed over the four levels.

The Assessment Procedure of TIM

The assessment is performed as interviews, with strategically selected representatives from the organization. The results of the interviews are analysed and presented as 'maturity profile', where the organization's score for each key area is given. Based on the maturity profile, an improvement plan is developed, considering the needs and visions of the organization.

14.3.2 Test Organization Model (TOM)

This model emphasizes that the main hurdles in improving a test process is not only the technical issues, but organizational issues are also responsible. Therefore, while designing the test maturity model, organizational issues should also be considered which have been neglected in other models. Thus, the purpose of TOM is to identify and prioritize the organizational bottlenecks and generate solutions for these problems. This model consists of the following components [143]:

- Questionnaire to identify and prioritize symptoms
- Improvement suggestions

Questionnaire

It focuses on organization issues, as discussed above. Therefore, most of the questions will be answered by management personnel. There are 20 questions in total. This questionnaire serves to calculate a TOM level. The questions are answered with a score of 1–5. The questionnaire has four parts. Part 1 and Part 2 are meant for administrative and analysis purposes. Part 3 and Part 4 are used to calculate a TOM level. The various parts are discussed here:

Part 1 This part has questions on the organization, e.g. name of the organization, address, employee details, turnover, etc.

Part 2 It is the data about the assessor, so that the person in the organization can be contacted.

Part 3 The questions related to objectives and constraints are designed in this part. But the answers are in the form of a rating from 1–5. It aims to prioritize the objectives of the organization.

Part 4 Questions related to the symptoms of poor test process maturity are raised here. There are 20 symptoms. Again, the answers are in the form of a rating. Here, there are two ratings. First rating is the maturity score related to every symptom that shows various maturity levels of that symptom. The maturity scores are categorized in three parts: low (score 1), medium (score 3), and high (score 5). If you feel that your score is in between, then you can provide 2 or 4 as well. Second rating is the impact level of that symptom which is answered in the form of priority. Again, there are priority levels from 1 to 5. A priority 1 symptom can be ignored. A priority 5 is extremely important and must be addressed.

All ratings are added and a TOM level is calculated.

Improvement Suggestions

Based on the TOM level, the model provides testing improvements based on a library of 83 potential improvements. Based on the assessment data entered, the model may generate up to 70 prioritized improvement suggestions.

14.3.3 Test Process Improvement (TPI) Model

The TPI model was developed by Koomen and Pol in 1997 [140,141]. It contains the following parts (Fig. 14.1):

- Key process areas
- Maturity levels
- Test maturity matrix
- Checklist
- Improvement suggestions

Figure 14.1 TPI model structure

In the TPI model, all key areas are not considered equally important for the performance of the whole test process. We can map some dependencies between the different key areas and levels. This mapping has been represented as a *test maturity matrix* in this model. Moreover, the concept of checkpoints at every level has been introduced to verify that the activities have been implemented at a level. If a test process passes all the checkpoints of a certain level, then the process is classified at that level. *Improvement suggestions* have also been added to the model, giving instructions and suggestions for reaching a certain maturity level.

Key Process Areas

In each test process, certain areas need specific attention in order to achieve a well-defined process. These key areas are the basis for improving and structuring the test process. Within the TPI model, 20 key areas have been recognized to determine the maturity of a test process. These 20 areas are further categorized into four groups identified as cornerstones according to the Tmap methodology [142]. These cornerstones, within each test process, require some degree of attention. For a balanced test process, the development of the cornerstones should be in balance. The cornerstones are (see Fig. 14.2):

- Lifecycle (L) of test activities related to the development cycle.
- Good organization (O).
- Right infrastructure and tools (I).
- Usable techniques (T) for performing the activities.

Figure 14.2 TPI key area groups

Lifecycle-related key areas

Test strategy The test strategy goal is to detect critical bugs early in the lifecycle. Moreover, there should be a complete mapping defined in the strategy to show which requirements and quality risks are discovered by what tests.

Lifecycle model The selected lifecycle model must be clearly defined with phases. Activities must be defined under each phase and all the details of every activity like goals, inputs, processes, outputs, dependencies, etc. must be mentioned.

Moment of involvement It emphasizes the concept of early testing. Testing starts as soon as requirements specifications are ready, so as to have less number of bugs. Debugging also becomes easy and less costly.

Techniques-related key areas

Estimating and planning Estimate the resources needed for testing and plan all the test activities.

Test specification techniques Define a standardized test specification technique through which good quality test cases can be designed.

Static test techniques Define the techniques and plan for performing static testing.

Metrics Develop and define standard metrics to measure every test activity.

Infrastructure and tools-related key areas

Test automation/tools Testing tools can have significant benefits, like shorter lead-times and more testing flexibility. Recognize and deploy the tools required at various levels of testing.

Test environment It consists of the following components: procedures, hardware, software, means of communication, facilities for building and using databases and files. The environment has a significant influence on the quality, lead-time, and costs of the testing process.

Office environment It includes all the requirements related to the physical look of the environment where testing is being conducted.

Organization-related key areas

Commitment and motivation All the personnel issues which motivate the testers to be committed towards their responsibility are recognized.

Test functions and training Form a team of skilled persons and provide training, if required.

Scope of methodology The aim is to have a methodology which is sufficiently generic to be applicable in every situation, and contains enough details so that it is not necessary to rethink the same items again each time.

Communication Good communication channels facilitate smooth running of a test process and create good conditions to optimize the test strategy.

Reporting There should be a proper mechanism for reporting bug, status of quality, etc.

Defect management Track the lifecycle of a defect and analyse the quality trends in the detected defects. Such analysis is used, for example, to give well-founded quality advice, process improvement, etc.

Testware management Every activity of testing must be documented and managed in a standard template.

Test process management To control every activity/process, four steps are essential: planning, execution, maintaining, and adjusting. These steps need to be identified for every activity/process.

Evaluation Evaluate all the items being produced, such as SRS, SDD, etc. by testing the intermediate stages of lifecycle and after the product is ready.

Low-level testing Unit testing and integration testing are well-known low-level tests. Low-level testing is efficient because it requires little communication and often, the finder is both the error-producer as well as the one who corrects the defect.

Maturity Levels

There are four basic levels in TPI consisting of 14 scales. Each level consists of a specific range of scales. The maturity level is assigned to every KPA. Further, levels (A, B, C, D) are used to assign a degree of maturity to each key area.

The maturity levels with their allotted scales are:

- **Ad hoc (Scale 0)**
- **Controlled (Scale 1–5)**
  - (i) Defined process steps
  - (ii) Defined test strategy
  - (iii) Test specification techniques
  - (iv) Bug reporting
  - (v) Testware management
- **Efficient (Scale 6–10)** The testing process is automated and integrated in the development organization.
- **Optimizing (Scale 11–13)** Continuous improvement to optimize the organization is a part of the regular methods of the organization.

Test Maturity Matrix

If we want to look at the key areas which need attention and improvement, all the information on maturity of a key area can be maintained in a matrix consisting of key area and maturity levels with their scales, as shown in Fig. 14.3. The main purpose of the matrix is to show the strong and weak points of the current test process. This helps in prioritizing the actions to be taken for improvement.

Key Area	Ad hoc	Controlled					Efficient					Optimizing		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Test Strategy														
Life Cycle Model														
...	Increasing Test Maturity →													

Figure 14.3 Test maturity matrix

Checkpoints

Checkpoints are the helping points to find what a maturity level demands. They are maintained in the form of a list of questions. If the answers to these questions are positive, it means the level is successfully achieved.

Improvement Suggestions

TPI model includes suggestions to improve the test process. These are different kinds of hints and ideas that will help to achieve a certain level of test maturity.

14.3.4 Test Maturity Model (TMM)

TMM was developed in 1996 at the Illinois Institute of Technology [2,3]. It has two major components:

1. A maturity model in which five maturity levels are distinguished (as in CMM)
2. An assessment model containing an assessment procedure, an assessment instrument/questionnaire, team training, and selection criteria

TMM Components

The TMM structure contains several components, which are described here. Figure 14.4 shows the framework of TMM.

Maturity levels Each maturity level represents a specific testing maturity in the evolution to a matured testing process. Each upper level assumes that the lower level has practices that have been executed.

There are five levels within TMM:

- Initial
- Phase definition
- Integration
- Management and measurement
- Optimization/defect prevention and quality control

Figure 14.4 Framework of TMM

Maturity goals To achieve a certain maturity level in the model, some goals have been identified that must be addressed. The maturity goals of TMM are the same as what we call the key process areas in CMM. Goals are identified for each level, except for level 1.

Maturity sub-goals For each maturity goal, the sub-goals that are more concrete are defined. They define the scope, boundaries, and accomplishments for a particular level.

Activities, tasks, and responsibilities (ATRs) To implement the sub-goals identified, there should be a defined set of activities and responsibilities. Therefore, a set of ATRs are designed for each sub-goal. The ATRs address implementation and organizational adaptation issues at a specific level, targeting to improve the testing capability. The responsibility for these activities and tasks is assigned to the three key participants in the testing process: managers, developers/testers, and users/clients.

Examples of ATRs are:

- Upper management provides adequate resources and funding from the committee for testing and debugging (manager view).

- Developers/testers work with managers to set testing and debugging goals (developers/testers view).
- The testing goals for each level of testing: unit, integration, system, and acceptance are set (developers/testers view).
- Users/clients meet with developers/testers to give their view of the nature of testing/debugging and policies. These goals and policies should represent their interests with respect to product quality (users/clients view).

TMM Levels

TMM has been deliberately designed similar to CMM. The idea is to ensure the growth in testing maturity is in tune with the growth in software capability maturity. The characteristics of five TMM levels are given below, as shown in Fig. 14.5.

Figure 14.5 Five levels of TMM

Initial level It is characterized by a random testing process. Testing activities are performed after coding in an ad-hoc way. The objective of testing at this level is to show only the functioning of the software.

Phase definition Testing and debugging are considered separate activities at this level. Testing still follows coding, but is a planned activity. The primary goal of testing at this maturity level is to show that the software meets its specifications.

Integration It assumes that testing is no longer a phase after coding; it is integrated into the entire software lifecycle and gets early testing consideration. Test objectives are established with respect to the requirements based on user and client needs and are used for test case plan, design, and success criteria. There is a test organization in place consisting of various persons with assigned responsibilities.

Management and measurement At this level, testing is recognized as a measured and quantified process. Testing starts as soon as the SRS is prepared, i.e. reviews

at all phases of the development process are now recognized as testing and quality control activities. Testware is maintained for reuse, defects are adequately logged.

Optimization, defect-prevention, and quality control At this level, the testing process is defined and managed. The test process can be measured for costs and other parameters and effectiveness can be monitored. The concept of bug-prevention instead of bug-detection and quality control are practiced. There are mechanisms put in place to fine-tune and improve the testing process.

Maturity goals and sub-goals As discussed above, TMM levels include various goals and under each goal, there are sub-goals. These goals and sub-goals are necessary in order to measure a test process maturity and correspondingly focus on those goals and sub-goals in which the process is deficient. If everything is defined in the model in the form of goals, sub-goals, and ATRs, then it becomes easy to measure the maturity of the process. The goals at each level are discussed below, as shown in Table 14.1.

Table 14.1 TMM maturity goals by level

Maturity Level	Goals
Level 1	--
Level 2	Develop and establish a test policy. Develop testing and debugging goals. Initiate a testing planning process. Institutionalize basic testing techniques and methods.
Level 3	Establish a software test organization. Establish a testing training program. Integrate testing into software life cycle. Control and monitor test process.
Level 4	Establish an organization-wide review program. Establish a test measurement program. Software quality evaluation.
Level 5	Application of process data for defect-prevention. Quality control. Test process optimization.

Level 2 – Phase Definition The maturity goals and sub-goals at Level 2 are given below.

Develop and establish a test policy An organization should clearly set the objectives and accordingly, develop a test policy. This goal aims to develop and establish a test policy and an overall test approach for the organization. The sub-goals, under this goal, are:

1. Define the organization's overall test objectives aligned with the overall business.
2. Objectives should be translated into a set of high-level key test performance indicators. The establishment of performance objectives and indicators provides clear direction and communication of expected and achieved levels of performance.
3. Define an overall test approach like the V-approach.
4. Identify the levels of testing in the test approach, such as unit, integration, and system testing levels and identify the goals, responsibilities, and main tasks at each level.

Develop testing and debugging goals This goal emphasizes that testing and debugging are two separate processes. Testing process now is a planned activity with identified objectives and test approach, rather than a random process. The sub-goals, under this goal, are given below:

1. Make separate teams for testing and debugging.
2. The teams must develop and record testing goals.
3. The teams must develop and record debugging goals.
4. The documented testing and debugging goals must be distributed to all project managers and developers.
5. Test plans should be based on these testing goals.

Initiate a test planning process If we want to make a process repeatable, defined, and managed, then it must be planned properly. The test planning may involve several activities, such as analysing risks, making the test strategy according to the test phase, and identifying risks and specifications for designing the test cases. The sub-goals, under this goal, are:

1. An organization-wide test planning team must be formed.
2. A framework for organization-wide test planning policies must be established and supported by the management.
3. A test plan template must be developed and made available to project managers and developers.
4. Test plan should be prepared with the inputs from the requirement specifications.

Institutionalize basic testing techniques and methods Some testing techniques and tools must be identified so that the test cases are designed and executed effectively. Examples of basic techniques and methods are black-box and white-box testing strategies, the use of a requirements traceability matrix, etc.

The sub-goals, under this goal, are:

1. An organization-wide test technology group must be formed to study, evaluate, and recommend a set of basic testing techniques and methods, and recommend a set of simple tools to support them.
2. Management must establish and encourage a set of policies that ensures recommended techniques and methods are consistently applied throughout the organization.

Test environment The idea under this goal is to establish and maintain an integrated software and hardware environment in which it is possible to execute the tests in a manageable and repeatable way. The major concern is that the environment should be closer to the actual one, so that more bugs can be found. The sub-goals, under this goal, are:

1. Test environments are specified and their availability on time is ensured in the projects.
2. For higher test levels, the test environment is as 'real-life' as possible.
3. Test environments are managed and controlled according to documented procedures.

Level 3 – Integration The maturity goals and sub-goals at Level 3 are given below.

Establish a software test organization A test organization including the various test persons with their assigned responsibilities is necessary, as testing includes many complex activities. The sub-goals under this goal are:

1. An organization-wide testing team must be established with a hierarchical order of various team members.
2. Roles and responsibilities must be defined for the testing team.
3. The testing team must participate in discussions with the customer and the developer to incorporate user needs, requirements, and other design issues into the testing process.

Establish a technical training program A technical training program for test planning, testing methods, standards, techniques, review process, and tools should be provided to the testing team. Training includes in-house courses, self-study, and mentoring programs. The sub-goals, under this goal, are:

1. Training needs, goals, and plans must be developed from time to time.
2. Management must establish an organizational training program, based on identified goals and plans.

3. An in-house training group must be established with the tools, facilities, and materials in place.

Integrate testing into the software lifecycle The early testing concept is recognized in this goal. All the team members of development and testing should realize at this level that the testing activity should be conducted parallel to all lifecycle phases in order to get a high-quality software. Testing activities must start as soon as the SRS is prepared. The sub-goals, under this goal, are:

1. Identify the phases of SDLC where testing will be performed in order to integrate the testing activities into the software lifecycle.
2. Develop a customized V-model, based on defined testing sub-phases.
3. Standard definitions must be developed for the testware to be used in the organization.
4. Testers must participate in review meetings like requirement specifications with analysts, design meetings with developers in order to get familiar with the customer, level requirements and all design issues. This will help testers in reviewing, planning, and designing every test activity.

Control and monitor the testing process Several controlling and monitoring activities ensure that the testing process proceeds according to the plan. Every test activity must be tracked and controlled according to the test plans. The sub-goals are:

1. The organization must develop mechanisms and policies to control and monitor the testing process.
2. A set of basic test process-related measurements must be defined, recorded, and distributed.
3. A set of corrective actions and contingency plans must be developed, recorded, and documented for use when the testing deviates significantly from what is planned.

Level 4 – Management and Measurement At this level, the meaning of 'testing activity' is expanded such that reviews, inspections, and walkthroughs at all phases of the lifecycle are also a part of it. Every testware is verified. This expanded definition of testing covers activities typically categorized as verification and validation activities. Moreover, the controlling and monitoring functions discussed in the previous level are now supported by an established test measurement program. Test cases and procedures are stored for reuse and regression testing. The maturity goals and sub-goals at Level 4 are discussed here.

Establish an organization-wide review program Reviews are conducted at all the phases of the lifecycle to identify, catalog, and remove defects from software work products and to test work products early and effectively. The sub-goals are:

1. The upper management must develop review policies, support the review process, and take responsibility for integrating them into the organizational culture.
2. The testing team and the SQA team must develop and document goals, plans, follow-up procedures, and recording mechanisms for reviews throughout the software lifecycle.
3. Items for review must be specified by the above-mentioned two teams.
4. Personnel must be trained so that they understand and follow proper review policies, practices, and procedures.

Establish a test measurement program A formalized test-measurement program should be prepared keeping in view the broad organizational goals. Measurements may include test progress, test costs, data on errors and defects, and product measures such as software reliability. The sub-goals are:

1. Organization-wide test measurement policies and goals must be defined.
2. A test measurement plan must be developed with mechanisms for data collection, analysis, and application.
3. Action plans that apply measurement results to test process improvements must be developed and documented.

Software quality evaluation The test-measurement program must be correlated with the quality goals of the software and these goals must be evaluated through the metrics developed. The sub-goals, under this goal, are:

1. Quality goals must be developed and known to all team members.
2. The testing process must be structured, measured, and evaluated to ensure that the quality goals can be achieved.

Level 5 – Optimization, Defect-prevention, and Quality control The maturity goals and sub-goals at Level 5 are given below:

Application of process data for defect-prevention At this level, organizations must be able to learn from their bug history and prevent the bugs before they occur. Bug-prevention is applied across all projects and across the organization. A defect-prevention team is responsible for the defect-prevention activities, interacting with developers to apply defect-prevention activities throughout the lifecycle. The sub-goals, under this goal, are given below:

1. A bug-prevention team must be established with management support.
2. Bugs history must be recorded at every phase of lifecycle.
3. A causal analysis mechanism must be established to identify the root causes of bugs.
4. Action plans must be developed through the interaction of managers, developers, and testers to prevent identified defects from recurring. These plans must be tracked.

Quality control At this level, organizations use statistical sampling, measurements of confidence levels, trustworthiness, and reliability goals to drive the testing process and consequently control the quality parameters. The sub-goals, under this goal, are:

1. SQA group must establish quality goals for software products such as product unit defectiveness, confidence levels, and trustworthiness.
2. Test managers must incorporate these quality goals into the test plans.
3. The test group must be trained in statistical methods.

Test process optimization There is scope of continuous improvement for any test process. Therefore, the test process is quantified and can be fine-tuned so that it is optimized. Optimizing the testing process involves evaluation of every test activity and adaptation of improvements. The sub-goals are:

1. Assign the responsibility to monitor the testing process and identify areas for improvement to a group.
2. A mechanism must be in place to evaluate new tools and technologies that may improve the capability and maturity of the testing process.
3. The effectiveness of the testing process must be continually evaluated in a measurable and optimal manner.

The key process areas (KPAs) of each level are discussed in the tables below.

Table 14.2 KPAs at Level 2

KPAs	Associated activities
Develop testing and debugging goals	Various goals, tasks, activities, and tools for each must be identified. Responsibilities for each must be assigned.
Initiate a testing planning process	Focus on initiating a planning process which involves stating objectives, analysing risks, outlining strategies, and developing test design specifications and test cases. Also addresses the allocation of recourses and responsibilities for testing on the unit, integration system, and acceptance levels.
Institutionalize basic testing techniques and methods	Emphasis on applying basic testing techniques and methods to improve test process capability across the organization.

Table 14.3 KPAs at Level 3

KPAs	Associated activities
Establish a software test organization	Establishment of a software test organization to identify a group of people who are responsible for testing.
Establish a testing training program	Establishment of a testing training program to ensure a skilled staff is available to the testing group.
Integrate testing into software life cycle	Integration of testing sub-phases into software life cycle. Also a mechanism must be established that allows testers to work with developers, which facilitates testing activity integration.
Control and monitor test process	Several controlling and monitoring activities are performed which provide visibility and ensure that the testing process proceeds according to plan.

Table 14.4 KPAs at Level 4

KPAs	Associated activities
Establish an organization-wide review program	Establishment of a review program to remove defects from software work products and to test work products early and effectively.
Establish a test measurement program	Organizational–wide test measurement policies and goals must be defined as well as action plans that apply measurement results to test process improvements must be developed and documented.
Software quality evaluation	Software quality evaluation involves defining measurable quality attributes and defining quality goals to evaluate software work products. Quality goals are tied to testing process adequacy.

Table 14.5 KPAs at Level 5

KPAs	Associated activities
Application of process data for defect-prevention	A defect-prevention team must be established with management support and the defects injected or removed must be identified and recorded during each phase.
Quality control	Organizations use statistical sampling, measurement of confidence levels, trustworthiness, and reliability goals to drive the testing process.
Test process optimization	Testing process is subjected to continuous improvement across projects and across the organization. A mechanism must be in place to evaluate new tools and technologies that may improve the capability and maturity of the testing process.

The Assessment Model

The TMM assessment model (TMM-AM) supports self-assessment of the testing process. The TMM-AM has three major components:

The tools for assessment It contains the tools for assessing the test process. For example, a questionnaire can be prepared. The questions designed may relate to maturity goals and process issues described at each level.

The assessment procedure The assessment procedure will give the assessment team guidelines on how to collect, organize, analyse, and interpret the data collected from questionnaires and personal interviews.

Training procedure A training procedure should be used to instruct the personnel in test process assessment.

SUMMARY

TMM models have been developed to measure and improve the maturity of a testing process. These models can be used along with SDLC maturity models like CMM. These models cannot replace SDLC maturity models, but they can assess an existing test process and suggest the areas where an organization can improve. Thus, an organization can use both SDLC and STLC maturity models in parallel.

This chapter discusses the importance, development, levels, and key process areas of various models: TPI, TOM, TIM, and TMM. The key process areas are important to recognize an organization, what its capabilities are, and what type of testing process it uses. Therefore, first identify the key process areas, check all of them with your organization, rate the organization at a maturity level, and strive for another higher level.

The major motive behind the maturity models is to produce good quality software by measuring the testing process. The test maturity models discussed in this chapter help in (i) improving the test process, (ii) attaining the next CMM level, (iii) identifying the areas to be improved, (iv) identifying the testing processes that can be adopted company-wide, and (v) providing a roadmap for implementing improvements.

Let us review the important concepts described in this chapter:

- The test maturity model informs whether the current practices of test process are sufficient or not.
- Testing maturity model provides important information to the testers and management, enabling them to improve the software quality.
- The testing improvement model (TIM) is used to identify the current state of practice in key areas and further, it suggests ways in which the weakness of a key area can be removed. It consists of a maturity model and an assessment procedure. The five key areas of TIM are: organization, planning and tracking, testware, test cases, and reviews.
- Test organization model (TOM) identifies and prioritizes the organizational bottlenecks and generates solutions for these problems. This model consists of the following components: (i) questionnaire to identify and prioritize symptoms and (ii) improvement suggestions.
- Test process improvement (TPI) model does not consider all key areas equally important for the performance of the whole test process. We can map some dependencies between the different key areas and levels. This mapping has been represented as a test maturity matrix in this model. Moreover, the concept of checkpoints at every level has

been introduced to verify that the activities have been completed at a level. It contains the following parts: key process areas, maturity levels, test maturity matrix, a checklist, and improvement suggestions.

- Within the TPI model, 20 key areas have been recognized to determine the maturity of the test process. These areas are further categorized into four groups identified as the cornerstones, namely: lifecycle (L) of test activities related to the development cycle, good organization (O), right infrastructure and tools (I), and usable techniques (T) for performing the activities.
- There are four basic maturity levels in TPI consisting of 14 scales. Each level consists of a specific range of scales. The maturity level is assigned to every KPA.
- In TPI, all the information of maturity of a key area can be maintained in a matrix consisting of key areas and maturity levels with their scales known as test maturity matrix.
- Test maturity model (TMM) has two major components: a maturity model with five maturity levels and an assessment model.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. The total number of questions in a TOM questionnaire are _____.
   (a) 30
   (b) 20
   (c) 10
   (d) 15
2. The TMM is characterized by _____ testing maturity levels.
   (a) 2
   (b) 3
   (c) 4
   (d) 5
3. Initial level is at TMM level _____.
   (a) 1
   (b) 2
   (c) 3
   (d) 4
4. Integration is at TMM level _____.
   (a) 1
   (b) 2
   (c) 3
   (d) 4

5. Management and measurement is at TMM level _____.
   (a) 1
   (b) 5
   (c) 3
   (d) 4
6. Optimization, defect-prevention, and quality control measurement is at TMM level _____.
   (a) 1
   (b) 5
   (c) 3
   (d) 4
7. Phase definition measurement is at TMM level _____.
   (a) 1
   (b) 2
   (c) 3
   (d) 4
8. The maturity goal 'develop testing and debugging goals' is at TMM level _____.
   (a) 1
   (b) 2
   (c) 3
   (d) 4
9. The maturity goal 'establish an organization-wide review program' is at TMM level _____.
   (a) 1
   (b) 2
   (c) 3
   (d) 4
10. The maturity goal 'control and monitor the testing process' is at TMM level _____.
   (a) 5
   (b) 2
   (c) 3
   (d) 4
11. The maturity goal 'quality control' is at TMM level _____.
   (a) 5
   (b) 2
   (c) 3
   (d) 4

Review Questions

1. What is the need of a test process maturity model?
2. Explain all the key areas mentioned in TIM.
3. Elaborate the assessment procedure in TIM.
4. What are the key areas and maturity levels mentioned in TPI?
5. What is the meaning of test maturity matrix in TPI? How do you develop this matrix?
6. How many levels are there in TMM?
7. Explain all the key process areas at each TMM level.
8. Compare and contrast the test process maturity models mentioned in the chapter.
9. What is the correlation between CMM and TMM?
10. How does TMM help in process improvement?
11. Develop a checklist for the activities to be done in all TMM levels, and find out if your organization follows them.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Part 5 Test Automation

CHAPTER

Chapter 15:
Automation and Testing Tools

The testing tools provide automation to almost every technique discussed in Part II. Therefore, testing tools can be adopted for all static and dynamic testing techniques to save time and money. This part discusses the need for automation and the various categories of testing tools. However, the testing tools are costly. Therefore, these tools should be selected carefully, keeping in mind their use in STLC. First, analyse the need for automation and then, go for the testing tool. One should not rely completely on testing tools. They cannot replace the whole testing activity. Thus, automated tools are merely a part of the solution and they are not a magical answer to the testing problem.

This part will make ground for the following concepts:

- Need for automation
- Static and dynamic tools
- Testing activity tools
- Costs incurred in testing tools
- Guidelines for automated testing
- Overview of some commercial tools

Chapter

15

Automation and Testing Tools

OBJECTIVES

After reading this chapter, you should be able to understand:

- Need for automating the testing activities
- Testing tools are not silver bullets to solve all the testing problems
- Static and dynamic testing tools
- Selection of testing tools
- Costs incurred in adopting a testing tool
- Guidelines of testing automation

As seen in Chapter 2, testing tools are a part of testing tactics. The testing techniques, either static or dynamic, can be automated with the help of software tools which can greatly enhance the software testing process. Testing today is not a manual operation but is assisted with many efficient tools that help the testers. Test automation is the most glamorous part of software testing. Testing automation is effective such that any kind of repetitive and tedious activities can be done by machines, and testers can utilize their time in more creative and critical work.

In the last decade, a lot of testing tools have been developed for use throughout the various SDLC phases. But the major part is the selection of tools from a pool of various categories of tools. Apart from the high cost of these tools, a single tool may not cover the whole testing automation. Thus, tools must be selected according to their application and needs of the organization. Automation is bound to fail if chosen for wrong reasons at the wrong places in SDLC.

However, automated testing should not be viewed as a replacement for manual testing. There is a misconception among professionals that software testing is easy as you only run the test cases with automated tools. This is the reason why newcomers prefer testing jobs. But the truth is that testers have many duties in the development and it is not only about running automated tools. There are many activities in the testing life cycle which cannot be automated and manual effort is required. Thus, automated tools are merely a part of the solution; they are not a magical answer to the testing problem. Automated testing tools will never replace the analytical skills required to conduct the test, nor will they replace manual testing. It must be seen as an enhancement to the manual testing process [12].

15.1 Need for Automation

If an organization needs to choose a testing tool, the following benefits of automation must be considered.

Reduction of testing effort In verification and validation strategies, numerous test case design methods have been studied. Test cases for a complete software may be hundreds of thousands or more in number. Executing all of them manually takes a lot of testing effort and time. Thus, execution of test suits through software tools greatly reduces the amount of time required.

Reduces the testers' involvement in executing tests Sometimes executing the test cases takes a long time. Automating this process of executing the test suit will relieve the testers to do some other work, thereby increasing the parallelism in testing efforts.

Facilitates regression testing As we know, regression testing is the most time-consuming process. If we automate the process of regression testing, then testing effort as well as the time taken will reduce as compared to manual testing.

Avoids human mistakes Manually executing the test cases may incorporate errors in the process or sometimes, we may be biased towards limited test cases while checking the software. Testing tools will not cause these problems which are introduced due to manual testing.

Reduces overall cost of the software As we have seen that if testing time increases, cost of the software also increases. But due to testing tools, time and therefore cost can be reduced to a greater level as testing tools ease the burden of the test case production and execution.

Simulated testing Load performance testing is an example of testing where the real-life situation needs to be simulated in the test environment. Sometimes, it may not be possible to create the load of a number of concurrent users or large amount of data in a project. Automated tools, on the other hand, can create millions of concurrent virtual users/data and effectively test the project in the test environment before releasing the product.

Internal testing Testing may require testing for memory leakage or checking the coverage of testing. Automation tools can help in these tasks quickly and accurately, whereas doing this manually would be cumbersome, inaccurate, and time-consuming.

Test enablers While development is not complete, some modules for testing are not ready. At that time, stubs or drivers are needed to prepare data, simulate environment, make calls, and then verify results. Automation reduces the effort required in this case and becomes essential.

Test case design Automated tools can be used to design test cases also. Through automation, better coverage can be guaranteed, than if done manually.

15.2 Categorization of Testing Tools

A single tool may not cover the whole testing process, therefore, a variety of testing tools are available according to different needs and users. The different categories of testing tools are discussed here:

15.2.1 Static and Dynamic Testing Tools

These tools are based on the type of execution of test cases, namely static and dynamic, as discussed in software testing techniques:

Static testing tools For static testing, there are static program analysers which scan the source program and detect possible faults and anomalies. These static tools parse the program text, recognize the various sentences, and detect the following:

- Statements are well-formed.
- Inferences about the control flow of the program.
- Compute the set of all possible values for program data.

Static tools perform the following types of static analysis:

Control flow analysis This analysis detects loops with multiple exits and entry points and unreachable code.

Data use analysis It detects all types of data faults.

Interface analysis It detects all interface faults. It also detects functions which are never declared and never called or function results that are never used.

Path analysis It identifies all possible paths through the program and unravels the program's control.

Dynamic testing tools These tools support the following:

- Dynamic testing activities.
- Many a times, systems are difficult to test because several operations are being performed concurrently. In such cases, it is difficult to anticipate conditions and generate representative test cases. Automated test tools enable the test team to capture the state of events during the execution of a program by preserving a snapshot of the conditions. These tools are sometimes called *program monitors*. The monitors perform the following [66] functions:

- List the number of times a component is called or line of code is executed. This information about the statement or path coverage of their test cases is used by testers.
- Report on whether a decision point has branched in all directions, thereby providing information about branch coverage.
- Report summary statistics providing a high-level view of the percentage of statements, paths, and branches that have been covered by the collective set of test cases run. This information is important when test objectives are stated in terms of coverage.

15.2.2 Testing Activity Tools

These tools are based on the testing activities or tasks in a particular phase of the SDLC. Testing activities can be categorized as [1]:

- Reviews and inspections
- Test planning
- Test design and development
- Test execution and evaluation

Now, we will discuss the testing tools based on these testing tasks.

Tools for review and inspections Since these tools are for static analysis on many items, some tools are designed to work with specifications but there are far too many tools available that work exclusively with code. In this category, the following types of tools are required:

Complexity analysis tools It is important for testers that complexity is analysed so that testing time and resources can be estimated. The complexity analysis tools analyse the areas of complexity and provide indication to testers.

Code comprehension These tools help in understanding dependencies, tracing program logic, viewing graphical representations of the program, and identifying the dead code. All these tasks enable the inspection team to analyse the code extensively.

Tools for test planning The types of tools required for test planning are:

1. Templates for test plan documentation
2. Test schedule and staffing estimates
3. Complexity analyser

Tools for test design and development Discussed below are the types of tools required for test design and development.

Test data generator It automates the generation of test data based on a user defined format. These tools can populate a database quickly based on a set of rules, whether data is needed for functional testing, data-driven load testing, or performance testing.

Test case generator It automates the procedure of generating the test cases. But it works with a requirement management tool which is meant to capture requirements information. Test case generator uses the information provided by the requirement management tool and creates the test cases. The test cases can also be generated with the information provided by the test engineer regarding the previous failures that have been discovered by him. This information is entered into this tool and it becomes the knowledge-based tool that uses the knowledge of historical figures to generate test cases.

Test execution and evaluation tools The types of tools required for test execution and evaluation are [1]:

Capture/playback tools These tools record events (including keystrokes, mouse activity, and display output) at the time of running the system and place the information into a script. The tool can then replay the script to test the system [22].

Coverage analysis tools These tools automate the process of thoroughly testing the software and provide a quantitative measure of the coverage of the system being tested. These tools are helpful in the following:

- Measuring structural coverage which enables the development and test teams to gain insight into the effectiveness of tests and test suites
- Quantifying the complexity of the design
- Help in specifying parts of the software which are not being covered
- Measure the number of integration tests required to qualify the design
- Help in producing integration tests
- Measuring the number of integration tests that have not been executed
- Measuring multiple levels of test coverage, including branch, condition, decision/condition, multiple conditions, and path coverage

Memory testing tools These tools verify that an application is properly using its memory resources. They check whether an application is:

- Not releasing memory allocated to it
- Overwriting/overreading array bounds
- Reading and using uninitialized memory

Test management tools Test management tools try to cover most of the activities in the testing life cycle. These tools may cover planning, analysis, and design. Some test management tools such as Rational's TestStudio are integrated with requirement and configuration management and defect tracking tools, in order to simplify the entire testing life cycle [12].

Network-testing tools There are various applications running in the client-server environments. However, these applications pose new complexity to the testing effort and increases potential for errors due to inter-platform connectivity. Therefore, these tools monitor, measure, test, and diagnose performance across an entire network including the following:

- Cover the performance of the server and the network
- Overall system performance
- Functionality across server, client, and the network

Performance testing tools There are various systems for which performance testing is a must but this becomes a tedious job in real-time systems. Performance testing tools help in measuring the response time and load capabilities of a system.

15.3 Selection of Testing Tools

The big question is how to select a testing tool. It may depend on several factors. What are the needs of the organization; what is the project environment; what is the current testing methodology; all these factors should be considered when choosing testing tools. Some guidelines to be followed by selecting a testing tool are given below.

Match the tool to its appropriate use Before selecting the tool, it is necessary to know its use. A tool may not be a general one or may not cover many features. Rather, most of the tools are meant for specific tasks. Therefore, the tester needs to be familiar with both the tool and its uses in order to make a proper selection.

Select the tool to its appropriate SDLC phase Since the methods of testing changes according to the SDLC phase, the testing tools also change. Therefore, it is necessary to choose the tool according to the SDLC phase, in which testing is to be done.

Select the tool to the skill of the tester The individual performing the test must select a tool that conforms to his skill level. For example, it would be inappropriate for a user to select a tool that requires programming skills when the user does not possess those skills.

Select a tool which is affordable Tools are always costly and increase the cost of the project. Therefore, choose the tool which is within the budget of the project. Increasing the budget of the project for a costlier tool is not desired. If the tool is under utilization, then added cost will have no benefits to the project. Thus, once you are sure that a particular tool will really help the project, then only go for it otherwise it can be managed without a tool also.

Determine how many tools are required for testing the system A single tool generally cannot satisfy all test requirements. It may be possible that many test tools are required for the entire project. Therefore, assess the tool as per the test requirements and determine the number and type of tools required.

Select the tool after examining the schedule of testing First, get an idea of the entire schedule of testing activities and then decide whether there is enough time for learning the testing tool and then performing automation with that tool. If there is not enough time to provide training on the tool, then there is no use of automation.

15.4 Costs incurred in Testing Tools

Automation is not free. Obviously employing the testing tools incur a high cost. Moreover, before acquiring the tools, significant work is required. Following are some facts pertaining to the cost incurred in testing tools [22, 12]:

Automated script development Automated test tools do not create test scripts. Therefore, a significant time is needed to program the tests. Scripts are themselves programming languages. Thus, automating test execution requires programming exercises.

Training is required It is not necessary that the tester will be aware of all the tools and can use them directly. He may require training regarding the tool, otherwise it ends up on the shelf or implemented inefficiently. Therefore, it becomes necessary that in a new project, cost of training on the tools should also be included in the project budget and schedule.

Configuration management Configuration management is necessary to track large number of files and test related artifacts.

Learning curve for the tools There is a learning curve in using any new tool. For example, test scripts generated by the tool during recording must be modified manually, requiring tool-scripting knowledge in order to make the script robust, reusable, and maintainable.

Testing tools can be intrusive It may be necessary that for automation some tools require that a special code is inserted in the system to work correctly and to be integrated with the testing tools. These tools are known as *intrusive tools* which require addition of a piece of code in the existing software system. Intrusive tools pose the risk that defects introduced by the code inserted specifically to facilitate testing could interfere with the normal functioning of the system.

Multiple tools are required As discussed earlier, it may be possible that your requirement is not satisfied with just one tool for automation. In such a case, you have to go for many tools which incur a lot of cost.

15.5 Guidelines for Automated Testing

Automation is not a magical answer to the testing problem. Testing tools can never replace the analytical skills required to conduct testing and manual testing. It incurs some cost as seen above and it may not provide the desired solution if you are not careful. Therefore, it is necessary that you carefully plan the automation before adopting it. Decide which tool and how many tools are required, how much resources are required including the cost of the tool and the time spent on training.

Discussed below are the guidelines to be followed if you have planned for automation in testing.

Consider building a tool instead of buying one, if possible It may not be possible every time. But if the requirement is small and sufficient resources allow, then go for building the tool instead of buying, after weighing the pros and cons. Whether to buy or build a tool requires management commitment, including budget and resource approvals.

Test the tool on an application prototype While purchasing the tool, it is important to verify that it works properly with the system being developed. However, it is not possible as the system being developed is often not available. Therefore, it is suggested that if possible, the development team can build a system prototype for evaluating the testing tool.

Not all the tests should be automated Automated testing is an enhancement of manual testing, but it cannot be expected that all test on a project can be automated. It is important to decide which parts need automation before going for tools. Some tests are impossible to automate, such as verifying a printout. It has to be done manually.

Select the tools according to organizational needs Do not buy the tools just for their popularity or to compete with other organizations. Focus on the needs of the organization and know the resources (budget, schedule) before choosing the automation tool.

Use proven test-script development techniques Automation can be effective if proven techniques are used to produce efficient, maintainable, and reusable test scripts. The following are some hints:

1. Read the data values from either spreadsheets or tool-provided data pools, rather than being hard-coded into the test-case script because this prevent test cases from being reused. Hard-coded values should be replaced with variables and whenever possible read data from external sources.
2. Use modular script development. It increases maintainability and readability of the source code.
3. Build library of reusable functions by separating the common actions into shared script library usable by all test engineers.
4. All test scripts should be stored in a version control tool.

Automate the regression tests whenever feasible Regression testing consumes a lot of time. If tools are used for this testing, the testing time can be reduced to a greater extent. Therefore, whenever possible, automate the regression test cases.

15.6 Overview of Some Commercial Testing Tools

Mercury Interactive's WinRunner It is a tool used for performing functional /regression testing. It automatically creates the test scripts by recording the user interactions on GUI of the software. These scripts can be run repeatedly whenever needed without any manual intervention. The test scripts can also be modified if required because there is support of Test Script language (TSL) with a 'C' like syntax. There is also provision for bringing the application to a known state if any problem has occurred during automated testing. WinRunner executes the statements by default with an interleaving of one second. But if some activities take more time to complete, then it synchronizes the next test cases automatically by waiting for the current operations to be completed.

Segue Software's SilkTest This tool is also for functional/regression testing. It supports 4Test as a scripting language which is an object-oriented scripting language. SilkTest has a provision for customized in-built recovery system

which helps in continuing the automated testing even if there is some failure in between.

IBM Rational SQA Robot It is another powerful tool for functional/regression testing. Synchronization of test cases with a default delay of 20 seconds is also available.

Mercury Interactive's LoadRunner This tool is used for performance and load testing of a system. Generally, the tool is helpful for client/server applications of various parameters with their actual load like response time, the number of users, etc. The major benefit of using this tool is that it creates virtual users on a single machine and tests the system on various parameters. Thus, performance and load testing is done with minimum infrastructure.

Apache's JMeter This is an open-source software tool used for performance and load testing.

Mercury Interactive's TestDirector TestDirector is a test management tool. It is a web-based tool with the advantage of managing the testing if two teams are at different locations. It manages the test process with four phases: specifying requirements, planning tests, running tests, and tracking defects. Therefore, there is advantage that the tests are planned as per the requirements evolved. Once the test plan is ready, the test cases are executed. Defect-tracking can be done in any phase of test process. During defect-tracking, new bug can be reported; assign responsibility to someone for bug repair; assign priority to the bug; bug repair status can be analysed, etc. This tool can also be integrated with LoadRunner or WinRunner.

SUMMARY

Testing tools are a big aid to the testers. Testing activities, if performed manually, may take a lot of time; thus there is need for automating the testing activities. This chapter discusses the importance and need of testing automation and various types of testing tools. The tools are mainly categorized as static and dynamic testing tools.

Testing tools are helpful to the testers, but they cannot replace all the manual testing activities. There are some places where we have to perform manual testing.

There was a time when it was thought that testing tools will fasten the speed of software development and the manual effort will be zero. But there is no silver bullet like this. The guidelines for automated testing are also provided at the end.

Let us review the important concepts described in this chapter:

- Testing tools are based on the type of execution of test cases, namely static and dynamic.

- Static tools perform control flow analysis, data use analysis, interface analysis, and path analysis.
- Automated test tools enable the test team to capture the state of events during the execution of a program by preserving a snapshot of conditions. These tools are sometimes called program monitors.
- Testing activity tools are based on the testing activities or tasks in a particular phase of the SDLC.
- The complexity analysis tools analyse the areas of complexity and provide indication to testers.
- Code comprehension tools help inspection team to analyse the code extensively by understanding dependencies, tracing program logic, viewing graphical representations of the program, and identifying the dead code.
- Test data generator automates the generation of test data based on a user-defined format.
- Test case generator uses the information provided by the requirement management tool and creates the test cases.
- Capture/playback tools record events at the time of running the system and place the information into a script which can be run by the tester later.
- Test coverage analysis tools provide a quantitative measure of the coverage of the system being tested.
- Memory testing tools verify that an application is properly using its memory resources.
- Test management tools try to cover most of the activities in the testing life cycle. These tools may cover planning, analysis, and design.
- Network testing tools monitor, measure, test, and diagnose performance across an entire network.
- Performance testing tools help in measuring the response time and load capabilities of a system.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Tools that enable the test team to capture the state of events during the execution of a program by preserving a snapshot of conditions are called _____.
   (a) monitor program
   (b) event program
   (c) snapshot program
   (d) code comprehension

2. Tools that help in understanding dependencies, tracing program logic, viewing graphical representations of the program, and identifying the dead code are called _____.
   (a) monitor program
   (b) event program
   (c) code comprehension
   (d) none of the above
3. Tools that automate the generation of test data based on a user-defined format are called _____.
   (a) test data generator
   (b) test case generator
   (c) code comprehension
   (d) none of the above
4. The tools that record events at the time of running the system and place the information into a script are called _____.
   (a) test data generator
   (b) test case generator
   (c) coverage analysis tools
   (d) capture playback tools
5. Tools that automate the process of thoroughly testing the software and provide a quantitative measure of the coverage of the system being tested are called _____.
   (a) test data generator
   (b) test case generator
   (c) coverage analysis tools
   (d) capture playback tools
6. WinRunner is a _____ tool.
   (a) test management
   (b) functional/regression testing
   (c) performance testing
   (d) none of the above
7. JMeter is a _____ tool.
   (a) test management
   (b) functional/regression testing
   (c) performance testing
   (d) none of the above
8. TestDirector is a _____ tool.
   (a) test management

(b) functional/regression testing
(c) performance testing
(d) none of the above

9. _____ is not a functional/regression testing tool.
   (a) JMeter
   (b) TestDirector
   (c) WinRunner
   (d) LoadRunner

Review Questions

1. What is the need of automating the testing activities?
2. What is the difference between static and dynamic tools?
3. Analyse and report which tools are available for test planning?
4. Analyse and report which tools are available for test design?
5. What are the guidelines for selecting a testing tool?
6. What are the costs incurred in adopting a testing tool?
7. Is a single testing tool sufficient for all testing activities?
8. Make a list of some important testing tools which are open-source on the Internet. Use them for testing your software.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Part 6

Testing for Specialized Environments

CHAPTERS

Chapter 16:
Testing Object-Oriented Software

Chapter 17:
Testing Web-based Systems

Testing in specialized environments needs more attention with more specialized techniques. This part is devoted to software testing in various specialized environments. Two specialized environments have been considered: testing for object-oriented software and testing for Web-based systems.

The major challenge of testing in specialized environments is that the meaning of different techniques changes according to the environment. For example, unit testing in general loses its meaning in object-oriented testing. Similarly, in Web-based testing, we first need to decide what a unit will be. Thus, using basic testing techniques, we need to explore the issues in current environment and apply these techniques correspondingly.

Some issues arise due to the environment in which the software is currently being developed and tested. For example, in object-oriented software, inheritance testing is a major issue. Therefore, techniques must be devised for inheritance testing. Similarly, in Web-based software, security of application is a major requirement and therefore, security testing becomes mandatory. Thus, the methods for security testing must be known for this testing. In this way, many issues need to be considered while testing specialized-environment software.

This part will make ground for the following concepts:

- Differences between traditional software and specialized-environment software (object-oriented software and Web-based software)
- Challenges in testing object-oriented software and web-based software
- Verification and validation of object-oriented software and Web-based software
- Special issues to be tested due to the specialized environment

Chapter

16

Testing Object-Oriented Software

OBJECTIVES

After reading this chapter, you should be able to understand:

- Issues in object-oriented testing
- Strategy for testing object-oriented software: Verification of UML model and validation
- Issues in testing a class
- Feature-based testing of class
- State-based testing of class
- Issues in testing inheritance
- Incremental testing method for inheritance
- Integration testing of object-oriented software
- Thread-based integration testing
- Implicit control flow-based integration testing
- UML-based testing and system testing based on use-cases

Object-oriented technology has evolved rapidly in the last two decades. Today, everyone is rushing out to adopt this technology. Object-oriented paradigm has affected almost every organization. The object model based on data abstraction, rather than function abstraction, manages the complexity, inherent in many different kinds of systems. This data-abstraction based on objects helps us to model the problem domain in a better understandable format. Thus, object-oriented technology is considered a natural way to understand and represent the problems. Further, this technology has given birth to the concept of reusability in software design. The concept of reusability leads to faster software development and high-quality programs.

Object-oriented technology (OOT) was established with a number of object-oriented programming languages. Today, OO paradigm is embedded in the complete software engineering approach. Software built with OO follows all the steps of *OO software engineering* (OOSE). Modeling, analysis, design, and testing of OO software is different as compared to the structured approach.

Testing an OO software is more challenging. Most of the testing concepts lose their meaning in OO technology, e.g. a unit in OOT is not a module but a class, and a class cannot be tested with input-output behaviour as in unit testing of a module. Thus, both testing strategies and techniques change in case of OO software.

16.1 OOT Basics

The major reason for adopting OOT and discarding the structured approach is the complexity of the software. Structured approach is unable to reduce the complexity as the software size increases, thereby increasing the schedule and cost of the software. On the other hand, OOT is able to tackle the complexity and models the software in a natural way.

In the structured approach, the emphasis is on action, not on data. But data is important for program existence. Global data is a structured approach wherein data can be corrupted by anyone. In OOT, data is hidden, not to be accessed by everyone. Functions in structured approach are not able to model the real world very well. But when we model the same problem with OOT, the modeling is quite obvious and natural. Let us review some of the basic concepts and terminology before discussing the testing issues of OO software.

16.1.1 Terminology

Object Grady Booch defined objects as [36]: 'Objects are abstraction of entities in the real world representing a particularly dense and cohesive clustering of information. It can be defined as a tangible entity that exhibits some well-defined behaviour. Objects serve to unify the ideas of algorithmic and data abstraction'.

Thus, objects are individual entities formed from two components: state and behaviour. The state is modeled by a data structure. The behaviour is modeled by a set of operations (or actions) which manipulate the data structure to obtain the required change of state and result. Together, they form a single coherent unit. The behaviour is invoked by sending a *message* to the object requesting a particular service.

Class All objects with the same structure and behaviour can be defined with a common blueprint. This blueprint called a class, defines the data and behaviour common to all objects of a certain type. Thus class is a template that describes collection of similar objects, e.g. vehicle is a class and car is one object of class vehicle. The operations provided by a class are also known as *features*, and each item of data defined by a class is a *data-member*. Any class that invokes the features of another class is known as a *client* of the class providing the features.

Encapsulation A class contains data as well as operations that operate on data. In this way, data and operations are said to be encapsulated into a single entity. Encapsulation avoids the accidental change in data such that no one can directly access the object's data.

Abstraction Shaw [127] defines abstraction as: 'A simplified description, or specification of a system that emphasizes some of the system's details or properties while suppressing others. A good abstraction is one that emphasizes details that are significant to the user and suppresses details that are immaterial to him'.

Inheritance It is a strong feature of OOT. It provides the ability to derive new class from existing classes. The derived class inherits all the data and behaviour (operations) of the original base class. Moreover, derived class can also add new features, e.g. employee is the base class and we can derive manager as derived class.

Polymorphism This feature is required to extend an existing OO system. Polymorphism means having many forms and implementations of a particular functionality. For example, when a user wants to draw some shape (line, circle, etc.), the system issues a draw command. The system is composed of many shapes, each of them having the behaviour to draw itself. So when the user wants to draw a circle, the circle object's draw command is invoked. By using polymorphism, the system figures out, as it is running, which type of shape is being drawn [37].

16.1.2 Object-Oriented Modeling and UML

Modeling is the method to visualize, specify different parts of a system, and the relationship between them. Modeling basically divides the complex problem into smaller understandable elements and provides a solution to the problem. In object-oriented development, *Unified modeling language* (UML) has become the *de facto* standard for analysis and design of OOS. UML provides a number of graphical tools that can be used to visualize a system from different viewpoints. The views generally defined in it are:

User view This view is from the user's viewpoint which represents the goals and objectives of the system as perceived by the user. It includes use-cases and use-case diagram as a tool for specifying the objectives.

Structural view This view represents the static view of the system representing the elements that are either conceptual or physical. Class diagrams and object diagrams are used to represent this view.

Behavioural view This view represents the dynamic part of the system representing behaviour over time and space. The tools used for behavioral view are: collaboration diagrams, sequence diagrams, state chart diagrams, and activity diagrams.

Implementation view This view represents the distribution of logical elements of the system. It uses component diagrams as a tool.

Environmental view This view represents the distribution of physical elements of the system. It depicts nodes which are part of the physical hardware for deployment of the system. It uses deployment diagram as a tool.

Let us review all the graphical tools used in the UML modeling. It is necessary to understand them as you need to verify and validate the OO software development through these models.

Use-case model Requirements for a system can be elicited and analysed with the help of a set of scenarios that identify how to use the system. The following components are used for this model:

Actors An actor is an external entity or agent that provides or receives information from the system. Formally, an actor is anything that communicates with the system or product and that is external to the system itself. An actor is represented as a stick diagram.

Use-case A use-case is a scenario that identifies and describes how the system will be used in a given situation. Each use-case describes an unambiguous scenario of interactions between actors and the system. Thus, use-case define functional and operational requirements of the system. A use-case is represented as

A complete set of use-cases describe all the means to use a system. Use-cases can be represented as use-case templates and use-case diagrams.

Use-case templates They are prepared as structured narrative mentioning the purpose of use-case, actors involved, and the flow of events.

Use-case diagrams These show the visual interaction between use-cases and actors.

Class diagrams A class diagram represents the static structure of the system using classes identified in the system with associations between them. In other words, it shows the interactions between the classes in the system.

Object diagrams An object diagram represents the static structure of the system at a particular instance of time. It shows a set of objects and their relationships at a point of time.

Class-responsibility-collaboration (CRC) diagrams This diagram represents a set of classes and the interaction between them using associations and messages sent and received by the classes. These diagrams are used to understand the messages exchanged between the classes.

CRC model index card This card represents a class. The card is divided into three sections:

- On the top of the card, name of class is mentioned.
- In the body of the card on the left side, responsibilities of the class are mentioned.
- In the body of the card on the right side, the collaborators (relationship between classes) are mentioned.

Sequence diagram It is a pictorial representation of all possible sequences along the time-line axis to show the functionality of one use-case, e.g. for the use-case *Deposit the money in a bank*, there are sequences of events–customer goes to cashier with cash and form, submits it to the cashier, cashier receives the cash and form, deposits the amount and updates the balance, returns receipt to the customer.

State chart diagrams These diagrams represent the states of reactive objects (which respond to external events) and transitions between the states. A state chart depicts the object's initial state where it performs some activity and transitions after which the object enters another state. Thus, these diagrams represent the behaviour of objects with respect to their states and transitions.

Activity diagrams It is a special case of a state chart diagram, wherein states are activity states and transitions are triggered by the completion of activities.

Component diagrams At the physical level, the system can be represented as components. The components can be in the form of packages or sub-systems containing classes. The component diagrams show the set of components and their relationships.

Deployment diagrams These diagrams are also at the physical level. After the components and their interaction have been identified in the component diagram, the components are deployed for execution in the actual hardware nodes identified according the implementation. Thus, these diagrams depict the mappings of components to the nodes involved in the physical implementation of a system.

16.2 Object-Oriented Testing

It is clear that object-oriented software is different in many ways as compared to the conventional procedural software. OO software is easy to design but difficult to test. As dimensions of OOS increase in comparison to procedural software, the testing of OOS should also be broadened such that all the properties of OOT are tested.

16.2.1 Conventional Testing and OOT

Firesmith [67] has examined the differences between conventional testing and object-oriented software testing in some depth. The difference between unit testing for conventional software and object-oriented software arises from the nature of classes and objects. We know that unit testing only addresses the testing of individual operations and data. This is only a subset of the unit testing for object-oriented software, as the meaning and behaviour of the resources encapsulated in the classes depend on other resources with which they are encapsulated. Integration testing (for conventional software) is truly the unit testing of object-oriented software.

The object-oriented paradigm lowers the complexity and hence, lessens the need for basis path testing. In a nutshell, there are differences between how a conventional software and an object-oriented software should be tested.

16.2.2 Object-Oriented Testing and Maintenance Problems

It is clear from the above discussion that it is not easy to understand and test the OO software. Moreover, it also becomes difficult to maintain it. David *et al.* [68] have recognized some of these problems which are discussed here:

Understanding problem The encapsulation and information hiding features are not easily understood the very first time. There may be a chain of invocations of member functions due to these features. A tester needs to understand the sequence of this chain of invocations and design the test cases accordingly.

Dependency problem There are many concepts in OO that gives rise to the dependency problem. The dependency problem is mainly due to complex relationships, e.g. inheritance, aggregation, association, template class instantiation, class nesting, dynamic object creation, member function invocation, polymorphism, and dynamic binding relationships. These relationships imply that one class inevitably depends on another class. For example, the inheritance relationship implies that a derived class reuses both the data members and the function members of a base class; and hence, it is dependent on the base class.

The complex relationships that exist in an OOS make testing and maintenance extremely difficult in the following way:

(i) Dependency of the classes increases the complexity of understanding and hence testing.

(ii) In OO software, the stub development is more difficult as it needs to understand the chain of called functions, possibly create and properly initialize certain objects, and write their code.

(iii) Template class feature is also difficult to test as it is impossible to predict all the possible uses of a template class and to test them.

(iv) Polymorphism and dynamic binding features add to the challenge of testing them as it is difficult to identify and test the effect of these features.

(v) As the dependencies increase in OOS, it becomes difficult to trace the impact of changes, increasing the chances of bugs, and hence making testing difficult.

State behaviour problem Objects have states and state-dependent behaviours. That is, the effect of an operation on an object also depends on the state of the object and may change its state. Thus, combined effect of the operations must be tested.

16.2.3 Issues in OO Testing

Based on the problems discussed for testing an OO software, the following are some issues which come up while performing testing on OO software.

Basic unit for testing There is nearly universal agreement that class is the natural unit for test case design. Other units for testing are aggregations of classes: class clusters, and application systems. The intended use of a class implies different test requirements, e.g., application-specific vs general-purpose classes, abstract classes, and parameterized (template) classes. Testing a class instance (an object) can verify a class in isolation. However, when verified classes are used to create objects in an application system, the entire system must be tested as a whole before it can be considered verified.

Implications of inheritance In case of inheritance, the inherited features require retesting because these features are now in, new context of usage. Therefore, while planning to test inheritance, include all inherited features. Moreover, planning should also consider the multiple contexts of usage in case of multiple inheritances.

Polymorphism In case of polymorphism, each possible binding of a polymorphic component requires a separate test to validate all of them. However, it may be hard to find all such bindings. This again increases the chance of errors and poses an obstacle in reaching the coverage goals.

White-box testing There is also an issue regarding the testing techniques for OO software. The issue is that the conventional white-box techniques cannot be adapted for OO software. In OO software, class is the major component to be tested, but conventional white-box testing techniques cannot be applied on testing a class. These techniques may be applicable to individual methods but not suitable for OO classes.

Black-box testing The black-box testing techniques may be suitable to OO software for conventional software, but sometimes re-testing of inherited features require examination of class structure. This problem also limits the black-box methods for OO software.

Integration strategies Integration testing is another issue which requires attention and there is need to devise newer methods for it. However, there is no obvious hierarchy of methods or classes. Therefore, conventional methods of incremental integration testing cannot be adopted.

16.2.4 Strategy and Tactics of Testing OOS

The strategy for testing OOS is the same as the one adopted earlier for procedural software, i.e. verification and validation. However, tactics for testing may differ depending upon the structure of OOS.

Object-oriented software testing is generally done bottom-up at four levels:

Method-level testing It refers to the internal testing of an individual method in a class. Existing testing techniques for conventional programming languages are fully applicable to method-level testing.

Class-level testing Methods and attributes make up a class. Class-level (or intra-class) testing refers to the testing of interactions among the components of an individual class.

Cluster-level testing Cooperating classes of objects make up a cluster. Cluster-level (or inter-class) testing refers to the testing of interactions among objects.

System-level testing Clusters make up a system. System-level testing is concerned with the external inputs and outputs visible to the users of a system.

16.2.5 Verification of OOS

The verification process is different in analysis and design steps. Since OOS is largely dependent on OOA and OOD models, it becomes important to verify them. For example, if there is problem in the definition of class attributes and is uncovered during analysis, then this problem will propagate in design and other phases too.

OOA is not matured enough in one iteration only. So it requires many iterations for a reliable analysis model. Verification can play an important role in these iterations of OOA. Each iteration should be verified so that we know that the analysis is complete. For example, consider a class in which a number of attributes are defined during the first iteration, and an attribute may be appended in another iteration due to some misunderstanding. If there is verification of OOA at each iteration, then this error may be pointed out in a review and thereby avoid problems. Thus, all OO models must be tested for their correctness, completeness, etc.

Verification of OOA and OOD models

- Assess the class-responsibility-collaborator (CRC) model and object-relationship diagram.
- Review system design (examine the object-behaviour model to check mapping of system behaviour to subsystems, review concurrency and task allocation, use use-case scenarios to exercise user interface design).
- Test object model against the object relationship network to ensure that all design objects contain necessary attributes and operations needed to implement the collaborations defined for each CRC card.
- Review detailed specifications of algorithms used to implement operations using conventional inspection techniques.

Verification points of some of the models are given below:

- Check that each model is correct in the sense that proper symbology and modeling conventions have been used.
- Check that each use-case is feasible in the system and unambiguous such that it should not repeat operation.
- Check that a specific use-case needs modification in the sense that it should be divisible further in more use-cases.
- Check that actors identified in use-case model are able to cover the functionality of the whole system. If any new actor is needed to fulfill the requirement, then add it.

- All interfaces between actors and use-cases in use-case diagram must be continuously checked such that there should be no inconsistency.
- All events and their sequences must be checked iteratively in a sequence diagram. If any detail of the operation is missing, then events can be added or deleted if required.
- Collaboration diagrams must be verified for distribution of processing between objects. Suppose the collaboration diagram was shaped like a star, with several objects communicating with a central object. Analyst may conclude that the system is too dependent on the central object and redesign the objects to distinguish the processing power more evenly [37].
- Each identified class must be checked against defined selection characteristics such that it should be unique and unambiguous.
- All collaborations in class diagrams must be verified continuously such that there should be no inconsistency.
- Check the description of each CRC index card to determine the delegated responsibility.

16.2.6 Validation Activities

All validation activities described earlier are also implemented in OOS. But the meaning of unit and integration testing changes due to the nature of OOS. Therefore, new methods or techniques are required to test all the features of OOS.

Unit/Class Testing

Unit or module testing has no meaning in the context of OOS. Since the structure of OOS is not based on action, module testing has lost its meaning. OOS is based on the concept of class and objects. Therefore, the smallest unit to be tested is a class. The nature of OOS is different as compared to procedural software; therefore many issues rise for testing a class. These issues are discussed in the next section.

Issues in testing a class

- A class cannot be tested directly. First, we need to create the instances of a class as object and then test it. Thus, an abstract class cannot be tested, as its objects cannot be created.
- Operations or methods in a class are the same as functions in traditional procedural software. One idea is that they can be tested independently using the previous white-box or black-box methods. But in the context

of OOS, this isolated testing has no meaning. Thus the methods should also be tested with reference to a class or class hierarchy in the direction of complete testing of a class.

- One important issue in class testing is the testing of inheritance. If some classes have been derived from a base class, then independent base class testing is not sufficient. All derived classes in the hierarchy must also be tested depending on the kind of inheritance. Thus, the following special cases also apply to inheritance testing:
  (a) Abstract classes can only be tested indirectly (because it can't be instantiated). Hence, abstract classes have to be tested by testing instance of its descendant specialized classes.
  (b) Special tests have to be written to test the inheritance mechanism, e.g. a child class can inherit from a parent and that we can create instance of the child class.
- There may be operations in a class that change the state of the object on their execution. Thus, these operations should be tested with reference to state change of object.
- Polymorphism provides us with elegant compact code, but it can also make testing more complicated. The identification of all the paths through a section of code that uses polymorphism is harder because the criteria which indicate different paths are not specifically mentioned in the code.

16.2.7 Testing of OO classes

Feature-based Testing of Classes

There are a number of code complexity metrics available which vary from simple ones such as the number of the source lines of code, to the McCabe cyclomatic complexity metric. These metrics can be used to measure the complexity of the individual features of a class providing the tester with an accurate way of determining which features to consider first for testing. The features of a class can be categorized into six main groups [69, 70]:

Create These are also known as *constructors*. These features perform the initial memory allocation to the object, and initialize it to a known state.

Destroy These are also known as *destructors*. These features perform the final memory de-allocation when the object is no longer required.

Modifiers The features in this category alter the current state of the object, or any part thereof.

Predicates The features in this category test the current state of the object for a specific instance. Usually, they return a BOOLEAN value.

Selectors The features in this category examine and return the current state of the object, or any part thereof.

Iterators The features in this group are used to allow all required sub-objects to be visited in a defined order.

Testing feature groups While testing the classes with features as discussed above, there may be some preferences of these feature groups according to which one is tested first. Turner and Robson [71] have provided the following guidelines for the preferred features to be tested:

(i) The *create* features of a class must be verified first, as these features are for the initialization of every object created. Therefore, they must be tested rigorously.

(ii) The *selector* features can be the next features to be verified as they do not alter the state of the object. To test them, alter the state of the object either by using the appropriate constructor (if available), or by directly altering the state of the object.

(iii) The *predicate* features of a class test for specific states of the object and therefore while constructing the test cases, care must be taken to place the object in the specific state for the required test result.

(iv) The *modifier* features of a class are used to alter the state of the object from one valid state to another. To detect if the transition was to the correct state, the selectors and predicate features should be used where possible.

(v) The *destructor* feature of a class performs any 'housekeeping', for example, returning any allocated memory to the heap. To test them, features which alter the state of the object and allocate memory are used to create the scenarios for the test cases, then the destructor is called.

Role of Invariants in Class Testing

If we are able to specify the valid states and in valid states for an object, it can help in testing the classes. The language Eiffel provides the facility of specifying the invariants. But, by providing the features for testing the validity of the object's current state, you can also achieve this ability. Turner and Robson [71] have also shown the importance of invariants in class testing.

For example, take a class to implement an integer Stack as:
Class Stack with following data members:

ItemCount, empty, full, push, pop, max_size

Here in this class, invariants can be defined as given below:

1. ItemCount cannot have a negative value.
2. There is an upper bound check on ItemCount, i.e. ItemCount ≤ max-size
3. Empty stack, i.e. ItemCount = 0
4. Full Stack, i.e. ItemCount = max_size

All invariants discussed above show the states of the class. These invariants allow the validity of the current state of the object to be tested easily. An invariant is also useful for ensuring that the data representation is in a valid state, even if it has been directly altered. This can help maintain the integrity of the class.

State-Based Testing

State-based testing can also be adopted for testing of object-oriented programs. The systems can be represented naturally with finite state machines as states correspond to certain values of the attributes and transitions correspond to methods. The features of a class provide the desired behaviour by interacting with the data-representation. Although these interactions are found in programs written in traditional languages, they are more visible and prevalent in object-oriented programming languages. State-based testing tests these interactions by monitoring the effects that features have on the state of an object. The state can be determined by the values stored in each of the variables that constitute its data representation. Each separate variable is known as a *substate*, and the values it may store are known as *substate-values.*

The restrictions on objects that are to be represented as state machines include [71]:

- The behaviour of the object can be defined with a finite number of states.
- There are a finite number of transitions between the states.
- All states are reachable from the initial state.
- Each state is uniquely identifiable through a set of methods.

State-based testing based on the *finite state automata*, tests each feature with all its valid input states. However, it may not be possible to use every single value for each variable of the data-representation due to their potentially large

range. Therefore, the categorization is applied in order for a suitable reduction to be achieved.

For designing the test cases, we need to have stimuli. Stimuli are particular values, obtained after analysing the feature's parameters, and have specific effects on the response of the feature, or are significant. The test cases then consist of applying each stimulus in combination with each possible state for the representation. The resultant states generated are then validated against those expected. The behaviour of an object can be predicted by matching the input states expected by one feature with the output states of another. This validation should ensure the correct operation of the class with no concern for the order in which the features are invoked.

Process of state-based testing The process of using state-based testing techniques is as follows:

1. Defines the states.
2. Define the transitions between states.
3. Define test scenarios.
4. Define test values for each state.

To see how this works, consider the Stack example again.

Define states The first step in using state-based testing is to define the states. We can create a state model with the following states in the Stack example:

1. Initial: before creation
2. Empty: ItemCount = 0
3. Holding: ItemCount > 0, but less than the max capacity
4. Full: ItemCount = max
5. Final: after destruction

Define transitions between states The next step is to define the possible transitions between states and to determine what triggers a transition from one state to another. Generally, when we test a class, a transition is triggered when a method is invoked. Thus, some possible transitions could be defined as follows:

1. **Initial → Empty:**

   Action = "create" e.g. "s = new Stack()" in Java
2. **Empty → Holding:**

   Action = "add"
3. **Empty → Full:**

Action = "add" if max_size= 1

4. **Empty → Final:**

   Action = "destroy" e.g. destructor call in C++, garbage collection in Java

5. **Holding → Empty:**

   Action = "delete"

Thus, for each identified state, we should list the following:

1. Valid actions (transitions) and their effects, i.e., whether or not there is a change of state.
2. Invalid actions that should produce an exception.

Define test scenario Tests generally consist of scenarios that exercise the object along a given path through the state machine. The following are some guidelines:

1. Cover all identified states at least once.
2. Cover all valid transitions at least once.
3. Trigger all invalid transitions at least once.

Test cases are designed such that they exercise operations that change state and those that do not change state separately. For instance,

Test case: Create-Add-Add-Delete-Add-delete-destroy

It is also needed to check the state of the object that is being tested throughout the scenario to ensure that the theoretical state model you have defined is actually the one implemented by the class you are testing.

Define test values for each state Finally, choose test values for each individual state. The consideration in choosing the values is that choose unique test values and do not reuse values that were used previously in the context of other test cases. This strategy provides more diversity in the test suite and increases the likelihood of detecting bugs.

16.2.8 Inheritance Testing

The issues in testing the inheritance feature are discussed below.

Superclass modifications When a method is changed in superclass, the following should be considered:

(i) The changed method and all its interactions with changed and unchanged methods must be retested in the superclass.

(ii) The method must be retested in all the subclasses inheriting the method as extension.

(iii) The 'super' references to the method must be retested in subclasses.

Inherited methods The following inherited methods should be considered for testing:

(i) *Extension:* It is the inclusion of superclass features in a subclass, inheriting method implementation and interface (name and arguments).

(ii) *Overriding:* It is the new implementation of a subclass method, with the same inherited interface as that of a superclass method.

Reusing of test suite of superclass It may not be possible to reuse superclass tests for extended and overridden methods due to the following reasons:

(i) Incorrect initialization of superclass attributes by the subclass.

(ii) Missing overriding methods.

(iii) Direct access to superclass fields from the subclass code can create a subtle side.

(iv) A subclass may violate an invariant from the superclass or create an invalid state.

Addition of subclass method When an entirely new method is added to a specialized subclass, the following should be considered:

(i) The new method must be tested with method-specific test cases.

(ii) Interactions between the new method and the old methods must be tested with new test cases.

Change to an abstract superclass interface 'Abstract' superclass is the class wherein some of the methods have been left without implementation; just the interface has been defined. In this case, all the subclasses must be retested, even if they have not changed.

Interaction among methods To what extent should we exercise interaction among methods of all superclasses and of the subclass under test?

Thus, inherited methods should be retested in the context of a subclass.

Example: If we change some method m() in a superclass, we need to retest m() inside all subclasses that inherit it.

Example: If we add or change a subclass, we need to retest all methods inherited from a superclass in the context of the new/changed subclass.

Inheritance of Invariants of Base Class

If an invariant is declared for a base class, the invariant may be extended by the derived class. Additional clauses may be added for new data-members

declared by the derived class. However, the data-representation of the parent class still complies with the original invariant and therefore does not require retesting.

If the invariant provided by the base class is retracted, then the base class must be retested with respect to the new invariant. A retraction involves the loosening of clauses, allowing the data members of the parent class to accept states that were previously not allowed. The retesting can be performed by re-executing all the original test cases, along with some additional test cases to cover the widened invariant. By clever analysis of the widened invariant, it may be possible to reduce the number of test cases from the original suite of test cases that need to be re-executed.

Incremental Testing

Wegner and Zdonik [73] proposed this technique to test inheritance. The technique first defines inheritance with an incremental hierarchy structure. Then this approach is used to test the inheritance of classes. This technique is based on the approach of a modifier to define a derived class. A *modifier* is first applied on the base class that alters its attributes. The modifier and the base class, along with the inheritance rules for the language, are used to define the derived class. We use the composition operator $\oplus$ to represent this uniting of base class and modifier, and we get

$$D = B \oplus M$$

where D is derived class, B is base class, and M is a modifier. This can be seen in Fig. 16.1.

Figure 16.1 Deriving class D from base class B using modifier M

The derived class designer specifies the modifier, which may contain various types of attributes that alter the base class to get the derived class. The following attributes have been identified:

- New attribute
- Inherited attribute
- Redefined attribute
- Virtual-new attribute

- Virtual-inherited attribute
- Virtual-redefined attribute

All the above mentioned attributes have been illustrated in the Fig. 16.2.

After defining the inheritance as an incremental hierarchical approach, now we will use this technique to test the inheritance. This is known as the incremental class technique for testing inheritance. To test the derived class, the issue is to find out which attributes and interactions between them need to be tested. This can be managed easily if we divide it into two categories: *Testing of features* (TF) and *testing of interactions* (TI). With this information for every class, the idea of incremental testing is that initially, the base class will be tested by testing each feature and interactions between them. But, their execution is maintained in a testing history of test cases. This testing history will be reused for the features which are being inherited without any modification and unchanged context, thereby reducing the effort being made in testing everything in the inherited subclasses. Only new features or those inherited-affected features and their interactions are tested as added efforts. The following steps describe the technique for incremental testing:

1. Test the base class by testing each member function individually and then testing the interactions among the member functions. All the testing information is saved in a testing history. Let it be *TH*.
2. For every derived class, determine the modifier *m* and make a set *M* of all features for all derived classes.
3. For every feature in *M*, look for the following conditions and perform testing:
   (i) If feature is new or virtual-new, then test the features and the interactions among them as well.
   (ii) If feature is inherited or virtual-inherited without any affect, then test only the interactions among the features. If there is a need to test the features again in the derived class, then the test history *TH* of base class can be reused for the test cases.
   (iii) If feature is redefined or virtual-redefined, then test the features and the interactions among them as well. For testing, use the test cases from testing history *TH* of the base class. However, if the redefined feature is affected somewhere or there is need to interact with new attribute, new test cases should be designed.

The benefit of this technique is that we don't need to analyse the derived classes for what features must be tested and what may not be tested, thereby reducing the time required in analysing and executing the test cases.

```
Class B
{
    float x;
    float y;
public:
    B() {}
    void method1(
float a, float b)
    {
    x=a; y=b;
    }

virtual float method2()
    {
    return x;
    }

    float method3()
    {
    return y;
    }

    float method4()
    {
    return method2();
    }

virtual float method5()
    {
    return x;
    }
};
```

```
Class D: public B
{
private:
int i;
public:
   D() {}
   void method1(int a)
     {
     i = a + 5;
     }

virtual float method2()
{
return 4*B::method2();
}

float method3()
{
return 6*B::method3();
}

virtual int method6()
{
return i;
}
};
```

```
D's attributes after the mapping

private:
int i;  //New
public:
D() {}  //New

void method1(float a, float b)
//Inherited
    {
    x=a; y=b;
    }

    void method1(int a) //New
    { i = a + 5; }

virtual float method2()
// virtual redefined
{
return 4*B::method2();
}

float method3() //redfined
{
return 6*B::method3();
}

float method4() //inherited
{
return method2();
}

virtual float method5()
//virtual inherited
{
return x;
}

virtual int method6()
//virtual new
{
return i;
}
```

Figure 16.2 Incremental modification technique

16.2.9 Integration Testing

Different classifications of testing levels have been proposed with the following terminology:

Inter-class testing This type of testing includes testing of any set of cooperating classes, aimed at verifying that the interaction between them is correct. There are no constraints on how these classes are selected.

Cluster testing This type of testing includes testing of the interactions between the different classes belonging to a subset of the system, having some specific properties called a *cluster*. Usually, a cluster is composed of cooperating classes providing particular functionalities (e.g., all the classes which can be used to access the file-system, or the classes composing a Java package). Clusters should provide a well-defined interface, i.e. their interfaces should be well-understood and they should mutually interact only by means of their interfaces.

Inter-cluster testing This type of testing includes testing of the interactions between already tested clusters. The result of the integration of all clusters is the whole system.

Thread-based Integration Testing

As long as we consider object-oriented systems as sets of cooperating entities exchanging messages, threads can be naturally identified with sequences of subsequent message invocations. Therefore, a thread can be seen as a scenario of normal usage of an object-oriented system. Testing a thread implies testing the interactions between classes according to a specific sequence of method invocations. This kind of technique has been applied by several authors. Kirani and Tsai [77] propose a technique for generating test cases from functional specification for module and integration testing of object-oriented systems. The method aims at generating test cases that exercise specific combinations of method invocations and provides information on how to choose classes to be integrated.

A similar approach is proposed by Jorgensen and Erickson [78], who introduce the notion of *method-message path* (*MM-path*), defined as a sequence of method executions linked by messages. For each identified MM-path, integration is performed by pulling together classes involved in the path and exercising the corresponding message sequence. Thus, integration level thread is a MM-path. More precisely, Jorgensen and Erickson identify two different levels for integration testing:

Message quiescence This level involves testing a method together with all methods it invokes, either directly or transitively.

Event quiescence This level is analogous to the message quiescence level, with the difference that it is driven by system-level events. Testing at this level

means exercising message chains (threads), such that the invocation of the first message of the chain is generated at the system interface level (i.e., the user interface) and, analogously, the invocation of the last message results in an event which can be observed at the system interface level. An end-to-end thread is called an *atomic system function* (*ASF*). Therefore, a system-level thread is a sequence of ASFs. ASFs have port events as their inputs and outputs, and thus a sequence of ASFs implies an interleaved sequence of port input and output events. The main drawback of this method is the difficulty in the identification of ASFs, which requires either the understanding of the whole system or an analysis of the source code.

Implicit Control Flow-based Integration Testing

The integration testing should also consider the methods invocations where the call is not obvious in the source code. For example, the C++ expression "x*y" for object *x* and *y*. In this case, there is a call to a method *via* operator overloading for the multiplication of two objects. Thus, these types of method invocations also need to be considered in integration testing. This is known as implicit control flow-based integration testing [74].

There are three approaches for this type of integration: *optimistic, pessimistic, and balanced.* To understand all three approaches, let us take an example as shown in Fig. 16.3. This example illustrates the very common example of object-oriented feature where we want to calculate the salary of employees of many categories, e.g. worker, assistant, and manager using the common method CalSalary(). The CalSalary() will be run for which employee is known at run time, i.e. it is a dynamically bound method. In this example, assume that employee is a base class; worker, assistant, and manager are three derived classes from employee. Each derived class uses the method Calsalary() with different computations. But at run time, it will be decided which CalSalary() method is invoked. Thus, method invocations are not obvious. In this case, we will use the following three integration approaches as discussed below.

Figure 16.3 Implicit control flow example

Optimistic approach This approach is based on the optimistic view that method invocation will be confined to the level of abstraction of the source code only. Therefore, in this type of integration we will test only those method invocation or interfaces which support this abstraction as shown in Fig. 16.4.

Figure 16.4 Optimistic approach to implicit control flow

Pessimistic approach This approach assumes that every class can call the common method CalSalary. Therefore, all interfaces of calling the method are tested here as shown in Fig. 16.5. However, the danger in this type of integration testing is the increase in the complexity in testing.

Figure 16.5 Pessimistic approach to implicit control flow

Balanced approach This approach is a balance between the above two approaches. We must take the interfaces according to the code as well as expected method invocations. But, here we take a mid approach that all combinations of interfaces in the pessimistic approach need not be tested. Instead, only one or two classes should test source and some combinations of interfaces will be tested such that other interfaces are likely to work properly. Therefore, we can make an equivalence set of interfaces which will be tested and others will be assumed to be working correctly as shown in Fig. 16.6.

16.2.10 UML-Based OO Testing

UML is a design and modeling language for object-oriented software and is widely used as a *de facto* standard that provides techniques to model the software from different perspectives. It supports facilities both for abstract high-

level modeling and for more detailed low-level modeling. It consists of a variety of graphical diagram types that can be extended for specific application areas. It is associated with a formal textual language, *OCL* (Object Constraint Language). UML also provides support for model-based testing.

Figure 16.6 Balanced approach to implicit control flow

UML diagrams in software testing

The following UML diagrams are helpful in the testing activities mentioned below:

- *Use-case diagrams*: testing of system-level functional requirements, acceptance testing
- *Class diagrams*: class (module / unit) testing, integration testing
- *Sequence diagrams, collaboration diagrams*: integration testing, testing of control and interaction between objects, testing of communication protocols between (distributed) objects
- *Activity diagrams*: testing of work flow and synchronization within the system, white-box testing of control flow
- *State diagrams* (state charts): state-based testing
- *Package diagrams*, *component diagrams*: integration testing
- *Deployment diagrams*: system testing

System Testing based on Use-Cases

Use-case In general, a use-case is a sequence of interactions by which the user accomplishes a task in a dialogue with the system. When we use use-case in testing, we follow the following:

- use-case = one particular way to use the system
- use-case (in testing) = user requirement
- set of all use-cases = complete functionality of the system
- set of all use-cases (in testing) = interface between the users (*actors*) and the system

Scenario It is an instance of a use-case, expressing a specific task, by a specific actor, at a specific time, and using specific data.

Use-case model This is a set of use-case diagrams, each associated with a textual description of the user's task. For testing, the use-case model must be extended with:

- the domain of each variable participating in the use-cases,
- the input/output relationships among the variables,
- the relative frequency of the use-cases,
- the sequential (partial) order among the use-cases.

Thus, use, case-based testing is a technique for generating test cases and recommended configurations for system-level testing. The testers build a test model based on the standard UML notions of use-cases, actors, and the relationships between these elements. The use-cases are enhanced with additional information, including inputs from actors, outputs to the actors, and how the use-case affects the state of the system.

To perform this testing, the tester needs to identify four things:

- the use-cases of interest,
- the actors involved in using the system,
- the input, output, and system effects for the use-cases,
- the flows of interest between the use-cases.

A use-case is a semantically meaningful function that provides some value from the user's point of view. For example, saving a file in a word processing system would be represented by the Save File use-case. Each use-case can have input parameters associated with it, and for each parameter, a set of logical partitions of the values that parameter can take may be identified. Finally, the use-cases can be connected using flows that describe a sequence of use-case that are performed to accomplish some goal.

SUMMARY

Object-oriented software is different in nature from procedural software. Its origin is based on reducing the complexity of the software development. However, this type of environment adds many issues that need to be resolved while testing it. This chapter discusses the issues regarding OO technology and techniques for testing the OO software. The meaning of unit is not the same as seen in the procedural software. Similarly, there is no obvious hierarchy as observed in procedural software; therefore top-down and bottom-up integration testing cannot be adopted straightaway. Inheritance is another issue which needs attention while testing, otherwise we will end up with heavy testing of base class and derived classes. The techniques to deal with the unit, integration, and system testing have been discussed.

The UML model is another big support for testing the OO software. Since OO software can be developed with the help of UML models in the form of graphical tools, these also help in testing the OO software at various levels. The chapter also discusses them and elaborates system testing with use-case model.

Let us review the important concepts described in this chapter:

- Object-oriented software is not easy to test as it is different in nature from procedural software.
- The encapsulation and information hiding features are not easily understood the very first time.
- There is dependency problem in OOS due to complex relationships, e.g. inheritance, aggregation, association, template class instantiation, class nesting, dynamic object creation, member function invocation, polymorphism, and dynamic binding relationships. These relationships imply that one class inevitably depends on another class.
- The natural unit for test case design in OOS is a class.
- The conventional white-box testing techniques cannot be applied on testing a class.
- Object-oriented software testing is generally done bottom-up at four levels: Class-level, method-level, cluster-level, and system-level.
- Since OOS is largely dependent upon OOA and OOD models, it becomes important to verify them. Thus, the verification of OOS is largely to verify OOA and OOD models.
- There are features of every class which can be prioritized for testing OO classes. This is known as feature-based testing of OO classes.
- State-based testing can also be adopted for the testing of object-oriented programs.
- Incremental testing is the technique to test the inheritance wherein inheritance is defined as an incremental hierarchy structure.
- Integration testing of OOS is done at the following levels: inter-class level, cluster-level, and inter-cluster level.
- Thread-based testing is the integration testing method wherein testing a thread implies testing interactions between classes according to a specific sequence of method invocations.
- Implicit control flow-based testing is the integration testing wherein we consider methods invocations where the call is not obvious in the source code. There are three approaches for this type of integration: optimistic, pessimistic, and balanced.
- Use, case-based testing is a technique for generating test cases and recommended configurations for system, level testing. The testers build a test model based on the standard UML notions of use-cases, actors, and the relationships between these elements.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Encapsulation is _____.
   (a) the ability to derive new class from existing classes

(b) avoiding the accidental change in data such that no one can directly access the object's data

(c) having many forms and implementations of a particular functionality

(d) none of the above

2. Inheritance is _____.

(a) the ability to derive new class from existing classes

(b) avoiding the accidental change in data such that no one can directly access the object's data

(c) having many forms and implementations of a particular functionality

(d) none of the above

3. Polymorphism is _____.

(a) the ability to derive new class from existing classes

(b) avoiding the accidental change in data such that no one can directly access the object's data

(c) having many forms and implementations of a particular functionality

(d) none of the above

4. A use-case is a _____.

(a) scenario

(b) test case

(c) an actor

(d) all of the above

5. Use-case diagrams show the _____.

(a) visual interaction between use-cases and test cases

(b) visual interaction between use-cases and classes

(c) visual interaction between use-cases and objects

(d) visual interaction between use-cases and actors

6. Dependency problem is caused by _____.

(a) encapsulation and information hiding features

(b) state dependent behaviours

(c) the complex relationships that exist in an OOS

(d) none of the above

7. Understanding problem is caused by _____.

(a) encapsulation and information hiding features

(b) state dependent behaviours

(c) the complex relationships that exist in an OOS

(d) none of the above

8. State behaviour problem is caused by _____.
   (a) encapsulation and information hiding features
   (b) state dependent behaviours
   (c) the complex relationships that exist in an OOS
   (d) none of the above
9. Method-level (or unit) testing refers to _____.
   (a) testing of interactions among the components of an individual class
   (b) the internal testing of an individual method in a class
   (c) testing of interactions among objects
   (d) external inputs and outputs visible to the users of a system
10. Class-level testing refers to _____.
   (a) testing of interactions among the components of an individual class
   (b) the internal testing of an individual method in a class
   (c) testing of interactions among objects
   (d) external inputs and outputs visible to the users of a system
11. Cluster-level testing refers to _____.
   (a) testing of interactions among the components of an individual class
   (b) the internal testing of an individual method in a class
   (c) testing of interactions among objects
   (d) external inputs and outputs visible to the users of a system
12. System-level testing refers to _____.
   (a) testing of interactions among the components of an individual class
   (b) the internal testing of an individual method in a class
   (c) testing of interactions among objects
   (d) external inputs and outputs visible to the users of a system
13. The features of a class can be categorized into _____ main groups.
   (a) 3
   (b) 5
   (c) 2
   (d) 6
14. _____ feature must be tested first in feature-based testing.
   (a) Destroy
   (b) Predicate
   (c) Modifiers
   (d) Create
15. _____ feature is an ideal candidate for use in the validation of other features.
   (a) Destroy

(b) Predicate
(c) Selectors
(d) Create

16. The modifier features of a class are used to _____.
    (a) alter the state of the object from one valid state to another valid state.
    (b) alter the state of the class from one valid state to another valid state.
    (c) all of the above
17. Inherited methods should be retested in the context of a _____.
    (a) superclass
    (b) subclass
    (c) objects
    (d) all of the above
18. System testing can be performed with _____.
    (a) deployment diagrams
    (b) class diagram
    (c) package diagram
    (d) use case diagram
19. Sequence diagrams are used for _____.
    (a) unit testing
    (b) integration testing
    (c) state testing
    (d) none of the above
20. Package diagrams are used for _____.
    (a) unit testing
    (b) integration testing
    (c) state testing
    (d) none of the above

Review Questions

1. What is the major reason for discarding the structured approach?
2. Differentiate an object and a class with example.
3. What is the difference between testing a procedural software and an object-oriented software?
4. What are the testing and maintenance problems introduced with object-oriented software?
5. What steps would you take to verify a use-case model?

6. Take an OOS project and verify its use-case model.
7. What steps would you take to verify a CRC model?
8. Take an OOS project and verify its CRC model.
9. What are the issues in testing a class?
10. Make a software to calculate the salary of employees in an organization, assuming the necessary details required using the object-oriented technology and prepare the following:
    (a) Use-case diagram
    (b) Sequence diagram
    (c) Collaboration diagram
    (d) CRC model
    (e) Class diagram
    (f) Verify use-case model
    (g) Verify CRC model
11. Perform feature-based testing on the classes identified in Question 10.
12. What is the role of invariants in class testing? Discuss with example.
13. List the inheritance issues in testing the classes of Question 10.
14. Identify inheritance relationship in the class identified in Question 10 and show example of every attribute in the incremental testing. Perform incremental testing on the classes mentioning all the attributes.
15. What are the integration testing levels of an OOS?
16. What is the procedure for performing thread-based integration testing?
17. Perform thread-based integration on the classes identified in Question 10.
18. What is the idea behind implicit control flow integration testing?
19. Is it possible to perform implicit control flow integration testing on the classes identified in Question 10.
20. Which UML diagrams are helpful in testing an OOS?
21. Perform system testing based on the use-cases identified in use-case model prepared in Question 10.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter

17

Testing Web-based Systems

OBJECTIVES

After reading this chapter, you should be able to understand:

- How the testing of a web-based system is different from testing traditional software
- Challenges in testing web-based systems
- Web engineering based analysis and design models which is the basis for testing web-based systems
- Interface testing
- Usability testing
- Content testing
- Navigation testing
- Configuration/Compatibility testing
- Security testing
- Performance testing
- Load testing
- Stress testing

Web-based systems have had a great impact on our daily lives. These systems have evolved from small website add-ons to large multi-tiered applications. The computing environment for this rapidly growing web technology is complex as Internet is heterogeneous, distributed, and multi-platform. Besides the dynamic computing environment for web-based systems, user requirements for new features also add to the complexity in designing and testing these systems.

The growing number of web applications combined with an ever-growing Internet user mass, emphasizes the importance of developing high-quality products. However, many attributes of quality web-based systems such as ease of navigation, accessibility, scalability, maintainability, usability, compatibility and interoperability, security, readability, and reliability are not given due consideration during development. Therefore, proper testing of web-based systems is needed in ensuring reliable, robust, and high-performing operation of web applications.

The testing of web-based systems is a complex task not only due to the overwhelming number of users on the web but there is also a lot of difference between traditional systems and the web-based systems. Traditional testing techniques are not adequate for web-based systems because they do not incorporate the problems associated with the computation environment, new features, and constraints of web-based applications.

17.1 Web-based System

The web-based software system consists of a set of web pages and components that interact to form a system which executes using web server(s), network,

HTTP, and a browser, and in which user input affects the state of the system. Thus, web-based systems are typical software programs that operate on the Internet, interacting with the user through an Internet browser. Some other terms regarding these systems are given below:

Web page is the information that can be viewed in a single browser window.

A *website* is a collection of web pages and associated software components that are related semantically by content and syntactically through links and other control mechanisms. Websites can be dynamic and interactive.

A *web application* is a program that runs as a whole or in part on one or more web servers and that can be run by users through a website. Web applications require the presence of web server in simple configurations and multiple servers in more complex settings. Such applications are called *web-based applications.* Similar applications, which may operate independent of any servers and rely on operating system services to perform their functions, are termed *web-enabled applications.* These days, with the integration of technologies used for the development of such applications, there is a thin line separating *web-based* and *web-enabled applications*, so we collectively refer to both of them as *web applications.*

17.2 Web Technology Evolution

17.2.1 First Generation/ 2-tier Web system

The web systems initially were based on a typical client-server model. The *client* is a web browser that people use to visit websites. The websites are on different computers, the *servers* and the HTML files are sent to the client by a software package called a *web server.* HTML files contain JavaScripts, which are small pieces of code that are interpreted on the client. HTML forms generate data that are sent back to the server to be processed by CGI programs. This 2-tier architecture consisting of two separate computers was a simple model suitable for small websites but with little security.

17.2.2 Modern 3-tier and N-tier Architecture

In the 2-tier architecture, it was difficult to separate presentation from business logic with the growth of websites. Due to this, applications were not scalable and maintainable. Moreover, having only one web server imposes a bottleneck; if there is a problem on that server, then the users cannot access the website (availability). Therefore, this simple client-server model was expanded first to the 3-tier model and then, to the N-tier model. To get quality

attributes, such as security, reliability, availability, scalability, and functionality, application server has been introduced wherein most of the software has been moved to a separate computer. Indeed, on large websites, the application server is actually a collection of application servers that operate in parallel. The application servers typically interact with one or more *database servers*, often running a commercial database (Fig. 17.1). Web and application servers often are connected by *middleware*, which are packages provided by software vendors to handle communication, data translation, and process distribution. Likewise, the application-database servers often interact through middleware. The web server that implements CGI, PHP, Java Servlets, or Active Server Pages (ASP), along with the application server that interacts with the database and other web objects is considered the middle tier. Finally, the database along with the DBMS server forms the third tier.

Figure 17.1 Modern websites

In the N-tier model, there are additional layers of security between potential crackers and the data and application business logic. Separating the presentation (typically on the web server tier) from the business logic (on the application server tier) makes it easier to maintain and expand the software both in terms of customers that can be serviced and services that can be offered. The use of distributed computing, particularly for the application servers, allows the web application to tolerate failures, handle more customers, and allows developers to simplify the software design.

17.3 Traditional Software and Web-based Software

Web systems are based on client-server architecture wherein a client typically enables users to communicate with the server. Therefore, these systems share some characteristics of client-server architecture. However, there are a number of aspects of web systems that necessitate having different techniques to test them. These are discussed below [147]:

1. Clients of the traditional client-server systems are platform-specific. This means that a client application is developed and tested for each supported

client operating system. But the web client is operating within the web browser's environment. Web browsers already consist of operating system-specific client software running on a client computer. But these browsers need to support HTML, as well as active contents to display web page information. For this purpose, browser vendors must create rendering engines and interpreters to translate and format HTML contents. In making these software components, various browsers and their releases introduce incompatibility issues.

2. Web-based systems have a more dynamic environment as compared to traditional client-server systems. In client-server systems, the roles of the clients and servers and their interactions are predefined and static as compared to web applications where client side programs and contents may be generated dynamically. Moreover, the environment for web applications is not predefined and is changing dynamically i.e. hardware and software are changing, configuration are ever-changing, etc. Web applications often are affected by these factors that may cause incompatibility and interoperability issues.
3. In the traditional client-server systems, the normal flow of control is not affected by the user. But in web applications, users can break the normal control flow. For example, users can press the back or refresh button in the web browser.
4. Due to the dynamic environment, web systems demand more frequent maintenance.
5. The user profile for web systems is very diverse as compared to client-server systems. Therefore, the load on web access due to this diversity is not predictable.

17.4 Challenges in Testing for Web-based Software

As discussed above, web-based software are different as compared to traditional systems. Therefore, we must understand the environment of their creation and then test them. Keeping in view the environment and behaviour of web-based systems, we face many challenges while testing them. These challenges become issues and guidelines when we perform testing of these systems. Some of the challenges and quality issues for web-based system are discussed here:

Diversity and complexity Web applications interact with many components that run on diverse hardware and software platforms. They are written in diverse languages and they are based on different programming approaches such as procedural, OO, and hybrid languages such as Java Server Pages

(JSPs). The client side includes browsers, HTML, embedded scripting languages, and applets. The server side includes CGI, JSPs, Java Servlets, and .NET technologies. They all interact with diverse back-end engines and other components that are found on the web server or other servers.

Dynamic environment The key aspect of web applications is its dynamic nature. The dynamic aspects are caused by uncertainty in the program behaviour, changes in application requirements, rapidly evolving web technology itself, and other factors. The dynamic nature of web software creates challenges for the analysis, testing, and maintenance for these systems. For example, it is difficult to determine statically the application's control flow because the control flow is highly dependent on user input and sometimes in terms of trends in user behaviour over time or user location. Not knowing which page an application is likely to display hinders statically modeling the control flow with accuracy and efficiency.

Very short development time Clients of web-based systems impose very short development time, compared to other software or information systems projects. For e.g. an e-business system, sports website, etc.

Continuous evolution Demand for more functionality and capacity after the system has been designed and deployed to meet the extended scope and demands, i.e. scalability issues.

Compatibility and interoperability As discussed above, there may also be compatibility issues that make web testing a difficult task. Web applications often are affected by factors that may cause incompatibility and interoperability issues. The problem of incompatibility may exist on both the client as well as the server side. The server components can be distributed to different operating systems. Various versions of browsers running under a variety of operating systems can be there on the client side. Graphics and other objects on a website have to be tested on multiple browsers. If more than one browser will be supported, then the graphics have to be visually checked for differences in the physical appearance. The code that executes from the browser also has to be tested. There are different versions of HTML. They are similar in some ways but they have different tags which may produce different features.

17.5 Quality Aspects

Based on the above discussion on the challenges of web software, some of the quality aspects which must be met while designing these systems are discussed here.

Reliability It is an expected feature that the system and servers are available at any time with the correct results. It means that the web systems must have high availability to be a reliable system. If one web software fails, the customer should be able to switch to another one. Thus, if web software is unreliable, websites that depend on the software will lose customers and the businesses may lose large amount of money.

Performance Performance parameters for web systems must also be tested. If they are not tested, they may even lead to the total failure of the system. Performance testing is largely on the basis of load testing, response time, download time of a web page, or transactions performed per unit time.

Security Since web systems are based on Internet, there are possibilities of any type of intrusion in these systems. Thus, security of web applications is another challenge for the web applications developers as well as for the testing team. Security is critical for e-commerce websites.

Tests for security are often broken down into two categories: testing the security of the infrastructure hosting the web application and testing for vulnerabilities of the web application. Some of the points that should be considered for infrastructure are firewalls and port scans. For vulnerabilities, there is user authentication, cookies, etc. Data collected must be secured internally. Users should not be able to browse through the directories in the server. A cookie is a text file on a user's system that identifies the user. Cookies must always be encrypted and must not be available to other users.

Usability This is another major concern while testing a web system. The users are real testers of these systems. If they feel comfortable in using them; then the systems are considered of high quality. Thus, usability also becomes an issue for the testers. The outlook of the system, navigation steps, all must be tested keeping the large categories of users in mind. Moreover, the users expect to be able to use the websites without any training. Thus, the software must flow according to the users' expectations, offer only needed information, and when needed, should provide navigation controls that are clear and obvious.

Scalability With the evolution of fast technology changes, the industry has developed new software languages, new design strategies, and new communication and data transfer protocols, to allow websites to grow as needed. Therefore, designing and building web software applications that can be easily scaled is currently one of the most interesting and important quality issues in software design. Scalability is defined as the web application's capacity to sustain the number of concurrent users and/or transactions, while sustaining sufficient response times to its users. To test scalability, web traffic loads must be determined in order to obtain the threshold requirement for scalability. Sometimes, existing traffic levels are used to simulate the load.

Availability Availability not only means that the web software is available at any time, but in today's environment it also must be available when accessed by diverse browsers. To be available in this sense, websites must adapt their presentations to work with all browsers, which require more knowledge and effort on the part of the developers.

Maintainability Web-based software has a much faster update rate. Maintenance updates can be installed and immediately made available to customers through the website. Thus, even small individual changes (such as changing the label on a button) can be installed immediately. One result of this is that instead of maintenance cycles of months or years, websites have maintenance cycles of days or even hours.

17.6 Web Engineering (WebE)

The diversity, complexity, and unique aspects of web-based software were not recognized and given due consideration by the developers early in the development of these systems. These systems also lack proper testing and evaluation. As a result, they faced a web crisis or what is called a *software crisis.*

> *Many organizations are heading toward a web crisis in which they are unable to keep the system updated and/or grow their system at the rate that is needed. This crisis involves the proliferation of quickly 'hacked together' web systems that are kept running via continual stream of patches or upgrades developed without systematic approaches.* [148]

Thus, to tackle the crisis in web-based system development, web engineering [150-156] as a new discipline has been evolved. This new discipline adopts a systematic approach to development of high-quality web-based systems.

> *Web engineering is an application of scientific, engineering, and management principles and disciplined and systematic approaches to the successful development, deployment, and maintenance of high-quality web-based systems and applications.* [149]

Web engineering activities involve all the aspects starting from the conception and development to the implementation, performance evaluation, and continual maintenance. We will discuss web-based analysis and design and then the testing of web-based systems in the subsequent sections.

17.6.1 Analysis and Design of Web-based Systems

Various methods have been proposed in the recent years for modeling web-based systems for analysis and design. Keeping in consideration the nature of web applications, the design of these systems must consider the following:

- Design for usability–interface design, navigation.
- Design for comprehension.
- Design for performance–responsiveness.
- Design for security and integrity.
- Design for evolution, growth, and maintainability.
- Design for testability.
- Graphics and multimedia design.
- Web page development.

It has been found that UML-based modeling [157-160] is more suitable according to the dynamic and complex nature of these systems. Therefore, we briefly mention all the UML-based models for analysis and design as discussed by Koch and Kraus [161]. The purpose to discuss them here is to get familiar with the analysis and design models in the context of web-based systems. Moreover, the understanding of these models will also help in testing these systems. As we have seen in testing for object-oriented software, testing of OO software was based on the models developed in analysis and design. On the same pattern, we will first understand UML-based analysis and design models and then move to testing these systems.

Conceptual Modeling

The conceptual model is a static view of the system that shows a collection of static elements of the problem domain. To model this, class and association modeling elements of UML are used. Additionally, role names, multiplicities, different forms of associations supported by the UML like aggregation, inheritance, composition, and association class can also be used to improve the modeling. If there are many classes in conceptual modeling, then packages may also be used. Since the conceptual model will show the objects involved in typical activities, use-cases with activity diagrams are also used for conceptual modeling.

Navigation Modeling

There are two types of navigation modeling: *navigation space model* and *navigation structure model*. The navigation space model specifies which objects can be visited by navigation through the application. It is a model at the analysis level. The navigation structure model defines how these objects are reached. It is a model at the design level.

The navigation models are represented by stereotyped class diagrams consisting of classes of those objects which can be visited by navigation through the web application and the associations which specify which objects can be

reached through navigation. The main modeling elements are the *stereotyped class «navigation class»* and *the stereotyped association «direct navigability»*. However, navigation model considers only those classes of the conceptual model that are relevant for navigation. Further, the navigation space model can be enhanced.

Presentation Modeling

This model transforms the navigation structure model in a set of models that show the static location of the objects visible to the user, i.e. a schematic representation of these objects. Using this presentation model, the discussion regarding interface design can be discussed with the customer. The sketches of these representations are drawn using a class diagram with UML composition notation for classes, i.e. containment represented by graphical nesting of the symbols of the parts within the symbol of the composite.

Web Scenarios Modeling

The dynamic behaviour of any web scenarios including the navigation scenarios are modeled here. The scenarios help in introducing more specification details. The navigation structure model is detailed here showing all the sequences of the behaviour of the objects, i.e. stimuli that trigger the transitions and explicitly including the actions to be performed. This modeling is done with the help of sequence interaction diagram/collaboration diagrams and state chart diagrams.

Task Modeling

The tasks performed by the user or system are modeled here. Basically, the use-cases as tasks are refined here. Since use-cases are refined by activity diagrams in UML, these diagrams are used for task modeling. The modeling elements for task modeling are those for activity diagrams, i.e. *activities, transitions, branche*s, etc. A task hierarchy can also be represented from coarse to fine-grained activities. The temporal order (with branches) between tasks is expressed by transitions between activities.

Task modeling also comprises of the description of the objects–called *referents*–that the user will perceive. For this purpose, presentation and conceptual objects are included in the task model. The relationship between tasks and these objects is expressed as object flow visualized as directed dashed lines. The ingoing presentation objects express user input through these presentation objects and outgoing presentation objects express output to the user through these presentation objects. The conceptual objects express input and output of tasks.

Configuration Modeling

Since web systems are diverse and complex in nature, we need to consider all the configuration and attributes that may be present on the client as well as on the server side. Moreover, some complex issues like deployment of process among multiple servers, compatibility with server OS, security issues, etc. are also considered. Deployment diagrams are used for configuration modeling.

17.6.2 Design Activities

Based on the discussion above, the following design activities have been identified for a web-based system [116]:

Interface Design

It considers the design of all interfaces recognized in the system including screen layouts. It also includes the types of navigation possible through the web system. All the interface design and navigational design issues are first discussed with the end-user. Interfaces should be designed such that they are flexible, consistent, and readable.

Content Design

All the contents regarding the web application identified as static elements in the conceptual modeling are designed here. The objects identified are known as *content objects*. Thus, these content objects and their relationships are designed here.

Architecture Design

The overall architecture of web applications is divided into the following three categories:

(i) An overall system architecture describing how the network and the various servers (web servers, application servers, and database servers) interact.

(ii) Content architecture design is needed to create a structure to organize the contents. Thus, this design is for structured presentation of the content objects. The content structure depends on factors like nature of the application, nature of the information, etc.

(iii) Application architecture is designed within the context of the development environment in which the web application is developed. This design provides an infrastructure so that functioning of the application in the form of components is clearly understood.

Presentation Design

This design is related to the look and feel of the application. The usability issues are considered here. The idea is that web application should be easy and flexible to use by its user. The design issues are related to layout of the web pages, graphics to be inserted, colour schemes, typefaces, etc.

Navigation design

Navigational paths are designed such that users are able to navigate from one place to another in the web application. The identified content objects, which are participating in navigation, become the navigation nodes and links between them define the navigation paths.

17.7 Testing of Web-Based Systems

Web-based systems have a different nature as compared to traditional systems. We have seen the key differences between them. Due to the environment difference and challenges of dynamic behaviour, complexity and diversity also makes the testing of these systems a challenge. These systems need to be tested not only to check whether it does what it is designed to do but also to evaluate how well it appears on the (different) web browsers. Moreover, they need to be tested for various quality parameters which are a must for these systems like security, usability, etc. Hence, a lot of effort is required for test planning and test designing.

Test cases should be written covering the different scenarios not only of the functional usage but also the technical implementation environment conditions such as network speeds, screen resolution, etc. For example, an application may work fine on Broadband Internet users but may perform miserably for users with dial-up connections. Web applications are known to give errors on slow networks, whereas they perform well on high-speed connections. Web pages don't render correctly for certain situations but work fine with others. Images may take longer to download for slower networks and the end-user perception of the application may not be good.

There may be a number of navigation paths possible through a web application. Therefore, all these paths must be tested. Along with multiple navigation paths, users may have varying backgrounds and skills. Testing should be performed keeping in view all the possible categories of users in view. This becomes the issues in usability testing.

Another issue of great concern is the security testing of web applications. There are two cases. For Intranet-based applications, there are no such threats on the application. However, in case of Internet-based applications, the users

may need to be authenticated and security measures may have to be much more stringent. Test cases need to be designed to test the various scenarios and risks involved.

Traditional software must be tested on different platforms, or it may fail in some platforms. Similarly, web-based software must be tested with all the dimensions which are making it diverse. For example, users may have different browsers while accessing the applications. This aspect also needs to be tested under compatibility testing. If we test the application only on Internet Explorer, then we cannot ensure that if works well on Netscape or other browsers. Because these browsers may not only render pages differently but also have varying levels of support for client side scripting languages such as Java Script.

The strategy for testing web-based systems is the same as for other systems, i.e. verification and validation. Verification largely considers the checking of analysis and design models which have been described above. Various types of testing derive from the design models only. However, the quality parameters are also important factors which form the other types of testing. We discuss these various types of web testing in the following sections.

17.7.1 Interface Testing

Interface is a major requirement in any web application. More importantly, the user interface with web application must be proper and flexible. Therefore as a part of verification, present model and web scenarios model must be checked to ensure all interfaces. The interfaces between the concerned client and servers should also be considered. There are two main interfaces on the server side: *web server and application server interface* and *application server and database server interface.*

Web applications establish links between the web server and the application server at the onset. The application server in turn connects to the database server for data retrieval, processing, and storage. It is an important factor that these connections or interfaces work seamlessly without any failure or degradation in performance of speed and accuracy. Testing should check for appropriate error messages, roll back in case of failure to execute or user interruption. The complexity of this test is in ensuring that the respective interface, be it web server or application or database interface, captures the errors and initiates the appropriate error messages to the web application.

Thus, in interface testing, all interfaces are checked such that all the interactions between these servers are executed properly. Errors are handled properly. If database or web server returns any error message for any query by the application server, then the application server should catch and display these

error messages appropriately to users. Check what happens if the user interrupts any transaction in between? Check what happens if connection to web server is reset in between? Compatibility of server with software, hardware, network, and database should also be tested.

17.7.2 Usability Testing

The presentation design emphasizing the interface between user and web application gives rise to usability testing. The actual user of application should feel good while using the application and understand every thing visible to him on it. Usability testing is not a functionality testing, but the web application is reviewed and tested from a user's viewpoint. The importance of usability testing can be realized with the fact that we can even lose users because of a poor design. For example, check that form controls, such as boxes and buttons, are easy to use, appropriate to the task, and provide easy navigation for the user. The critical point for designers and testers in this attesting is that web application must be as pleasant and flexible as possible to the user.

Usability testing may include tests for navigation. It refers to how the user navigates the web pages and uses the links to move to different pages. Besides this, content should be logical and easy to understand. Check for spelling errors. Use of dark colours annoy users and should not be used in the site theme. You can follow some standards that are used for web page and content building. Content should be meaningful. All the anchor text links should work properly. Images should be placed properly with the proper sizes.

For verification of web application, the presentation design must be checked properly so that most of the errors are resolved at the earlier stages only. Verification can be done with a technique called *card-sorting technique* given by Michael D. Levi and Frederick G. Conrad. According to this technique, a group of end-users are given a set of randomly ordered index cards, each of which is labeled with a concept from the task domain. The users scatter all the index cards on the desk, sort them according to a category. Further arrange these groups in broader category. Write a name for each of the larger groupings, write it on a slip of paper, and attach each slip to the corresponding group. After this process of sorting, compare the card sort results to the original presentation design of the application. In this comparison, we may find several areas where we can improve the underlying hierarchy so that users can easily find the information they were looking for.

For validation, a scenario-based usability testing can be performed. This type of testing may take the help of use-cases designed in the use-case model for the system. All the use-cases covering usability points can become the base for designing test cases for usability testing. In this usability testing, some categories of users are invited to perform the testing. The testers meet the group

of participants to describe the system in general terms, give an overview of the process, and answer any questions. Participants are seated in front of a desktop computer and asked to work through the scenario questions. At the end of the session, a group discussion is held to note down the participants' reactions and suggestions for improvement. The results of user testing can also be taken from the participants in the form of a questionnaire. As they use the application, they answer these questions and give feedback to the testers in the end. Besides this, web server logs can also be maintained for the session of usage by the participants. It provides the testers with a time-stamped record of each participant's sessions providing an excellent approximation of users' journeys through a site.

The log mining method discussed above can be used for improvement in application even after the release of the product. The actual user session logs can be recorded and evaluated from the usability point. The general guidelines for usability testing are:

1. Present information in a natural and logical order.
2. Indicate similar concepts through identical terminology and graphics. Adhere to uniform conventions for layout, formatting, typefaces, labeling, etc.
3. Do not force users to remember key information across documents.
4. Keep in consideration that users may be from diverse categories with various goals. Provide understandable instructions where useful. Lay out screens in such a manner that frequently accessed information is easily found.
5. The user should not get irritated while navigating through the web application. Create visually pleasing displays. Eliminate information which is irrelevant or distracting.
6. Content writer should not mix the topics of information. There should be clarity in the information being displayed.
7. Organize information hierarchically, with more general information appearing before more specific detail. Encourage the user to delve as deeply as needed, but to stop whenever sufficient information has been received.
8. Check that the links are active such that there are no erroneous or misleading links.

17.7.3 Content Testing

The content we see on the web pages has a strong impression on its user. If these contents are not satisfactory to him, he may not visit the web page again.

Check the completeness and correctness properties of web application content. Check that certain information is available on a given web page, links between pages exist, or even check the existence of the web pages themselves (completeness property). Furthermore, web application content may need to be checked against semantic conditions to see if they meet the web document (correctness property). Therefore the contents should be correct, visible, flexible to use, organized, and consistent.

This type of testing targets the testing of static and dynamic contents of web application. Static contents can be checked as a part of verification. For instance, forms are an integral part of any website. Forms are used to get information from users and to keep interacting with them. First, check all the validations on each field. Check for the default values of fields and also wrong inputs to the fields in the forms. Options to create forms if any, form delete, view, or modify the forms must also be checked.

Static testing may consider checking the following points:

1. Various layouts.
2. Check forms for their field validation, error message for wrong input, optional and mandatory fields with specified length, buttons on the form, etc.
3. A table is present and has the expected number of rows and columns and pre-defined properties.
4. Grammatical mistakes in text description of web page.
5. Typographical mistakes.
6. Content organization.
7. Content consistency.
8. Data integrity and errors while you edit, delete, modify the forms.
9. Content accuracy and completeness.
10. Relationship between content objects.
11. Text contents.
12. Text fragments against formatting expectations. This differs slightly from simple text checking in that the formatting tags can be located loosely on the page as opposed to a fixed string for text content.
13. Graphics content with proper visibility.
14. Media contents to be placed at appropriate places.
15. All types of navigation links like internal links, external links, mail links, broken links to be placed at appropriate places.
16. All links on a web page are active.

A checklist for content verification can be prepared as was seen in Chapter 3.

There may also be dynamic contents on a web page. Largely, dynamic testing will be suitable in testing these dynamic contents. These dynamic contents can be in many forms. One possibility is that there are constantly changing contents, e.g. weather information web pages or online newspaper. Another case may be that web applications are generated dynamically from information contained in a database or in a cookie. Many web applications today work interactively in the manner that in response to a user request for some information, it interacts with some DBMS, extracts the relevant data, creates the dynamic content objects for this extracted data, and sends these content objects to the user for display. In the same manner, the information can be generated dynamically from cookies also, i.e. dynamic content objects for cookies are also there.

The problem in the design of these dynamic contents is that there may be many errors due to its dynamic behaviour. Therefore, testing of these dynamic contents becomes necessary to uncover the errors. Changing contents on a web page must be tested whether the contents are appearing every time in the same format. Moreover, there is consistency between the changed content and static content.

Test all database interface-related functionality for all dynamic content objects. Check if all the database queries are executing correctly, data is retrieved correctly, and also updated correctly. Load testing or performance testing can also be done on database.

Cookies are small files stored on the user machine. These are basically used to maintain the session, mainly the login sessions. The testing of the entire interface with these cookies must also be tested. Test the application by enabling or disabling the cookies in browser options. Test if the cookies are encrypted before writing to user machine. Check the effect on application security by deleting the cookies.

17.7.4 Navigation Testing

We have checked the navigation contents in interface testing. But to ensure the functioning of correct sequence of those navigations, navigation testing is performed on various possible paths in the web application. Design the test cases such that the following navigations are correctly executing:

1. Internal links
2. External links
3. Redirected links
4. Navigation for searching inside the web application

The errors must be checked during navigation testing for the following:

1. The links should not be broken due to any reason.
2. The redirected links should be with proper messages displayed to the user.
3. Check that all possible navigation paths are active.
4. Check that all possible navigation paths are relevant.
5. Check the navigations for the back and forward buttons, whether they are working properly.

17.7.5 Configuration/Compatibility Testing

Diversity in configuration for web applications makes the testing of these systems very difficult. As discussed above, there may be various types of browsers supporting different operating systems, variations in servers, networks, etc. Therefore, configuration testing becomes important so that there is compatibility between various available resources and application software. The testers must consider these configurations and compatibility issues so that they can design the test cases incorporating all the configurations. Some points to be careful about while testing configuration are:

1. There are a number of different browsers and browser options. The web application has to be designed to be compatible for majority of the browsers.
2. The graphics and other objects on a website have to be tested on multiple browsers. If more than one browser will be supported, then the graphics have to be visually checked for differences in the physical appearance. Some of the things to check are centering of objects, table layouts, colours, monitor resolution, forms, and buttons.
3. The code that executes from the browser also has to be tested. There are different versions of HTML. They are similar in some ways but they have different tags which may produce different features. Some of the other codes to be tested are Java, JavaScript, ActiveX, VBscripts, Cgi-Bin Scripts, and Database access. Cgi-Bin Scripts have to be checked for end-to-end operations and is most essential for e-commerce sites. The same goes for database access.
4. All new technologies used in the web development like graphics designs, interface calls like different API's, may not be available in all the operating systems. Test your web application on different operating systems like Windows, Unix, MAC, Linux, Solaris with different OS flavors.

17.7.6 Security Testing

Today, the web applications store more vital data and the number of transactions on the web has increased tremendously with the increasing number of users. Therefore, in the Internet environment, the most challenging issue is to protect the web applications from hackers, crackers, spoofers, virus launchers, etc. Through security testing, we try to ensure that data on the web applications remain confidential, i.e. there is no unauthorized access. Security testing also ensures that users can perform only those tasks that they are authorized to perform.

In a web application, the risk of attack is multifold, i.e. it can be on the web software, client-side environment, network communications, and server-side environments. Therefore, web application security is particularly important because these are generally accessible to more users than the desktop applications. Often, they are accessed from different locations, using different systems and browsers, exposing them to different security issues, especially external attacks. It means that web applications must be designed and developed such that they are able to nullify any attack from outside. Therefore, this issue is also related to testing of web application in terms of security. We need to design the test cases such that the application passes the security test.

Security Test Plan

Security testing can be divided into two categories: testing the security of the infrastructure hosting the web application and testing for vulnerabilities of the web application. Firewalls and port scans can be the solution for security of infrastructure. For vulnerabilities, user authentication, restricted and encrypted use of cookies, data communicated must be planned. Moreover, users should not be able to browse through the directories in the server.

Planning for security testing can be done with the help of some threat models. These models may be prepared at the time of requirement gathering and test plan can also be prepared correspondingly. These threat models will help in incorporating security issues in designing and later can also help in security testing.

Find all the component interfaces for performing security testing on a component. This is because most of the security bugs can be found on the interfaces only. The interfaces are then prioritized according to their level of vulnerability. The high-priority interfaces are tested thoroughly by injecting mutated data to be accessed by that interface in order to check the security.

While performing security testing, the testers should take care that they do not modify the configuration of the application or the server, services running on the server, and existing user or customer data hosted by the application.

Various Threat Types and their corresponding Test cases

Unauthorized user/Fake identity/Password cracking When an unauthorized user tries to access the software by using fake identity, then security testing should be done such that any unauthorized user is not able to see the contents/ data in the software.

Cross-site scripting (XSS) When a user inserts HTML/client-side script in the user interface of a web application and this insertion is visible to other users, it is called *cross-site scripting* (XSS). Attacker can use this method to execute malicious script or URL on the victim's browser. Using cross-site scripting, attacker can use scripts like JavaScript to steal user cookies and information stored in the cookies. To avoid this, the tester should additionally check the web application for XSS.

Buffer overflows Buffer overflow is another big problem when handling memory allocation if there is no overflow check on the client software. Due to this problem, malicious code can be executed by the hackers. In the application, check the buffer overflow module and the different ways of submitting a range of lengths to the application.

URL manipulation There may be chances that communication through HTTP is also not safe. The web application uses the HTTP GET method to pass information between the client and the server. The information is passed in parameters in the query string. Again, the attacker may change some information in query string passed from GET request so that he may get some information or corrupt the data. When one attempts to modify the data, it is known as fiddling of data. The tester should check if the application passes important information in the query string and design the test cases correspondingly. Write the test cases such that a general user tries to modify the private information.

SQL injection Hackers can also put some SQL statements through the web application user interface into some queries meant for querying the database. In this way, he can get vital information from the server database. Even if the attacker is successful in crashing the application, from the SQL query error shown on the browser, the attacker can get the information they are looking for. Design the test cases such that special characters from user inputs should be handled/escaped properly in such cases.

Denial of service When a service does not respond, it is denial of service. There are several ways that can make an application fail. For example, heavy load put on the application, distorted data that may crash the application, overloading of memory, etc. Design the test cases considering all these factors.

17.7.7 Performance Testing

Web applications' performance is also a big issue in today's busy Internet environment. The user wants to retrieve the information as soon as possible without any delay. This assumes the high importance of performance testing of web applications. Is application able to respond in a timely manner or is it ready to take the maximum load or beyond that? These questions imply that web applications must also be tested for performance. Performance testing helps the developer to identify the bottlenecks in the system and can be rectified.

In performance testing, we evaluate metrics like response time, throughput, and resource utilization against desired values. Using the results of this evaluation, we are able to predict whether the software is in a condition to be released or requires improvement before it is released. Moreover, we can also find the bottlenecks in the web application. Bottlenecks for web applications can be code, database, network, peripheral devices, etc.

Performance Parameters

Performance parameters, against which the testing can be performed, are given here:

Resource utilization The percentage of time a resource (CPU, Memory, I/O, Peripheral, Network) is busy.

Throughput The number of event responses that have been completed over a given interval of time.

Response time The time lapsed between a request and its reply.

Database load The number of times database is accessed by web application over a given interval of time.

Scalability The ability of an application to handle additional workload, without adversely affecting performance, by adding resources such as processor, memory, and storage capacity.

Round-trip time How long does the entire user-requested transaction take, including connection and processing time?

Types of Performance Testing

Performance tests are broadly divided into following categories:

Load testing Can the system sustain at times of peak load? The site should handle many simultaneous user requests, large input data from users, simultaneous connection to DB, heavy load on specific pages, etc. When

we need to test the application with these types of loads, then load testing is performed on the system. It focuses on determining or validating performance characteristics of the system when subjected to workloads and load volumes anticipated during production operations. It refers to how much load (the best example of load in web application is how many concurrent users) you can put on the web application and it will still serve flawlessly.

There are a few types of load testing which we can be performed. We can perform *capacity testing* to determine the maximum load the web service can handle before failing. Capacity testing reveals the web services' ultimate limit. We may also perform *scalability testing* to determine how effectively the web service will expand to accommodate an increasing load.

Stress testing Generally, stress refers to stretching the system beyond its specification limits. Web stress testing is performed to break the site by giving stress and check how the system reacts to stress and how the system recovers from crashes. It focuses on determining or validating performance characteristics of the system when subjected to conditions beyond those anticipated during production operations. It also tests the performance of the system under stressful conditions, such as limited memory, insufficient disk space, or server failure. These tests are designed to determine under what conditions an application will fail, how it will fail, and how gracefully it may recover from the failure. Some examples of graceful failure are: the system saves state at the time of failure and does not crash suddenly; On restarting it, the system recovers from the last good state; the system shows meaningful error messages to the user.

SUMMARY

Web-based systems are specialized software where conventional testing is not applicable. We need to understand the issues related to web-based system and applications in order to understand the testing issues and perform web-based testing. Web applications in today's environment are highly accessible by every type of users. Their uses demand available, reliable, and secure web applications. Due to these considerations, testing of web software becomes a challenge for the testers.

Web applications interact with many components that run on diverse hardware and software platforms. It is difficult to determine statically the application's control flow because the control flow is highly dependent on user input and sometimes in terms of trends on user behaviour over time or user location. Web applications often are affected by factors that may cause incompatibility and interoperability issues. The problem of incompatibility may exist on both the client as well as the server side. Thus, web applications are diverse, complex, dynamic, and face the problems of incompatibility and interoperability.

Web engineering is the solution for web-based systems like software engineering is for general software. Web engineering is the systematic discipline which includes the process of

analysing, designing, building, and testing the web-based systems. In this chapter, we have discussed the analysis and design models for the understanding of testing of web-based systems. After this, web testing types have been categorized taking analysis and design models as the base.

Let us review the important concepts described in this chapter:

- The web-based software system consists of a set of web pages and components that interact to form a system which executes using web server(s), network, HTTP, and a browser, and in which user input affects the state of the system.
- Web applications interact with many components that run on diverse hardware and software platforms.
- The key aspect of web applications is its dynamic nature. The dynamic aspects are caused by uncertainty in the program behaviour, changes in application requirements, rapidly evolving web technology itself, and other factors.
- Clients of web-based systems impose very short development time, compared to other software or information systems projects.
- Testing of web-based systems needs techniques considering the different nature of web software.
- Web engineering is the application of scientific, engineering, and management principles and disciplined and systematic approaches to the successful development, deployment, and maintenance of high-quality web-based systems and applications.
- Conceptual model is a static view of the system that shows a collection of static elements of the problem domain.
- The navigation space model specifies which objects can be visited by navigation through the application. It is a model at the analysis level.
- Presentation modeling transforms the navigation structure model into a set of models that show the static location of the objects visible to the user, i.e. a schematic representation of these objects.
- Web scenarios modeling models the dynamic behaviour of any web scenarios including the navigation scenarios.
- Task modeling models the tasks performed by the user or system. Basically, the use-cases as tasks are refined here.
- Configuration modeling considers the entire configuration and attributes that may be present on the client as well as the server side.
- Interface design considers the design of all interfaces recognized in the system including screen layouts.
- Content design considers the design of all the contents regarding the web application identified as static elements in the conceptual modeling.
- Architecture design considers the design of the overall system architecture of the web application, content architecture design needed to create a structure to organize the contents, and application architecture designed within the context of the development environment in which the web application is developed.

- Presentation design considers the design related to the look and feel of the application.
- Navigation design considers the design of navigational paths such that users are able to navigate from one place to another in the web application.
- Interface testing tests the user interface with web application, the interfaces between the concerned client and servers, server and application server interface, and application server and database server interface.
- Usability testing ensures that web application is as pleasant and flexible as possible to the user.
- Content testing checks that certain information is available on a given web page, links between pages exist, and also the existence of the web pages themselves (completeness property). Furthermore, web application content may need to be checked against semantic conditions to see if they are met by the web document (correctness property).
- Navigation testing ensures the functioning of correct sequence of navigations like internal links, external links, redirected links, navigation for searching inside the web application, etc.
- Configuration/compatibility testing checks the compatibility between various available resources and application software. The testers must consider these configurations and compatibility issues so that they can design the test cases incorporating all the configurations.
- Security testing tests that data on the web applications remain confidential. It also ensures that users can perform only those tasks that they are authorized to perform.
- Performance testing helps the developer to identify the bottlenecks in the system that can be rectified.
- Load testing focuses on determining or validating performance characteristics of the system when subjected to workloads and load volumes anticipated during production operations.
- Stress testing is performed to break the site by giving stress and check how the system reacts to the stress and how it recovers from crashes.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Web-based systems impose very _____ development time, compared to other software or information systems projects.
   (a) long
   (b) short
   (c) same
   (d) none of the above
2. The problem of incompatibility may exist on the
   (a) client side only

(b) server side only
(c) both client as well as server side
(d) none of the above

3. Conceptual model is a _____ view of the system that shows a collection of static elements of the problem domain.
   (a) static
   (b) dynamic
   (c) both
   (d) none of the above
4. The navigation space model is at the _____.
   (a) analysis level
   (b) design level
   (c) testing level
   (d) none of the above
5. Navigation structure model is at the _____.
   (a) analysis level
   (b) design level
   (c) testing level
   (d) none of the above
6. Presentation modeling transforms the _____ model in a set of models that show the static location of the objects visible to the user.
   (a) navigation structure
   (b) navigation space
   (c) object structure
   (d) none of the above
7. When one attempts to _____ the data, this is known as fiddling of data.
   (a) add
   (b) modify
   (c) delete
   (d) none of the above
8. Web applications require the presence of web server in simple configurations and multiple servers in more complex settings. Such applications are more precisely called _____.
   (a) web-enabled applications
   (b) website
   (c) web page
   (d) web-based applications

9. Applications, which may operate independent of any servers and rely on operating system services to perform their functions, are called _____.
   (a) web-enabled applications
   (b) website
   (c) web page
   (d) web-based applications
10. Simple client-server model was expanded first to _____ model and then the N-tier later on.
   (a) 3-tier
   (b) 4-tier
   (c) 5-tier
   (d) none of the above
11. A cookie is a _____ file on a user's system that identifies the user.
   (a) hyper-text
   (b) text
   (c) graphic
   (d) none of the above
12. _____ is defined as the web application's capacity to sustain the number of concurrent users and/or transactions, while sustaining sufficient response times to its users.
   (a) Availability
   (b) Maintainability
   (c) Scalability
   (d) none of the above
13. Web-based software has a much _____ update rate.
   (a) faster
   (b) slower
   (c) normal
   (d) none of the above
14. The navigation models are represented by stereotyped _____ diagrams.
   (a) activity
   (b) class
   (c) use-case
   (d) none of the above
15. Web-scenario modeling is done with the help of _____ diagrams.
   (a) interaction diagrams
   (b) state chart diagrams
   (c) activity diagrams
   (d) none of the above

REVIEW QUESTIONS

1. What is the difference between web-based and web-enabled applications?
2. What is the difference among 2-tier, 3-tier, and N-tier web system architecture?
3. What is the difference between traditional software and web-based software?
4. 'Web-based systems are diverse, complex, and dynamic in nature'. Comment on this.
5. What are the quality aspects to be considered in web testing?
6. What is web crisis? Is it same as software crisis?
7. What is web engineering? Is it same as software engineering?
8. Take a web application project and prepare all the analysis and design models discussed in this chapter.
9. Consider all types of interfaces in the project taken in Question 8 and design the test cases for interface testing.
10. What will be the criteria for performing unit testing, integration testing, and system testing of a web application?
11. Design a checklist for verification of a web-based software.
12. List the quality aspects of a website and perform performance testing for it.
13. What are the possible types of vulnerabilities in a web-based system? How do we handle them in security testing?
14. What is the difference between load and stress testing?
15. Search material on testing tools used for various types of web testing. Prepare a comparison list of all of them.
16. What is the difference among XSS, buffer overflow, URL manipulation, and SQL injection?

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Part 7 Tracking the Bug

CHAPTER

Chapter 18:
Debugging

The testing process ends with the successful finding of bugs. But this process is a waste if we do not track this bug to find the exact location of errors. It means that the next immediate process after testing is debugging, wherein the bug is tracked to its exact location.

However, debugging as a process has not been given much attention. Due to this, debugging becomes a complex process. It is also said that debugging is more an art than a technique. It may be possible that debugging as an art may give better results sometimes, but not always. Moreover, the proficiency to use it as an art comes with experience. Thus, the time has come that we formalize the debugging methods as techniques, rather than randomized tracking of bug. This part discusses this issue and several debugging techniques.

This part will make ground for the following concepts:

- Debugging as an art or technique
- Debugging is a process
- Debugging techniques
- Debuggers

Chapter

18

Debugging

OBJECTIVES

After reading this chapter, you should be able to understand:

- Debugging is not testing and testing is not debugging
- Debugging is more an art than a technique
- Steps of a general debugging process
- Debugging with Memory Dump
- Debugging with Watch Points
- Backtracking method
- Debugging guidelines
- Various types of Debuggers

Debugging is not a part of the testing domain. Therefore, debugging is not testing. It is a separate process performed as a consequence of testing. But the testing process is considered a waste if debugging is not performed after the testing. Testing phase in the SDLC aims to find more and more bugs, remove the errors, and build confidence in the software quality. In this sense, testing phase can be divided into two parts:

1. Preparation of test cases, executing them, and observing the output. This is known as *testing*.
2. If output of testing is not successful, then a failure has occurred. Now the goal is to find the bugs that caused the failure and remove the errors present in the software. This is called *debugging*.

Debugging is the process of identification of the symptoms of failures, tracing the bug, locating the errors that caused the bug, and correcting these errors. Describing it in more concrete terms, debugging is a two-part process. It begins with some indication of the existence of an error. It is the activity of [2]:

1. Determining the exact nature of the bug and location of the suspected error within the program.
2. Fixing or repairing the error.

18.1 Debugging: An Art or Technique?

Since software testing has not reached a maturity-level discipline, debugging is also an unplanned activity in most of the projects. Due to the lack of attention given to the debugging process, it has been developed as an art more

than a technique. However, viewing the debugging process as an art has given better results as compared to technique. The art of analysing the bug history can help a lot in debugging. Similar projects, more often than not, have a history of being affected by similar bugs in the past. Moreover, thinking critically in testing and debugging also supports debugging as an art. We start with the symptoms of failures and critically examine every symptom that may lead to the location of the actual error.

18.2 Debugging Process

As discussed, the goal of debugging process is to determine the exact nature of failure with the help of symptoms identified, locate the bugs and errors, and finally correct it. The debugging process (see Fig. 18.1) is explained in the following steps:

Figure 18.1 Debugging process

- Check the result of the output produced by executing test cases prepared in the testing process. If the actual output matches with the expected output, it means that the results are successful. Otherwise, there is failure in the output which needs to be analysed.
- Debugging is performed for the analysis of failure that has occurred, where we identify the cause of the problem and correct it. It may be possible that symptoms associated with the present failure are not suf-

ficient to find the bug. Therefore, some additional testing is required so that we may get more clues to analyse the causes of failures.

- If symptoms are sufficient to provide clues about the bug, then the cause of failure is identified. The bug is traced to find the actual location of the error.
- Once we find the actual location of the error, the bug is removed with corrections.
- Regression testing is performed as bug has been removed with corrections in the software. Thus, to validate the corrections, regression testing is necessary after every modification.

18.3 Debugging is Difficult

Debugging is a time-consuming process and sometimes becomes frustrating if you are not able to find the cause of the failure. Here are some reasons why:

- Debugging is performed under a tremendous amount of pressure which may cause problems instead of leading to clues of the problem. This pressure happens due to the following reasons:
  (i) Self-induced pressure to fix the suspected bug as early as possible since it is related to the individual performance and ego.
  (ii) Organizational time-bound pressure to fix the bugs is always there.
- The gap between the faults and failures is very large in case of debugging of software systems. Without examining the whole program, we cannot locate the error directly. That is why, debugging starts with observing the failures for finding clues of the problem. With the clues, we then trace the bug and error. Moreover, the symptoms are also not clear after observing the failures. This makes debugging a time-consuming process.
- The complex design of the software affects the debugging process. Highly coupled and low-cohesive module is difficult to debug.
- Experience matters in the debugging process. A new member of the team will find it difficult or cumbersome as compared to experienced members.
- Sometimes, failures don't happen. This does not mean that there is no bug. What happens is that we are unable to reproduce that failure always. This type of bug consumes a lot of time.

18.4 Debugging Techniques

18.4.1 Debugging with Memory Dump

In this technique, a printout of all registers and relevant memory locations is obtained and studied. The storage locations are in octal or hexadecimal format. The relevant data of the program is observed through these memory locations and registers for any bug in the program. However, this method is inefficient. This method should be used as the last option. Following are some drawbacks of this method [2]:

1. There is difficulty of establishing the correspondence between storage locations and the variables in one's source program.
2. The massive amount of data with which one is faced, most of which is irrelevant.
3. It is limited to static state of the program as it shows the state of the program at only one instant of time.

18.4.2 Debugging with Watch Points

The program's final output failure may not give sufficient clues about the bug. It is possible that intermediate execution, at some points in the program, may provide sufficient cause of the problem. At a particular point of execution in the program, value of variables or other actions can be verified. These particular points of execution are known as *watch points*. Debugging with watch points can be implemented with the following methods:

Output statements In this method, output statements can be used to check the state of a condition or a variable at some watch point in the program. Therefore, output statements are inserted at various watch points; program is complied and executed with these output statements. Execution of output statements may give some clues to find the bug. This method displays the dynamics of a program and allows one to examine information related to the program failure. This method has the following drawbacks:

(i) It may require many changes in the code. These changes may mask an error or introduce new errors in the program.

(ii) After analysing the bug, we may forget to remove these added statements which may cause other failures or misinterpretations in the result.

Breakpoint execution This is the advanced form of the watch point used with an automated debugger program. *Breakpoint* is actually a watch point inserted at various places in the program. But these insertions are not placed in the actual user program and therefore need not be removed manually like output statements.

The other difference in this method is that program is executed up to the breakpoint inserted. At that point, you can examine whatever is desired. Afterwards, the program will resume and will be executed further for the next breakpoint. Thus, breakpoints allow the programmer to control execution of the program and specify how and where the application will stop to allow further examination. Since there can be many breakpoints in the same program, all breakpoints can also be viewed by the programmer. Breakpoints displayed in the view show the source location of each breakpoint as well as its status.

Breakpoints have an obvious advantage over output statements. Some are discussed here:

(a) There is no need to compile the program after inserting breakpoints, while this is necessary after inserting output statements.

(b) Removing the breakpoints after their requirement is easy as compared to removing all inserted output statements in the program.

(c) The status of program variable or a particular condition can be seen after the execution of a breakpoint as the execution temporarily stops after the breakpoint. On the other hand in case of output statements, the full program is executed and output is viewed after the total execution of the program.

Breakpoints can be categorized as follows:

(i) *Unconditional breakpoint:* It is a simple breakpoint without any condition to be evaluated. It is simply inserted at a watch point and its execution stops the execution of the program.

(ii) *Conditional breakpoint:* On the activation of this breakpoint, one expression is evaluated for its Boolean value. If true, the breakpoint will cause a stop; otherwise, execution will continue.

(iii) *Temporary breakpoint:* This breakpoint is used only once in the program. When it is set, the program starts running, and once it stops, the temporary breakpoint is removed. The temporary nature is one of its attributes. In other respects, it is just like other breakpoints.

(iv) *Internal breakpoint:* These are invisible to the user but are key to debugger's correct handling of its algorithms. These are the breakpoints set by the debugger itself for its own purposes.

Single stepping The idea of single stepping is that the users should be able to watch the execution of the program after every executable instruction. After every instruction execution, the users can watch the condition or status of variable. Single stepping is implemented with the help of internal breakpoints.

Step-into It means execution proceeds into any function in the current source statement and stops at the first executable source line in that function.

Step-over It is also called skip, instead of step. It treats a call to a function as an atomic operation and proceeds past any function calls to the textually succeeding source line in the current scope.

18.4.3 BACKTRACKING

This is a logical approach for debugging a program. The following are the steps for backtracking process (see Fig. 18.2):

Figure 18.2 Backtracking

(a) Observe the symptom of the failure at the output side and reach the site where the symptom can be uncovered. For example, suppose you are not getting display of a value on the output. After examining the symptom of this problem, you uncover that one module is not sending the proper message to another module. It may also be the case that function *X* is not working. This can be uncovered using the above mentioned techniques, e.g. using breakpoints.

(b) Once you have reached the site where symptom has been uncovered, trace the source code starting backwards and move to the highest level of abstraction of design. The bug will be found in this path. For tracing backwards, collect source code of all the modules which are related at this point.

(c) Slowly isolate the module through logical backtracking using data flow diagrams (DFDs) of the modules wherein the bug resides.

(d) Logical backtracking in the isolated module will lead to the actual bug and error can thus be removed.

Backtracking requires that the person debugging must have knowledge regarding the design of the system so that he can understand the logical flow of the program through DFDs. This method also requires the manual observation of the source code. This method is very effective as compared to the

other methods in pinpointing the error location quickly, if you understand the logic of the program.

18.5 Correcting the Bugs

The second phase of the debugging process is to correct the error when it has been uncovered. But it is not as easy as it seems. The design of the software should not be affected by correcting the bug or due to new modifications. Before correcting the errors, we should concentrate on the following points:

(a) Evaluate the coupling of the logic and data structure where corrections are to be made. Highly coupled module correction can introduce many other bugs. That is why low-coupled module is easy to debug.

(b) After recognizing the influence of corrections on other modules or parts, plan the regression test cases to perform regression testing as discussed earlier.

(c) Perform regression testing with every correction in the software to ensure that the corrections have not introduced bugs in other parts of the software.

18.5.1 Debugging Guidelines

Fresh thinking leads to good debugging Don't spend continuous hours and hours on debugging a problem. The continuous involvement in the problem will collapse your efficiency. So rest in between and start with a fresh mind to concentrate on the problem.

Don't isolate the bug from your colleagues It has been observed that bugs cannot be solved in isolation. Show the problem to other colleagues, explain to them what is happening, and discuss the solution. There will be a lot of clues that have not been tried yet. It has also been observed that by just discussing the problem with others may suddenly lead you to come up with some solution. Therefore, it is always better to share the problem to others.

Don't attempt code modifications in the first attempt Debugging always starts with the analysis of clues. If you don't analyse the failures and clues and simply change the code randomly with the view—'let's see what happens with this change', this thinking will lead you nowhere but with junk code added to the software.

Additional test cases are a must if you don't get the symptom or clues to solve the problem Design test cases with the view to execute those parts of the program

which are causing the problem. This test case execution will provide symptoms to be analysed.

Regression testing is a must after debugging Don't simply fix the error and forget about it. The process of debugging does not end with fixing the error. After fixing the error, you need to realize the effects of this change on other parts. Test cases must be designed to uncover any effects on bugs produced on other parts. That is why regression testing is necessary after fixing the errors.

Design should be referred before fixing the error Any change in the code to fix the bug should be according to pre-specified design of the software. While making corrections, that design should not be violated by any means.

18.6 Debuggers

In the process of debugging, the help of debugging tools can also be taken to speed up the process. Debugger is a tool to help track down, isolate, and remove bugs from the software program. It controls the application being debugged so as to allow the programmer to follow the flow of program execution and at any desired point, stop the program and inspect the state of the program to verify its correctness. Debuggers can be used to

- Illustrate the dynamic nature of a program.
- Understand a program as well as to find and fix the bugs.
- Control the application using special facilities provided by the underlying operating system to give the use very fine control over the program under test.

18.6.1 Types of Debuggers

Kernel debugger It is for dealing with problems with an OS kernel on its own or for interactions between heavily OS-dependent application and the OS.

Basic machine-level debugger It is used for debugging the actual running code as they are processed by the CPU.

In-circuit emulator It emulates the system services so that all interactions between an application and the system can be monitored and traced. In-circuit emulators sit between the OS and the bare hardware and can watch and monitor all processes and all interactions between applications and OS.

Interpretive programming environment debugger Debugger is well integrated into the runtime interpreter and has very tight control over the running application.

SUMMARY

It is generally considered that debugging does not come under the domain of software testing. However, testing is wasteful if you do not remove the bugs reported and perform debugging. Therefore, debugging as a consequent process of testing is important to understand, both for the developers and the testers. This chapter shows the importance of the debugging process and its general process steps. Debugging has evolved with many techniques but it still gives better results if performed as an art. Various debugging techniques and debuggers have been discussed in this chapter along with the guidelines to perform the debugging.

Let us have a quick review of the important concepts described in this chapter:

- Debugging is the process of identification of symptoms of failures, tracing the bug, locating the errors that caused the bug, and correcting these errors.
- In debugging with the memory dump technique, printout of all registers and relevant memory locations is obtained and studied for any bug in the program.
- In debugging with watch points, at a particular point of execution in the program, value of variables or other actions can be verified. These particular points of execution are known as watch points.
- Output statements can be used to check the state of a condition or a variable at some watch point in the program.
- Breakpoint is actually a watch point inserted at various places in the program. But these insertions are not placed in the actual user program and therefore need not be removed manually like output statements.
- In single stepping, the users are able to watch execution of the program after every executable instruction.
- Debugger is a tool to help track down, isolate, and remove bugs from the software program.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Debugging is a part of testing domain.
   (a) true
   (b) false
   (c) none of the above
2. Viewing the debugging process as an art has given _____ results, as compared to technique.
   (a) poor
   (b) better
   (c) satisfactory
   (d) none of the above

3. The debugging process is to determine the exact nature of failure with the help of _____ identified.
   (a) symptoms
   (b) test cases
   (c) bugs
   (d) none of the above
4. Highly coupled and low-cohesive module is _____ to debug.
   (a) easy
   (b) difficult
   (c) cannot be debugged
   (d) none of the above
5. Technique in which a printout of all registers and relevant memory locations is obtained and studied, is called _____.
   (a) debugging
   (b) backtracking
   (c) debugging with watch points
   (d) debugging with memory dump
6. The technique in which the value of variables or other actions can be verified at a particular point of execution in the program is called _____.
   (a) debugging
   (b) backtracking
   (c) debugging with watch points
   (d) debugging with memory dump
7. Backtracking requires the debugger to have the knowledge of _____ of the system.
   (a) design
   (b) testing
   (c) code
   (d) none of the above

Review Questions

1. What is the importance of the debugging process?
2. Discuss the steps required for performing a debugging process.
3. Why is debugging considered a difficult process?
4. Take a debugger and give examples of all breakpoints in it, if supported.
5. Backtracking is considered an important debugging method. Take a project and list of its reported failures. Solve these failures with the help of backtracking.
6. Discuss various types of debuggers.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

INCOME TAX CALCULATOR
A Case Study

Step
1

Introduction to Case Study

All the techniques learnt in this book can be practised using a case study. For this purpose, a case study of *Income Tax Calculator* application has been taken. The application has been designed and developed for the readers and all the test case design techniques have been applied on it. However, the application presented and implemented is only for illustrative purposes and it is not claimed that this application is free from defects and can be used practically for calculating the income tax of a person. The idea is only to present a working application and show how to perform testing on it.

The case study has been presented in the following sequence:

Requirement Specifications and Verification

The requirements for the case study have been collected and SRS ver 1.0 was prepared initially. The tax slabs and other details in this case study have been compiled from *www.incometaxindia.gov.in*. This draft of SRS was in a raw form. After this, verification on SRS ver 1.0 was performed and found that many features were not present in SRS. During verification on SRS, the checklist presented in Appendix has been used. The readers are advised that they should also perform verification using checklists and find some more deficiencies in SRS. In this way, SRS ver 2.0 was prepared as a result of verification on SRS ver 1.0. Another round of verification was performed on SRS ver 2.0 and finally we get SRS ver 3.0.

The readers are advised to prepare an SDD of this application and perform verification exercises on it to and get a final version of SDD.

Black-box Testing on SRS ver 3.0

Once the SRS is prepared, some black-box test cases have been designed using the techniques studied in Chapter 4. The test cases can be executed on the implemented executable application. The executable application can be directly taken from the CD.

Source Code

The application based on SRS ver 3.0 has been implemented in C language. There are two files: *TaxCalculator.c* and *Taxcalculator.h*. The readers can get these files directly from the CD and use and modify them the way they want.

White-Box Testing

The source code of *TaxCalculator.h* has been tested using white-box testing techniques. All the major white-box testing techniques have been applied on this source code. The test cases can be executed on the implemented executable application. The executable application along with the source code of application can be taken directly from the CD.

The readers should follow this sequence for studying the full case study and learn the testing techniques presented in this book. The case study provides the way to learn the testing concepts and techniques in a practical way.

Step

2

Income Tax Calculator SRS ver 1.0

A system is proposed to calculate the income tax of a person residing in India, provided his salary, savings, status, and donations are known. The system will accept personal details, income details, savings details and calculate total salary, net tax payable, educational cess, and hence the total tax payable.

Income Tax slabs 2009/2010 for Men

Income: up to 1.5 lacs	NO INCOME TAX
Income : 1.5 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

Income Tax slabs 2009/2010 for Women

Income : up to 1.8 lacs	NO TAX
Income : 1.8 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

Income Tax slabs 2009/2010 for Senior Citizen

Income : up to 2.25 lacs	NO TAX
Income : 2.25 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

In addition to the income tax calculated according to the above income tax slabs, a 3% of education cess will be charged on the total income tax paid (not on the total taxable income). If the taxable income exceeds Rs 10 lacs, a 10% surcharge on the total income tax (not on the total taxable income) is also charged.

Donations with 100% rebate

- The Prime Minister's National Relief Fund.
- The Prime Minister's Armenia Earthquake Relief Fund.
- The Africa (Public Contributions-India) Fund.
- The National Foundation for Communal Harmony.
- A university or any educational institution of national eminence as maybe approved by the prescribed authority. Please note that the prescribed authority in case of a university or a non-technical institution of national eminence is the Director-General (Income-Tac exemption) in concurrence with the Secretary, UGC. In case of any technical institution of national eminence, the prescribed authority is the Director-General (Income-Tax Exemption) in concurrence with the Secretary, All India Council of Technical Education.
- The Maharashtra Chief Minister's Earthquake Relief Fund.
- Any Zila Saksharta Samiti constituted in any district under the chairmanship of the Collector of that district for the purpose of improvement of primary education in villages and towns in such a district and for literacy and post literacy activities.
- The National Blood Transfusion Council or any State Blood Transfusion council whose sole objective is the control, supervision, regulation, or encouragement in India of the services related to operation and requirements of blood banks.
- Any fund set up by a State Government to provide medical relief to the poor.
- The Army Central Welfare Fund or the Indian Naval Benevolent Fund or the Air Force Central Welfare Fund established by the armed forces of the Union for the welfare of the past and present members of such forces or their dependants.
- The Andhra Pradesh Chief Minister's Cyclone Relief Fund, 1996.
- The National Illness Assistance Fund.
- The Chief Minister's Relief Fund or the Lieutenant Governor's Relief Fund in any State or Union Territory.
- The Government, or any local authority, institution or association as maybe approved by the Central Government for the purpose of promoting family planning.

Step

3

Verification on Income Tax Calculator SRS ver 1.0

Verification on Income Tax Calculator SRS ver 1.0 is presented here. The reader is advised to use checklists provided in the Appendix while performing verification and use his/her intelligence. The missing features found in this verification are highlighted.

A system is proposed to calculate the income tax of a person residing in India provided his salary, savings, status, and donations are known. The system will accept personal details, income details, savings details and calculate total salary, net tax payable, educational cess, and hence total tax payable.

Is the software meant only for salaried person or for anyone? There is no mention about the functional flow of the system about how it works. Who will interface with the system? How are the savings considered in calculating the net tax? There is no high-level functionality diagram representing interfaces and data flow.

Income Tax slabs 2009/2010 for Men

Income: up to 1.5 lacs	NO INCOME TAX
Income : 1.5 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

Income Tax slabs 2009/2010 for Women

Income : up to 1.8 lacs	NO TAX
Income : 1.8 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

Income Tax slabs 2009/2010 for Senior Citizen

Income : up to 2.25 lacs	NO TAX
Income : 2.25 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

In addition to the income tax calculated according to the above income tax slabs, a 3% of education cess will be charged on the total income tax paid (not on the total taxable income). If the taxable income exceeds Rs 10 lacs, a 10% surcharge on the total income tax (not on the total taxable income) is also charged

Donations with 100% rebate

- The Prime Minister's National Relief Fund.
- The Prime Minister's Armenia Earthquake Relief Fund.
- The Africa (Public Contributions-India) Fund.
- The National Foundation for Communal Harmony.
- A University or any educational institution of national eminence as maybe approved by the prescribed authority. In case of any technical institution of national eminence, the prescribed authority is the Director General (Income-Tax Exemption) in concurrence with the Secretary, All India Council of Technical Education.
- The Maharashtra Chief Minister's Earthquake Relief Fund.
- Any Zila Saksharta Samiti constituted in any district under the chairmanship of the Collector of that district for the purpose of improvement of primary education in villages and towns in such a district and for literacy and post literacy activities.
- The National Blood Transfusion Council or any State Blood Transfusion council whose sole objective is the control, supervision, regulation, or encouragement in India of the services related to operation and requirements of blood banks.
- Any fund set up by a State Government to provide medical relief to the poor.
- The Army Central Welfare Fund or the Indian Naval Benevolent Fund or the Air Force Central Welfare Fund established by the armed forces of the Union for the welfare of the past and present members of such forces or their dependants.
- The Andhra Pradesh Chief Minister's Cyclone Relief Fund, 1996.
- The National Illness Assistance Fund.
- The Chief Minister's Relief Fund or the Lieutenant Governor's Relief Fund in any State or Union Territory.
- The Government, or any local authority, institution or association as maybe approved by the Central Government for the purpose of promoting family planning.

What will be the rebate if the donation is not in the above list? In addition, the following items are missing in the SRS which are necessary to avoid bugs and misunderstanding:

1. High-level diagrams depicting external and internal interfaces.
2. The user interaction with the system.
3. Software Functions/Features.
4. Inputs and outputs formats and their ranges.
5. Software/Hardware Requirements.

Step

4

Income Tax Calculator SRS ver 2.0

A system is proposed to calculate the income tax of a person residing in India provided his income, savings, status, and donations are known. The system will accept personal details, income details, and savings details, and calculate the total salary, net tax payable, educational cess, and hence the total tax payable. The user gets the information about total tax to be paid.

The system will first accept personal details, income, donations, and savings. For donations, it provides a list of categories in which 100% rebate is provided. The user will look for the option provided and inform the system whether the donation lies in that list. If the donation lies in the list, 100% rebate will be provided, otherwise 50%. The system will check whether the savings are less than Rs 1 lac. If yes, then the whole amount will be deducted from the taxable income. Otherwise, Rs 1 lac will be deducted. Then the system will calculate the total tax and check if it exceeds Rs 10 lacs. If yes, a 10% surcharge on the total income tax (not on the total taxable income) is also charged and a 3% of education cess will be charged on the total income tax paid (not on the total taxable income). Finally, the system will show the net tax as per the following details:

Income Tax slabs 2009/2010 for Men

Income: up to 1.5 lacs	NO TAX
Income : 1.5 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

Income Tax slabs 2009/2010 for Women

Income : up to 1.8 lacs	NO TAX
Income : 1.8 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

Income Tax slabs 2009/2010 for Senior Citizen

Income : up to 2.25 lacs	NO TAX
Income : 2.25 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

Donations with 100% rebate

- The Prime Minister's National Relief Fund.
- The Prime Minister's Armenia Earthquake Relief Fund.
- The Africa (Public Contributions-India) Fund.
- The National Foundation for Communal Harmony.
- A university or any educational institution of national eminence as maybe approved by the prescribed authority. In case of any technical institution of national eminence, the prescribed authority is the Director General (Income-Tax Exemption) in concurrence with the Secretary, All India Council of Technical Education.
- The Maharashtra Chief Minister's Earthquake Relief Fund.
- Any Zila Saksharta Samiti constituted in any district under the chairmanship of the Collector of that district for the purpose of improvement of primary education in villages and towns in such a district and for literacy and post literacy activities.
- The National Blood Transfusion Council or any State Blood Transfusion council whose sole objective is the control, supervision, regulation, or encouragement in India of the services related to operation and requirements of blood banks.
- Any fund set up by a State Government to provide medical relief to the poor.
- The Army Central Welfare Fund or the Indian Naval Benevolent Fund or the Air Force Central Welfare Fund established by the armed forces of the Union for the welfare of the past and present members of such forces or their dependants.

- The Andhra Pradesh Chief Minister's Cyclone Relief Fund, 1996.
- The National Illness Assistance Fund.
- The Chief Minister's Relief Fund or the Lieutenant Governor's Relief Fund in any State or Union Territory.
- The Government, or any local authority, institution or association as maybe approved by the Central Government for the purpose of promoting family planning.

Functional Requirements

Accept Personal Details

The function will accept the following details to be entered by user.

- Name (3 to 15 alphabets with spaces in between)
- Date of Birth (dd/mm/yyyy)
- Permanent address (3 to 30 characters)
- Sex (M/F one alphabet only)
- Status: Salaried or not (Y/N one alphabet only)

If the user enters the answer Y (Yes) to the status entry, then the function will display the following three entries, otherwise it will not.

- Designation (if Salaried) (3 to 15 alphabets)
- Name of the employer (if salaried) (3 to 25 alphabets with spaces)
- Address of the employer (if salaried) (3 to 30 characters)
- PAN number (10 characters including alphabets and digits 0–9)

- TDS circle where annual return/statement under section 206 is to be filed (3 to 15 alphabets with spaces)
- Period: From (dd/mm/yyyy)
  To (dd/mm/yyyy)
  Assessment year (yyyy-yy)

Accept Income Details

The function will enquire whether the user is a salaried person or has some other source of income. If the user is not a salaried person, the system will ask for the source of income. The user may enter various types of source of incomes as given below:

Source of Income: (3 to 20 alphabets with spaces)

Amount: (positive real numbers with maximum two decimal places)

The function will aggregate all the amounts of income as gross total income.

If the person is salaried, the function asks for the following details:

1. **Gross Salary**
   (a) Salary as per the provisions contained in the section 17(1)
   (b) Value of the prerequisites under section 17(2) (As per form number 12BA, wherever applicable)
   (c) Profits in lieu of salary under section 17(3) (As per form number 12BA, wherever applicable)
   (d) Total (to be calculated by this function)
2. **Less allowance to the extent exempt under section 10**
   This function will add the exempted allowances.
3. **Balance**
   This function will calculate the difference of the gross salary and the exempted allowances.
4. **Deductions**
   - Entertainment allowance
   - Tax on employment
5. **Aggregate**
   This function will calculate the aggregate of the deductions entered above.
6. **Income Chargeable Under The Head 'Salaries'**
   The function will calculate the difference of item 3–item 5.
7. **Add**
   Any other item reported by the employee.
   User may enter multiple incomes. The function will add all these incomes.
8. **Gross Total Income**
   The function will add item 6 and 7.
   All the amounts will be positive real numbers with maximum 2 decimal places.

Accept Savings & Donations Details

The function will ask the user to enter the total savings and the donations in the following format.

Saving type (3 to 20 alphabets with spaces)

Deductible amount (positive real numbers with maximum two decimal places)

The user may enter multiple savings. The function will add all the deductible amounts in aggregate deductible amount.

Accept Tax Deduction Details

If the person is salaried, then this function will accept the details if tax deducted by the employer during the year is in the following format:

Amount of tax deposited (positive real numbers with maximum two decimal places)

Date (dd/mm/yyyy)

Challan Number (5 to 20 characters)

The above details may be entered multiple times. The function will add all the amounts of tax deposited.

Amount of TDS (positive real numbers with maximum two decimal places)

The function will add all the amounts of tax deposited and the amount of TDS in the total tax deducted.

Generate Tax Payable Details

This function calculates the tax payable by the person in the following format:

Taxable income Function will calculate this by taking difference of gross total income in the function Accept Income Details and aggregate deductible amount in the function Accept Savings & Donation Details.

Tax on taxable income Function will calculate this using the appropriate slab of user as given above.

Surcharge Function will calculate the surcharge as if tax on taxable income exceeds Rs 10 lacs, a 10% surcharge is charged.

Education cess Function will calculate the education cess as a 3% of tax on taxable income.

Tax payable Function will sum up tax on taxable income, surcharge, and education cess.

Relief under section 89 User will enter the amount, if applicable.

Tax payable after relief (if applicable) Function will deduct relief amount from tax payable.

Total tax deducted Displayed from the function Accept Tax Deduction Details.

Tax payable/refundable The function will find the difference of tax payable and the total tax deducted. If the difference is positive, then this amount is the net tax to be paid by the person, otherwise the amount is due on the government to be refunded.

Step

5

Verification on Income Tax Calculator SRS ver 2.0

Verification on Income Tax Calculator SRS ver 2.0 is presented here. The reader is advised to use checklists while performing verification and use his/her intelligence. The missing features found in this verification are highlighted.

A system is proposed to calculate the income tax of a person residing in India provided his income, savings, status, and donations are known. The system will accept personal details, income details, savings details, and calculate total salary, net tax payable, educational cess, and hence total tax payable. The user gets the information about total tax to be paid.

The system will first accept personal details, income, donations, and savings. For donations, it provides a list of categories in which 100% rebate is provided. The user will look for the option provided and informs the system whether the donation lies in that list. If the donation lies in the list, 100% rebate will be provided otherwise 50%. The system will check whether the savings are less than Rs 1 lac. If yes, then whole amount will be deducted from the taxable income. Otherwise, Rs 1 lac will be deducted. Then the system will calculate the total tax and checks if it exceeds Rs 10 lacs, a 10% surcharge on the total income tax (not on the total taxable income) is also charged and a 3% of education cess will be charged on the total income tax paid (not on the total taxable income). Finally, the system will show the net tax as per the following details:

Income Tax slabs 2009/2010 for Men

Income: up to 1.5 lacs	NO TAX
Income : 1.5 lacs to 3 lacs	10 %
Income : 3 lacs to 5 lacs	20 %
Income : above 5 lacs	30 %

Income Tax slabs 2009/2010 for Women

Income : up to 1.8 lacs	NO TAX
Income : 1.8 lacs to 3 lacs	10 %
Income : 3 lacs to 5 lacs	20 %
Income : above 5 lacs	30 %

Income Tax slabs 2009/2010 for Senior Citizen

Income : up to 2.25 lacs	NO TAX
Income : 2.25 lacs to 3 lacs	10 %
Income : 3 lacs to 5 lacs	20 %
Income : above 5 lacs	30 %

Donations with 100% rebate

- The Prime Minister's National Relief Fund.
- The Prime Minister's Armenia Earthquake Relief Fund.
- The Africa (Public Contributions-India) Fund.
- The National Foundation for Communal Harmony.
- A University or any educational institution of national eminence as maybe approved by the prescribed authority. In case of any technical institution of national eminence, the prescribed authority is the Director General (Income-Tax Exemption) in concurrence with the Secretary, All India Council of Technical Education.
- The Maharashtra Chief Minister's Earthquake Relief Fund.
- Any Zila Saksharta Samiti constituted in any district under the chairmanship of the Collector of that district for the purpose of improvement of primary education in villages and towns in such a district and for literacy and post literacy activities.
- The National Blood Transfusion Council or any State Blood Transfusion council whose sole objective is the control, supervision, regulation, or encouragement in India of the services related to operation and requirements of blood banks.
- Any fund set up by a State Government to provide medical relief to the poor.

- The Army Central Welfare Fund or the Indian Naval Benevolent Fund or the Air Force Central Welfare Fund established by the armed forces of the Union for the welfare of the past and present members of such forces or their dependants.
- The Andhra Pradesh Chief Minister's Cyclone Relief Fund, 1996.
- The National Illness Assistance Fund.
- The Chief Minister's Relief Fund or the Lieutenant Governor's Relief Fund in any State or Union Territory.
- The Government, or any local authority, institution or association as maybe approved by the Central Government for the purpose of promoting family planning.

Functional Requirements

Accept Personal Details

The function will accept the following details to be entered by user.

- Name (3 to 15 alphabets with spaces in between)
- Date of Birth (dd/mm/yyyy)
- Permanent address (3 to 30 characters)

The address should not contain any character. The allowed characters like alphabets, digits, spaces, and commas should be mentioned.

- Sex (M/F one alphabet only)
- Salaried or not (Y/N one alphabet only)

- Designation (if Salaried) (3 to 15 alphabets)
- Name of the employer (if salaried) (3 to 25 alphabets with spaces)
- Address of the employer (if salaried) (3 to 30 characters)
- PAN number (10 characters including alphabets and digits 0-9)

Including means any character can be entered. The word should be *consisting*.

- TDS circle where annual return/ statement under section 206 is to be filed (3 to 15 alphabets with spaces)
- Period: From (dd/mm/yyyy)
- To (dd/mm/yyyy)
- Assessment year (yyyy-yy)

It is not clear whether the user can make a wrong entry and move ahead to the next entry or he cannot move ahead until he enters a correct entry.

Accept Income Details

The function will enquire whether the user is a salaried person or has some other source of income. If the user is not a salaried person, the system will ask for the source of income. The user may enter various types of source of incomes as given below:

Source of Income: (3 to 20 alphabets with spaces)

Amount: (positive real numbers with maximum two decimal places)

The function will aggregate all the amounts of income as gross total income. If the person is salaried, the function asks for the following details:

1. **Gross Salary**
   (a) Salary as per the provisions contained in the section 17(1)
   (b) Value of the prerequisites under section 17(2) (As per form number 12BA, wherever applicable)
   (c) Profits in lieu of salary under section 17(3) (As per form number 12BA, wherever applicable)
   (d) Total (to be calculated by this function)
2. **Less allowance to the extent exempt under section 10**

   This function will add the exempted allowances.
3. **Balance**

   This function will calculate the difference of the gross salary and the exempted allowances.
4. **Deductions**
   - Entertainment allowance
   - Tax on employment

5. **Aggregate**

   This function will calculate the aggregate of the deductions entered above.

6. **Income Chargeable Under the Head 'Salaries'**

   The function will calculate the difference of item 3 – item 5.

7. **ADD**

   Any other item reported by the employee

   User may enter multiple incomes. The function will add all these incomes.

8. **Gross Total Income**

   The function will add item 6 and 7.

   All the amounts will be positive decimal numbers with maximum 2 decimal places.

It is not clear whether the user can make a wrong entry and move ahead on the next entry or he cannot move ahead until he enters a correct entry.

Accept Savings & Donations Details

The function will ask the user to enter the total savings and the donations in the following format.

Saving Type: (3 to 20 alphabets with spaces)

Deductible amount: (positive real numbers with maximum two decimal places)

The user may enter multiple savings. The function will add all the deductible amounts in aggregate deductible amount.

It is not clear whether the user can make a wrong entry and move ahead to the next entry or he cannot move ahead until he enters a correct entry.

Accept Tax Deduction Details

If the person is salaried, then this function will accept the details if tax deducted by the employer during the year in the following format:

- Amount of tax deposited (positive real numbers with maximum two decimal places)
- Date (dd/mm/yyyy)
- Challan Number (5 to 20 characters)

The above details may be entered multiple times. The function will add all the amounts of tax deposited.

Amount of TDS The function will add all the amounts of tax deposited and amount of TDS in total tax deducted.

It is not clear whether the user can make a wrong entry and move ahead to the next entry or he cannot move ahead until he enters a correct entry.

Generate Tax Payable Details

This function calculates the tax payable by the person in the following format:

Taxable income The function will calculate this by taking difference of gross total income in the function Accept Income Details and aggregate deductible amount in the function Accept Savings & Donation Details.

Tax on taxable income The function will calculate this using the appropriate slab of user as given above.

Surcharge The function will calculate the surcharge as if tax on taxable income exceeds Rs 10 lacs, a 10% surcharge is charged.

Education cess The function will calculate the education cess as a 3% of tax on taxable income.

Tax payable The function will sum up the tax on taxable income, surcharge, and education cess.

Relief under section 89 User will enter the amount, if applicable.

Tax payable after relief (if applicable) The function will deduct the relief amount from the tax payable.

Total tax deducted Displayed from the function Accept Tax Deduction Details.

Tax payable/refundable The function will find difference of tax payable and total tax deducted. If difference is positive, then this amount is the net tax to be paid by the person, otherwise the amount is due on the government to be refunded.

User interface requirements and system requirements are not mentioned.

Step

6

Income Tax Calculator SRS ver 3.0

A system is proposed to calculate the income tax of a person residing in India provided his income, savings, status, and donations are known. The system will accept personal details, income details, savings details and calculate total salary, net tax payable, educational cess, and hence total tax payable. The user gets the information about total tax to be paid.

The system will first accept personal details, income, donations, and savings. For donations, it provides a list of categories in which 100% rebate is provided. The user will look for the option provided and informs the system whether the donation lies in that list. If the donation lies in the list, 100% rebate will be provided, otherwise 50%. The system will check whether the savings are less than Rs 1 lac. If yes, then the whole amount will be deducted from the taxable income. Otherwise, Rs 1 lac will be deducted. Then the system will calculate the total tax and checks if it exceeds Rs 10 lacs, a 10% surcharge on the total income tax (not on the total taxable income) is also charged and a 3% of education cess will be charged on the total income tax paid (not on the total taxable income). Finally, the system will show the net tax as per the following details:

Income Tax slabs 2009/2010 for Men

Income: up to 1.5 lacs	NO TAX
Income : 1.5 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

Income Tax slabs 2009/2010 for Women

Income : up to 1.8 lacs	NO TAX
Income : 1.8 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

Income Tax slabs 2009/2010 for Senior Citizen

Income : up to 2.25 lacs	NO TAX
Income : 2.25 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

Donations with 100% rebate

- The Prime Minister's National Relief Fund.
- The Prime Minister's Armenia Earthquake Relief Fund.
- The Africa (Public Contributions-India) Fund.
- The National Foundation for Communal Harmony.
- A University or any educational institution of national eminence as maybe approved by the prescribed authority. In case of any technical institution of national eminence, the prescribed authority is the Director General (Income-Tax Exemption) in concurrence with the Secretary, All India Council of Technical Education.
- The Maharashtra Chief Minister's Earthquake Relief Fund.
- Any Zila Saksharta Samiti constituted in any district under the chairmanship of the Collector of that district for the purpose of improvement of primary education in villages and towns in such a district and for literacy and post literacy activities.
- The National Blood Transfusion Council or any State Blood Transfusion council whose sole objective is the control, supervision, regulation, or encouragement in India of the services related to operation and requirements of blood banks.
- Any fund set up by a State Government to provide medical relief to the poor.
- The Army Central Welfare Fund or the Indian Naval Benevolent Fund or the Air Force Central Welfare Fund established by the armed forces of the

Union for the welfare of the past and present members of such forces or their dependants.

- The Andhra Pradesh Chief Minister's Cyclone Relief Fund, 1996.
- The National Illness Assistance Fund.
- The Chief Minister's Relief Fund or the Lieutenant Governor's Relief Fund in any State or Union Territory.
- The Government, or any local authority, institution or association as maybe approved by the Central Government for the purpose of promoting family planning.

Functional Requirements

Accept Personal Details (APD)

The function will accept the following details to be entered by the user. The user cannot move to the next entry unless he enters the correct entry.

- Name (3 to 15 alphabets with spaces in between)
- Date of Birth (dd/mm/yyyy)
- Permanent address (3 to 30 characters) The allowed characters are alphabets, digits, spaces, and commas only.
- Sex (M/F one alphabet only)
- Status: Salaried or not (Y/N one alphabet only)

If the user enters the answer Y (Yes) to the status entry, then the function will display the following three entries, otherwise it will not.

- Designation (if salaried) (3 to 15 alphabets)
- Name of the employer (if salaried) (3 to 25 alphabets with spaces)
- Address of the employer (if salaried) (3 to 30 characters)
- PAN number (10 characters consisting alphabets and digits 0–9)
- TDS circle where annual return/statement under section 206 is to be filed (3 to 15 alphabets with spaces)
- Period: From (dd/mm/yyyy)
  To (dd/mm/yyyy)
  Assessment year (yyyy-yy)

Accept Income Details (AID)

The function will enquire whether the user is a salaried person or has some other source of income. If the user is not a salaried person, the system will ask for the source of income. The user may enter various types of source of incomes as given below. The user cannot move to the next entry unless he enters the correct entry.

Source of Income: (3 to 20 alphabets with spaces)

Amount: (positive real numbers with maximum two decimal places)

The function will aggregate all the amounts of income as gross total income. If the person is salaried, the function asks for the following details:

1. **Gross Salary**
   (a) Salary as per the provisions contained in the section 17(1)
   (b) Value of the perquisites under section 17(2) (As per form number 12BA, wherever applicable)
   (c) Profits in lieu of salary under section 17(3) (As per form number 12BA, wherever applicable)
   (d) Total (to be calculated by this function)
2. **Less allowance to the extent exempt under section 10**
   This function will add the exempted allowances.
3. **Balance**
   This function will calculate the difference of the gross salary and the exempted allowances.
4. **Deductions**
   - Entertainment allowance (EA)
   - Tax on employment (TE)
5. **Aggregate**
   This function will calculate the aggregate of the deductions entered above.
6. **Income Chargeable Under the Head 'Salaries'**
   The function will calculate the difference of item 3 – item 5.

7. **Add**
   Any other item reported by the employee
   User may enter multiple incomes. The function will add all these incomes.
8. **Gross Total Income**
   The function will add item 6 and 7.
   All the amounts will be positive real numbers with maximum 2 decimal places.

Accept Savings & Donations Details (ASD)

The function will ask the user to enter the total savings and the donations in the following format.

Saving Type: (3 to 20 alphabets with spaces)

Deductible amount: (positive real numbers with maximum two decimal places)

The user may enter multiple savings. The function will add all the deductible amounts in aggregate deductible amount.

The user cannot move to the next entry unless he enters the correct entry.

Accept Tax Deduction Details

If the person is salaried, then this function will accept the details if tax deducted by the employer during the year in the following format:

- Amount of TDS: (positive real numbers with maximum two decimal places)
- Amount of tax deposited (positive real numbers with maximum two decimal places)
- Date (dd/mm/yyyy)
- Challan Number (5 to 20 characters)

The above details may be entered multiple times. The function will add all the amounts of tax deposited and amount of TDS in total tax deducted.

The user cannot move to the next entry unless he enters the correct entry.

Generate Tax Payable Details

This function calculates the tax payable by the person in the following format:

Taxable income Function will calculate this by taking difference of gross total income in the function Accept Income Details and aggregate deductible amount in the function Accept Savings & Donation Details.

Tax on taxable income Function will calculate this using the appropriate slab of user as given above.

Surcharge Function will calculate the surcharge as if tax on taxable income exceeds Rs 10 lacs, a 10% surcharge is charged.

Education cess Function will calculate the education cess as a 3% of tax on taxable income.

Tax payable Function will sum up tax on taxable income, surcharge, and education cess.

Total tax deducted Displayed from the function Accept Tax Deduction Details.

Tax payable/refundable The function will find difference of tax payable and total tax deducted. If difference is positive, then this amount is the net tax to be paid by the person, otherwise the amount is due on the government to be refunded.

User Interface Requirements

Personal Detail Screen

The personal detail screen is displayed wherein the user can enter his various details as given below:

- Name
- Date of Birth
- Permanent address
- Sex (M/F)
- Status (Salaried or not): Y/N

If the user presses Y, then the following screen is displayed:

- Designation
- Name of the employer
- Address of the employer
- PAN number
- TDS circle where annual return/statement under section 206 is to be filed
- Period: From (dd/mm/yyyy)
  To (dd/mm/yyyy)
  Assessment year (yyyy-yy)

If the user presses N, then the following screen is displayed:

- PAN number
- TDS circle where annual return/statement under section 206 is to be filed
- Period: From (dd/mm/yyyy)
  To (dd/mm/yyyy)
  Assessment year (yyyy-yy)

Income Details Screen

The income detail screen is displayed wherein the user can enter his various details as given below.

If the user is not a salaried person, the system will ask for the source of income in the following screen:

- Source of Income:
- Amount:
- Enter more? (Y/N)

If the person is salaried, the function asks for the following details:

1. **Gross Salary**
   (a) Salary as per the provisions contained in the section 17(1)
   (b) Value of the perquisites under section 17(2) (As per form number 12BA, wherever applicable)
   (c) Profits in lieu of salary under section 17(3) (As per form number 12BA, wherever applicable)
   (d) Total (to be calculated by this function)
2. **Allowance to the extent exempt under section 10**
   - Enter more? (Y/N)
3. **Deductions**
   - Entertainment allowance (EA)
   - Tax on employment (TE)
4. **Income Chargeable Under the Head 'Salaries'**
   - Displayed by the system
5. **Any other item reported by the employee**
   - Enter Income:
   - Enter more? (Y/N)
6. **Gross Total Income:** Displayed by the system

Savings & Donations Details Screen

The saving detail screen is displayed wherein the user can enter his various details as given below:

Saving Type: (3 to 20 alphabets with spaces)

Deductible amount: Enter more? (Y/N)

Tax Deduction Details Screen

If the person is salaried, then tax deducted by the employer during the year in the following format is entered:

- Amount of tax deposited
- Date
- Challan Number
- Enter more? (Y/N)
- Amount of TDS:

Tax Payable Detail Screen

This screen is the report screen generated by the system, not modifiable by the user.

Step

7

Black-Box Testing on Units/Modules of Income Tax Calculator SRS ver 3.0

Black-Box Testing on various modules of Income Tax Calculator SRS ver 3.0 is presented here. The details of the various modules have been reproduced to understand the functionality of the module and later, the black-box test cases of modules have been given. The reader is advised to execute these test cases on the running executable application provided in the CD.

Accept Personal Details (APD)

The function will accept the following details to be entered by the user. The user cannot move to the next entry unless he enters the correct entry.

- Name (3 to 15 alphabets with spaces in between)
- Date of Birth (dd/mm/yyyy)
- Permanent address (3 to 30 characters) The allowed characters are alphabets, digits,
- Spaces, and commas only.
- Sex (M/F one alphabet only)
- Status: Salaried or not (Y/N one alphabet only)

If the user enters the answer Y (Yes) to the status entry, then the function will display the following three entries, otherwise it will not.

- Designation (if salaried) (3 to 15 alphabets)
- Name of the employer (if salaried) (3 to 25 alphabets with spaces)
- Address of the employer (if salaried) (3 to 30 characters)
- PAN number (10 characters consisting alphabets and digits 0–9)
- TDS circle where annual return/statement under section 206 is to be filed (3 to 15 alphabets with spaces)
- Period: From (dd/mm/yyyy)
- To (dd/mm/yyyy)
- Assessment year (yyyy-yy)

This module will be tested with equivalence class partitioning methods.

Using the information of the module, the following equivalence classes are generated:

C1 = {3 ≤ Name ≤ 15}

C2 = {Name < 3}

C3 = {Name > 15}

C4 = {Name: Any invalid character other than alphabets and spaces between the alphabets}

C5 = {Name: Blank}

C6 = {Date of Birth: digits only}

C7 = {Date of Birth: Any invalid character other than digit}

C8 = {Date of Birth: Blank}

C9 = {3 ≤ Permanent address ≤ 30}

C10 = {Permanent address < 3}

C11 = {Permanent address > 30}

C12 = {Permanent address: Any invalid character other than alphabets, digits, and spaces between them}

C13 = {Permanent address: Blank}

C14 = {Sex: M/F}

C15 = {Sex: any character other than M/F}

C16 = {Sex: Blank}

C17 = {Status: Y/N}

C18 = {Status: any character other than Y/N}

C19 = {Status: Blank}

C20 = {3 ≤ Designation ≤ 15}

C21 = {Designation < 3}

C22 = {Designation > 15}

C23 = {Designation: Any invalid character other than alphabets and spaces between the alphabets}

C24 = {Designation : Blank}

C25 = {3 ≤ Name of Employer ≤ 25}

C26 = {Name of Employer < 3}

C27 = {Name of Employer > 25}

C28 = {Name of Employer: Any invalid character other than alphabets and spaces between the alphabets}

C29 = {Name of Employer: Blank}

C30 = {Address of the employer < 3}

C31 = {Address of the employer > 30}

C32 = {Address of the employer: Any invalid character other than alphabets, digits and spaces between them}

C33 = {Address of the employer: Blank}

C34 = {PAN number: 10 characters consisting of alphabets and digits only}

C34 = {PAN number < 10}

C34 = {PAN number > 10}

C34 = {PAN number: 10 characters consisting of any invalid character other than alphabets and digits only}

C35 = {PAN number: Blank}

C36 = {3 ≤ TDS Circle ≤ 15}

C37 = {TDS Circle < 3}

C38 = {TDS Circle > 15}

C39 = {TDS Circle: Any invalid character other than alphabets and spaces between the alphabets}

C40 = {TDS Circle: Blank}

C41 = {Period From: digits only}

C42 = {Period From: Any invalid character other than digit}

C43 = {Period From: Blank}

C44 = {Period To: digits only}

C45 = {Period To: Any invalid character other than digit}

C46 = {Period To: Blank}

After preparing the classes for this function, one test case per class should be designed as given below:

Test case ID	Class covered	Name	Date of Birth	Permanent Address	...	Expected Output
APD1	C1	Harish				Normal behaviour
APD2	C2	Na				Name too short
APD3	C3	Abdul Ghaffar Khan				Name too long
APD4	C4	;'.,sfh';.,s				Name should contain only alphabets and spaces.
APD5	C5					Please enter a valid name entry.

APD6	C6	Harish	01/07/1984			Normal behaviour
APD7	C7	Harish	Ab/07/qw12			Date should contain only digits.
APD8	C8	Harish				Please enter a valid Date entry.
....	...	...	...	...	...	...

Accept Income Details (AID)

The function will enquire whether the user is a salaried person or has some other source of income. If the user is not a salaried person, the system will ask for the source of income. The user may enter various types of source of incomes as given below. The user cannot move to the next entry unless he enters the correct entry.

Source of Income: *(3 to 20 alphabets with spaces)*

Amount: *(positive real numbers with maximum two decimal places)*

These two entries can be tested with equivalence class partitioning methods. The classes are:

C1 = {3 ≤ Source of Income ≤ 20}

C2 = {Source of Income < 3}

C3 = {Source of Income > 20}

C4 = {Source of Income: Blank}

C5 = {Amount: positive real numbers with maximum two decimal places}

C6 = {Amount: negative number or characters other than digit}

C7 = {Amount: Blank }

After preparing the classes, one test case per class should be designed as given below:

Test Case ID	Class covered	Source of Income	Amount	Expected Output
AID1	C1	Agriculture		Normal behaviour
AID2	C2	Tc		Entry too short
AID3	C3	Income from consultancy		Entry too long
AID4	C4			Please enter the source of income
AID5	C5	Agriculture	40000.00	Normal behaviour
AID6	C6	Agriculture	Rs 23000.00	Please enter the positive real numbers only
AID7	C7	Agriculture		Please enter amount

The function will aggregate the total amount of income as gross total income.

If the person is salaried, the function asks for the following details:

1. **Gross Salary**
   (a) Salary as per the provisions contained in the section 17(1)
   (b) Value of the perquisites under section 17(2) (As per form number 12BA, wherever applicable)
   (c) Profits in lieu of salary under section 17(3) (As per form number 12BA, wherever applicable)
   (d) Total (to be calculated by this function)

The entries, a, b, c can also be tested with equivalence classes.

C8 = {a: valid entry}

C9 = {a: invalid entry}

C10 = {a:blank}

C11 = {b: valid entry}

C12 = {b: invalid entry}

C13 = {b:blank}

C14 = {c: valid entry}

C15 = {c: invalid entry}

C16 = {c: blank}

Therefore, the test cases will be as follows.

Test Case ID	Class covered	a	b	c	Expected Output
AID8	C8	2000.00			Normal Behaviour
AID9	C9	–1200			Please enter a valid entry.
AID10	C10				Please enter a valid amount.
AID11	C11	2000.00	1000.00		Normal Behaviour
AID12	C12	2000.00	Rs 200		Please enter a valid entry.
AID13	C13	2000.00			Please enter a valid amount.
AID14	C14	2000.00	1000.00	3000.00	Normal Behaviour
AID15	C15	2000.00	1000.00	Rs 200	Please enter a valid entry.
AID16	C16	2000.00	1000.00		Please enter a valid amount.

2. **Less allowance to the extent exempt under section 10**

   This function will add the exempted allowances.

3. **Balance**

   This function will calculate the difference of the gross salary and the exempted allowances.

4. **Deductions**
   - Entertainment allowance (EA)
   - Tax on employment (TE)

These two entries can be tested with equivalence class partitioning methods. The classes are:

C17 = {EA: valid entry}
C18 = {EA: invalid entry}
C19 = {EA: Blank}
C20 = {TE: valid entry}
C21 = {TE: invalid entry}
C22 = {TE: Blank}

The test cases will be as follows:

Test Case ID	Class covered	EA	TE	Expected Output
AID17	C17	2000.00		Normal behaviour
AID18	C18	–200		Please enter a valid amount.
AID19	C19			Please enter a valid amount.
AID20	C20	2000.00	100.00	Normal behaviour
AID21	C21	2000.00	@12.00	Please enter a valid amount.
AID22	C22	2000.00		Please enter a valid amount.

5. **Aggregate**
   This function will calculate the aggregate of the deductions entered above.
6. **Income Chargeable Under the Head 'Salaries'**
   The function will calculate the difference of item 3 – item 5.
7. **Add**
   Any other item reported by the employee
   User may enter multiple incomes. The function will add all these incomes.

This entry can be tested with equivalence class partitioning methods. The classes are:

C23 = {Income: valid entry}
C24 = {Income: invalid entry}
C25 = {Income: Blank}

The test cases will be:

Test Case ID	Class covered	Income	Expected Output
AID23	C23	200.00	Normal behaviour
AID24	C24	–1000	Please enter a valid amount.
AID25	C25		Please enter a valid amount.

8. **Gross Total Income**
   The function will add item 6 and 7.

All the amounts will be positive real numbers with a maximum of two decimal places.

Accept Savings & Donations Details (ASD)

The function will ask the user to enter the total savings and the donations in the following format.

Saving Type: (3 to 20 alphabets with spaces)

Deductible amount: (positive real numbers with maximum two decimal places)

The user may enter multiple savings. The function will add all the deductible amounts in aggregate deductible amount.

The user cannot move to the next entry unless he enters the correct entry.

These two entries can be tested with equivalence class partitioning methods. The classes are:

C1 = {3 ≤ Saving Type ≤ 20}

C2 = {Saving Type < 3}

C3 = {Saving Type > 20}

C4 = {Saving Type: Blank}

C5 = {Deductible amount: positive real numbers with maximum two decimal places}

C6 = {Deductible amount: negative number or characters other than digit}

C7 = {Deductible amount: Blank }

After preparing the classes, one test case per class should be designed as given below:

Test Case ID	Class covered	Saving Type	Deductible Amount	Expected Output
ASD1	C1	LIC		Normal behaviour
ASD2	C2	MF		Entry too short
ASD3	C3	National saving certificate		Entry too long
ASD4	C4			Please enter the saving type
ASD5	C5	LIC	2000.00	Normal behaviour
ASD6	C6	LIC	Rs 23000.00	Please enter the positive real numbers only
ASD7	C7	LIC		Please enter amount

Accept Tax Deduction Details (ATD)

If the person is salaried, then this function will accept the details if tax deducted by the employer during the year is in the following format:

- Amount of tax deposited (positive real numbers with maximum two decimal places)
- Date (dd/mm/yyyy)
- Challan Number (5 to 20 characters)

The above details may be entered multiple times. The function will add the total amounts of tax deposited.

The user cannot move to the next entry unless he enters the correct entry.

Amount of TDS: (positive real numbers with maximum two decimal places)

Testing of these entries can be done with equivalence class partitioning methods. The classes are:

C1 = {Amount of tax deposited: valid entry}

C2 = {Amount of tax deposited: invalid entry}

C3 = {Amount of tax deposited: Blank}

C4 = {Date : digits only}

C5 = {Date : Any invalid character other than digit}

C6 = {Date : Blank}

C7 = {5 ≤ Challan Number ≤ 20}

C8 = {Challan Number < 5}

C9 = {Challan Number > 20}

C10 = {Challan Number: Blank}

Test Case ID	Class covered	Amount of tax deposited	Date	Challan Number	Expected Output
ATD1	C1	2000			Normal behaviour
ATD2	C2	Rs 2000			Please enter the valid amount.
ATD3	C3				Blank Entry. Please enter the valid amount.
ATD4	C4	2000	12/02/2009		Normal behaviour
ATD5	C5	2000	12/feb/2009		Please enter a valid date.
ATD6	C6	2000			Blank entry. Please enter a valid date.
ATD7	C7	2000	12/02/2009	SDE345	Normal behaviour
ATD8	C8	2000	12/02/2009	SDE	Challan number too short
ATD9	C9	2000	12/02/2009	Sdefrtg5667 89asdf5678	Challan number too long
ATD10	C10	2000	12/02/2009		Blank entry. Please enter challan number.

The testing of date field can be done separately with the help of BVA technique. So, now we will test this field with the following specifications:

$1 \leq mm \leq 12$

$1 \leq dd \leq 31$

$2009 \leq yyyy \leq 2099$

Test Cases Using BVC

	Month	Day	Year
Min value	1	1	2009
Min^+ value	2	2	2010
Max value	12	31	2099
Max^- value	11	30	2098
Nominal value	6	15	2060

Using these values, test cases can be designed as shown below:

Test Case ID	Month	Day	Year	Expected Output
1	1	15	2060	Normal behaviour
2	2	15	2060	Normal behaviour
3	11	15	2060	Normal behaviour
4	12	15	2060	Normal behaviour
5	6	1	2060	Normal behaviour
6	6	2	2060	Normal behaviour
7	6	30	2060	Normal behaviour
8	6	31	2060	Invalid input
9	6	15	2009	Normal behaviour
10	6	15	2010	Normal behaviour
11	6	15	2098	Normal behaviour
12	6	15	2099	Normal behaviour
13	6	15	2060	Normal behaviour

Test Cases Using Robust Testing

	Month	Day	Year
Min^- value	0	0	2008
Min value	1	1	2009
Min^+ value	2	2	2010
Max value	12	31	2099
Max^- value	11	30	2098
Max^+ value	13	32	3000
Nominal value	6	15	2060

Using these values, test cases can be designed as shown below:

Test Case ID	Month	Day	Year	Expected Output
1	0	15	2060	Invalid date
2	1	15	2060	Normal behaviour
3	2	15	2060	Normal behaviour
4	11	15	2060	Normal behaviour
5	12	15	2060	Normal behaviour
6	13	15	2060	Invalid date
7	6	0	2060	Invalid date
8	6	1	2060	Normal behaviour

9	6	2	2060	Normal behaviour
10	6	30	2060	Normal behaviour
11	6	31	2060	Invalid input
12	6	32	2060	Invalid date
13	6	15	2008	Invalid date
14	6	15	2009	Normal behaviour
15	6	15	2010	Normal behaviour
16	6	15	2098	Normal behaviour
17	6	15	2099	Normal behaviour
18	6	15	3000	Invalid date
19	6	15	2060	Normal behaviour

Test Cases Using Worst Case Testing

	Month	Day	Year
Min value	1	1	2009
Min^+ value	2	2	2010
Max value	12	31	2099
Max^- value	11	30	2098
Nominal value	6	15	2060

Using these values, test cases can be designed as shown below:

Test Case ID	Month	Day	Year	Expected Output
1	1	1	2009	Normal behaviour
2	1	1	2010	Normal behaviour
3	1	1	2060	Normal behaviour
4	1	1	2098	Normal behaviour
5	1	1	2099	Normal behaviour
6	1	2	2009	Normal behaviour
7	1	2	2010	Normal behaviour
8	1	2	2060	Normal behaviour
9	1	2	2098	Normal behaviour
10	1	2	2099	Normal behaviour
11	1	15	2009	Normal behaviour
12	1	15	2010	Normal behaviour
13	1	15	2060	Normal behaviour
14	1	15	2098	Normal behaviour
15	1	15	2099	Normal behaviour
16	1	30	2009	Normal behaviour
17	1	30	2010	Normal behaviour
18	1	30	2060	Normal behaviour
19	1	30	2098	Normal behaviour
20	1	30	2099	Normal behaviour
21	1	31	2009	Normal behaviour
22	1	31	2010	Normal behaviour

23	1	31	2060	Normal behaviour
24	1	31	2098	Normal behaviour
25	1	31	2099	Normal behaviour
26	2	1	2009	Normal behaviour
27	2	1	2010	Normal behaviour
28	2	1	2060	Normal behaviour
29	2	1	2098	Normal behaviour
30	2	1	2099	Normal behaviour
31	2	2	2009	Normal behaviour
32	2	2	2010	Normal behaviour
33	2	2	2060	Normal behaviour
34	2	2	2098	Normal behaviour
35	2	2	2099	Normal behaviour
36	2	15	2009	Normal behaviour
37	2	15	2010	Normal behaviour
38	2	15	2060	Normal behaviour
39	2	15	2098	Normal behaviour
40	2	15	2099	Normal behaviour
41	2	30	2009	Invalid date
42	2	30	2010	Invalid date
43	2	30	2060	Invalid date
44	2	30	2098	Invalid date
45	2	30	2099	Invalid date
46	2	31	2009	Invalid date
47	2	31	2010	Invalid date
48	2	31	2060	Invalid date
49	2	31	2098	Invalid date
50	2	31	2099	Invalid date
51	6	1	2009	Normal behaviour
52	6	1	2010	Normal behaviour
53	6	1	2060	Normal behaviour
54	6	1	2098	Normal behaviour
55	6	1	2099	Normal behaviour
56	6	2	2009	Normal behaviour
57	6	2	2010	Normal behaviour
58	6	2	2060	Normal behaviour
59	6	2	2098	Normal behaviour
60	6	2	2099	Normal behaviour
61	6	15	2009	Normal behaviour
62	6	15	2010	Normal behaviour
63	6	15	2060	Normal behaviour
64	6	15	2098	Normal behaviour
65	6	15	2099	Normal behaviour

66	6	30	2009	Normal behaviour
67	6	30	2010	Normal behaviour
68	6	30	2060	Normal behaviour
69	6	30	2098	Normal behaviour
70	6	30	2099	Normal behaviour
71	6	31	2009	Invalid date
72	6	31	2010	Invalid date
73	6	31	2060	Invalid date
74	6	31	2098	Invalid date
75	6	31	2099	Invalid date
76	11	1	2009	Normal behaviour
77	11	1	2010	Normal behaviour
78	11	1	2060	Normal behaviour
79	11	1	2098	Normal behaviour
80	11	1	2099	Normal behaviour
81	11	2	2009	Normal behaviour
82	11	2	2010	Normal behaviour
83	11	2	2060	Normal behaviour
84	11	2	2098	Normal behaviour
85	11	2	2099	Normal behaviour
86	11	15	2009	Normal behaviour
87	11	15	2010	Normal behaviour
88	11	15	2060	Normal behaviour
89	11	15	2098	Normal behaviour
90	11	15	2099	Normal behaviour
91	11	30	2009	Normal behaviour
92	11	30	2010	Normal behaviour
93	11	30	2060	Normal behaviour
94	11	30	2098	Normal behaviour
95	11	30	2099	Normal behaviour
96	11	31	2009	Invalid date
97	11	31	2010	Invalid date
98	11	31	2060	Invalid date
99	11	31	2098	Invalid date
100	11	31	2099	Invalid date
101	12	1	2009	Normal behaviour
102	12	1	2010	Normal behaviour
103	12	1	2060	Normal behaviour
104	12	1	2098	Normal behaviour
105	12	1	2099	Normal behaviour
106	12	2	2009	Normal behaviour
107	12	2	2010	Normal behaviour

108	12	2	2060	Normal behaviour
109	12	2	2098	Normal behaviour
110	12	2	2099	Normal behaviour
111	12	15	2009	Normal behaviour
112	12	15	2010	Normal behaviour
113	12	15	2060	Normal behaviour
114	12	15	2098	Normal behaviour
115	12	15	2099	Normal behaviour
116	12	30	2009	Normal behaviour
117	12	30	2010	Normal behaviour
118	12	30	2060	Normal behaviour
119	12	30	2098	Normal behaviour
120	12	30	2099	Normal behaviour
121	12	31	2009	Normal behaviour
122	12	31	2010	Normal behaviour
123	12	31	2060	Normal behaviour
124	12	31	2098	Normal behaviour
125	12	31	2099	Normal behaviour

Generate Tax Payable Details

This function calculates the tax payable by the person in the following format:

Taxable income The function will calculate this by taking the difference of gross total income in the function Accept Income Details and aggregate deductible amount in the function Accept Savings & Donation Details.

Tax on taxable income The function will calculate this using the appropriate slab of user as given above.

The following conditions are there for calculating the appropriate tax slab of a person:

- Is Sex Male?
- Age > 65?
- Income: up to 1.5 lacs
- Income: up to 1.8 lacs
- Income : 1.5 lacs to 3 lacs
- Income : 1.8 lacs to 3 lacs
- Income : 3 lacs to 5 lacs
- Income : above 5 lacs
- Income : up to 2.25 lacs
- Income : 2.25 lacs to 3 lacs

The following outputs will be there:

- No tax
- 10%
- 20%
- 30%

The test cases for these conditions and outputs can be designed using a decision table, as given below:

		R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
Condition Stub	C1: Is Sex Male?	T	T	T	T	F	F	F	F	I	I	I	I
	C2: Age > 60?	F	F	T	F	F	F	F	F	T	T	T	T
	C3: Income: up to 1.5 lacs	T	F	F	F	I	F	F	F	I	F	F	F
	C4: Income: up to 1.8 lacs	F	I	F	F	T	F	F	F	I	F	F	F
	C5: Income : 1.5 lacs to 3 lacs	F	T	F	F	F	I	F	F	I	I	F	F
	C6: Income : 1.8 lacs to 3 lacs	F	F	F	F	F	T	F	F	I	I	F	F
	C7: Income : 3 lacs to 5 lacs	F	F	T	F	F	F	T	F	F	F	T	F
	C8: Income : above 5 lacs	F	F	F	T	F	F	F	T	F	F	F	T
	C9: Income : up to 2.25 lacs	F	I	F	F	F	I	F	I	T	F	F	F
	C10: Income : 2.25 lacs to 3 lacs	F	I	F	F	F	I	F	F	F	T	F	F
Action Stub	A1: No Tax	X				X				X			
	A2: 10%		X				X				X		
	A3: 20%			X				X				X	
	A4: 30%				X				X				X

Surcharge The function will calculate the surcharge as: If tax on taxable income exceeds Rs 10 lacs, a 10% surcharge is charged.

Education cess The function will calculate the education cess as a 3% of tax on taxable income.

Tax payable The function will sum up the tax on taxable income, surcharge, and education cess.

Relief under section 89 User will enter the amount, if applicable.

Tax payable after relief (if applicable) The function will deduct relief amount from tax payable.

Total Tax deducted Displayed from the function Accept Tax Deduction Details.

Tax payable/refundable The function will find the difference of tax payable and the total tax deducted. If difference is positive, then this amount is the net tax to be paid by the person, otherwise the amount is due on the government to be refunded.

Step

8

White-Box Testing on Units/Modules of Income Tax Calculator

Here, we will discuss how the modules of Income Tax Calculator is tested using white-box testing techniques. The modules of TaxCalculator.h (refer CD) have been reproduced here for understanding their functionality and coding details. The reader is advised to refer to the full code for implementation details of the application and execute these test cases on the running executable application provided in the CD.

Basis Path Testing on income_details_non_sal()

Source Code of income_details_non_sal()

```
float income_details_non_sal()
     {
1'   char source[20]="abc";
2'   float amount,total =0;
3'   int flag1=1,flag2=1,i;
4'   char income_ch='y';

1    while((income_ch=='y')||(income_ch=='Y'))
2        {
3        while(flag1==1)
4            {
5            printf("\nEnter SOURCE\t:");
6            gets(source);
7            for(i=0;i<strlen(source);i++)
8                {
9                 if(((toascii(source[i]) >= 65) && (toascii(source[i])
                  <= 122)) || (toascii(source[i]) == 32))
10                       {
11                       flag1=0;
12                       }
```

```
13                  else
14                  {
15                  printf("\nSource can contain only charcter Error at
                          position number %d",i);
16                  flag1=1;
17                  break;
18                  }
19              }//end for
20              if((strlen(source)<3)||(strlen(source)>20))
21              {
22              printf("\nSource can contain a max of 20 characters");
23              flag1=1;
24              }
25  }
26  while(flag2==1)
27  {
28  printf("\nEnter Amount\t:");
29  scanf("%f",&amount);
30  if(amount>0)
31          {
32          flag2=0;
33          }
34      else
35          {
36          printf("\nAmount cannot be less than or equal to 0");
37          flag2=1;
38          }
39  }
40  printf("\n\nPress any key to proceed");
41  getch();
42  clrscr();
43  patt("INCOME Details");
44  printf("\nSOURCE\t:%s",source);
45  printf("\nAMOUNT\t:%f",amount);
46  total=total+amount;
47  printf("\nDo you want to enter more(y/n)\t:");
48  income_ch=getche();
49  flag1=1;
50  flag2=1;
51  }
52  printf("\nTotal\t\t:%f",total);
53  return(total);
54  }
```

DD Graph for Module income_details_non_sal()

Cyclomatic Complexity of income_details_non_sal()

1. $V(G) = e - n + 2P$
   $= 29 - 23 + 2$
   $= 8$
2. $V(G) =$ Number of predicate nodes + 1
   $= 7 + 1$
   $= 8$
3. $V(G) =$ No. of regions
   $= 8$

Independent Paths of income_details_non_sal()

Since the cyclomatic complexity of the graph is 8, there will be 8 independent paths in the graph, as shown below:

1. $N_1 N_2 N_{23}$
2. $N_1 N_2\ N_3\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
3. $N_1 N_2\ N_3\ N_4\ N_5\ N_6\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
4. $N_1 N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_8\ N_{10}\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
5. $N_1 N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_8\ N_9\ N_{11}\ N_6\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
6. $N_1 N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_8\ N_9\ N_{11}\ N_6\ N_{12}\ N_{13}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
7. $N_1 N_2\ N_3\ N_4\ N_{15}\ N_{16}\ N_{17}\ N_{19}\ N_{20}\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
8. $N_1 N_2\ N_3\ N_4\ N_{15}\ N_{16}\ N_{17}\ N_{18}\ N_{20}\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$

Test Case Design on income_details_non_sal() from the list of Independent Paths

Test Case ID	Inputs		Expected Output	Independent paths covered by Test Case
	Source	Amount		
1	Agriculture	400000	Source Agriculture Amount 400000 Do you want to enter more(y/n) Y	1, 2, 3, 5, 8
	Others	100000	Source Others Amount 100000 Do you want to enter more(y/n) N	
			Total 500000	
2	1234		Source can contain only character.	1, 2, 3, 4
3	Agriculture and others		Source can contain a max of 20 characters.	1, 2, 3, 6
4	Agriculture	0	Amount cannot be less than or equal to 0	1, 2, 3, 5, 7

Data Flow Testing on income_details_non_sal()

Definition Nodes and Usage Nodes

Variable	Defined At	Used At
source	1', 6	7, 9, 20, 44
amount	29	30, 45, 46
total	2', 46	46, 52
flag1	3', 11, 16, 23, 49	3
flag2	3', 32, 37, 50	26
i	7	7, 9, 15
income_ch	4', 48	1

du and dc paths

Variable	du Path(beg-end)	dc?
source	1'–7	No
	1'–9	No
	1'–20	No
	1'–44	No
	6–7	Yes
	6–9	Yes
	6–20	Yes
	6–44	Yes
amount	29–30	Yes
	29–45	Yes
	29–46	Yes
total	2'–46	Yes
	2'–52	No
	46–46	Yes
	46–52	Yes
flag1	3'–3	Yes
	11–3	No
	16-3	No
	23–3	Yes
	49–3	Yes

flag2	3'–26	Yes
	32–26	Yes
	37–26	Yes
	50–26	Yes
i	7–7	Yes
	7–9	Yes
	7–15	Yes
income_ch	4'–1	Yes
	48–1	Yes

Basis Path Testing on income_details_sal()

Source code of income_details_sal()

```
double income_details_sal()
{
1'   float t_d, d1, d2, sal1, sal2, sal3, t_sal, sal_all, sal_all_tot=0, ei,
     t_ei=0, net_t_sal=0, bal;
2'   char sal_ch='y',ei_ch='y';
3'   int f1=1,f2=1,f3=1,f4=1;
4'   double gross;

1    while(f2==1)
2         {
3         printf("\n1.\tGROSS SALARY\t:");
4         printf("\n\ta) Salary as per the provisions contained in the sec-
              tion 17(1)\t:");
5         scanf("%f",&sal1);
6         printf("\n\tb) Value of the perquisites under section 17(2)\n(As
              per form number 12BA , wherever applicable )\t:");
7         scanf("%f",&sal2);
8         printf("\n\tc) Profits in lieu of salary under section 17(3)\n(As
              per form number 12BA , wherever applicable )\t:");
9         scanf("%f",&sal3);
10        if((sal1<0)||(sal2<0)||(sal3<0))
11             {
12             f2=1;
13             }
14        else
15             {
16             f2=0;
```

```
17          }
18      }
19      t_sal=sal1+sal2+sal3;
20      printf("\n\td)\tTotal\n\t\t\t\t:%f",t_sal);
21      sal_all_tot=0;
22      while((sal_ch=='y')||(sal_ch=='Y'))
23      {
24      while(f1==1)
25          {
26          printf("\n2.\tAllowance to the extent exempt under section
            10\t:");
27          scanf("%f",&sal_all);
28          if(sal_all<0)
29           {
30           printf("\nEnter correct value");
31           }
32          else
33           {
34           f1=0;
35           }
36          }
37      sal_all_tot=sal_all_tot+sal_all;
38      printf("\nEnter more?(Y/N)\t:");
39      sal_ch=getche();
40      if((sal_ch=='y')||(sal_ch=='Y')||(sal_ch=='n')||(sal_ch=='N'))
41           {
42           f1=0;
43           }
44      else
45           {
46           printf("\nPlease enter y or n");
47           }
48      }
49      printf("\nTotal allowance\t\t:%f",sal_all_tot);
50      bal=t_sal-sal_all_tot;
51      printf("\nBalance\t:%f",bal);
52      while(f3==1)
53      {
54      printf("\n3.\tDeductions\t:");
55      printf("\n\tEntertainment allowance(EA)\t:");
56      scanf("%f",&d1);
```

```
57        printf("\tTax on employment (TE)\t:");
58        scanf("%f",&d2);
59        if((d1<0)||(d2<0))
60             {
61             f3=1;
62             }
63        else
64             {
65             f3=0;
66             }
67        }
68        t_d=d1+d2;
69        printf("\nTotal deductions\t:%f",t_d);
70        net_t_sal=bal-t_d;
71        printf("\n4.\tINCOME CHARGABLE UNDER THE HEAD SALARIES'\t:%f",net_t_sal);
72        while((ei_ch=='y')||(ei_ch=='Y'))
73        {
74        while(f4==1)
75        {
76        printf("\n5.\tAny other income reported by the Employee\t:");
77        printf("\n\t\tEnter Income\t:");
78        scanf("%f",&ei);
79        if(ei<0)
80             {
81             f4=1;
82             }
83        else
84             {
85             f4=0;
86             }
87        }
88        t_ei=t_ei+ei;
89        printf("\n\t\tEnter more?(Y/N)\t:");
90        ei_ch=getche();
91        }
92        gross=net_t_sal+t_ei;
93        printf("\n6.Gross Total Income:\t%f",gross);
94        return(gross);
95   }
```

DD Graph for Module income_details_sal()

Cyclomatic Complexity of income_details_sal()

1. $V(G) = e - n + 2P$
   $= 52 - 42 + 2$
   $= 12$
2. $V(G) =$ Number of predicate nodes + 1
   $= 11 + 1$
   $= 12$
3. $V(G) =$ No. of Regions
   $= 12$

Independent Paths of income_details_sal()

Since the cyclomatic complexity of the graph is 12, there will be 12 independent paths in the graph as shown below:

1. $N_1\ N_2\ N_8\ N_9\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{26}\ N_{32}\ N_{41}\ N_{42}$
2. $N_1\ N_2\ N_3\ N_4\ N_6\ N_7\ N_2\ N_8\ N_9\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{26}\ N_{32}\ N_{41}\ N_{42}$
3. $N_1\ N_2\ N_3\ N_4\ N_5\ N_7\ N_2\ N_8\ N_9\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{26}\ N_{32}\ N_{41}\ N_{42}$
4. $N_1\ N_2\ N_8\ N_9\ N_{10}\ N_{11}\ N_{12}\ N_{18}\ N_{19}\ N_{22}\ N_{21}\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{26}\ N_{32}\ N_{41}\ N_{42}$
5. $N_1\ N_2\ N_8\ N_9\ N_{10}\ N_{11}\ N_{12}\ N_{13}\ N_{14}\ N_{16}\ N_{17}\ N_{12}\ N_{18}\ N_{19}\ N_{22}\ N_{21}\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{26}\ N_{32}\ N_{41}\ N_{42}$
6. $N_1\ N_2\ N_8\ N_9\ N_{10}\ N_{11}\ N_{12}\ N_{13}\ N_{14}\ N_{15}\ N_{17}\ N_{12}\ N_{18}\ N_{19}\ N_{22}\ N_{21}\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{26}\ N_{32}\ N_{41}\ N_{42}$
7. $N_1\ N_2\ N_8\ N_9\ N_{10}\ N_{11}\ N_{12}\ N_{18}\ N_{19}\ N_{20}\ N_{21}\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{26}\ N_{32}\ N_{41}\ N_{42}$
8. $N_1\ N_2\ N_8\ N_9\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{27}\ N_{28}\ N_{30}\ N_{31}\ N_{25}\ N_{26}\ N_{32}\ N_{41}\ N_{42}$
9. $N_1\ N_2\ N_8\ N_9\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{27}\ N_{28}\ N_{29}\ N_{31}\ N_{25}\ N_{26}\ N_{32}\ N_{41}\ N_{42}$
10. $N_1\ N_2\ N_8\ N_9\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{27}\ N_{28}\ N_{30}\ N_{31}\ N_{25}\ N_{26}\ N_{32}\ N_{33}\ N_{34}\ N_{40}\ N_{32}\ N_{41}\ N_{42}$
11. $N_1\ N_2\ N_8\ N_9\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{27}\ N_{28}\ N_{30}\ N_{31}\ N_{25}\ N_{26}\ N_{32}\ N_{33}\ N_{34}\ N_{35}\ N_{36}\ N_{38}\ N_{39}\ N_{34}\ N_{40}\ N_{32}\ N_{41}\ N_{42}$
12. $N_1\ N_2\ N_8\ N_9\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{27}\ N_{28}\ N_{30}\ N_{31}\ N_{25}\ N_{26}\ N_{32}\ N_{33}\ N_{34}\ N_{35}\ N_{36}\ N_{37}\ N_{39}\ N_{34}\ N_{40}\ N_{32}\ N_{41}\ N_{42}$

Test Case Design on income_details_sal() from the list of Independent Paths

Test Case ID	Inputs								Expected Output	Independent path covered by Test Case
	Sal1	Sal2	Sal3	Sal_all	Enter more?	EA	TE	Other Income		
1	2000	4000	9000						Total 15000	1, 2, 5, 7, 8, 10, 11
				2000	Y					
				1000	n				Total Allowance 3000 Balance 12000	
						200	300		Total deductions 500 Income under Head Salaries 11500	
								12000	Enter more? Y	
								12000	Enter more? N Gross Total Income: 35500	
2	2000	–300	500							2, 3
3	2000	4000	9000						Total 15000	2, 6
				–300					Enter correct value	
4	2000	4000	9000						Total 15000	2, 4, 5, 7
				1000	t				Please enter y or n	
5	2000	4000	9000						Total 15000	2, 5, 7, 9
				1000	n				Total Allowance 1000 Balance 14000	
						–100	200			
6	2000	4000	9000						Total 15000	2, 5, 7, 8, 10, 12
				2000	Y					
				1000	n				Total Allowance 3000 Balance 12000	
						200	300		Total deductions 500 Income under Head Salaries 11500	
								–1000		

DATA FLOW TESTING ON income_details_sal()

Definition nodes and Usage nodes

Variable	Defined At	Used At
t_d	68	69, 70
d1	56	59, 68
d2	58	59, 68
sal1	5	10, 19
sal2	7	10, 19
sal3	9	10, 19
t_sal	19	20
sal_all	27	28, 37
sal_all_tot	21, 37	37, 49, 50
ei	78	79, 88
t_ei	1', 88	88, 92
net_t_sal	1', 70	71, 92
bal	50	70
sal_ch	2', 39	40
ei_ch	2', 90	72
f1	3', 34, 42	24
f2	3', 12, 16	1
f3	3', 61, 65	52
f4	3', 81, 85	74
gross	92	93

du and dc paths

Variable	du Path(beg-end)	dc?
t_d	68–69	Yes
	68–70	Yes
d1	56–59	Yes
	56–68	Yes
d2	58–59	Yes
	58–68	Yes
sal1	5–10	Yes
	5–19	Yes
sal2	7–10	Yes
	7–19	Yes
sal3	9–10	Yes
	9–19	Yes
t_sal	19–20	Yes
sal_all	27–28	Yes
	27–37	Yes
sal_all_tot	21–37	No

	21–49	No
	21–50	No
	37–37	No
	37–49	Yes
	37–50	Yes
ei	78–79	Yes
	78–88	Yes
t_ei	1'–88	No
	1'–92	No
	88–88	No
	88–92	Yes
net_t_sal	1'–71	No
	1'–92	No
	70–71	Yes
	70–92	Yes
bal	50–70	Yes
sal_ch	2'–40	No
	39–40	Yes
ei_ch	2'–72	Yes
	90–72	Yes
f1	3'–24	Yes
	34–24	Yes
	42–24	Yes
f2	3'–1	Yes
	12–1	Yes
	16–1	Yes
f3	3'–52	Yes
	61–52	Yes
	65–52	Yes
f4	3'–74	Yes
	81–74	Yes
gross	85–74	Yes
	92–93	Yes

Basis Path testing on module savings()

Source Code of module savings()

```
float savings()
     {
1    char saving_type[20];
2    float amount,total=0;
3    int flag1=1,flag2=1,i;
4    char sav_ch='y';
5    while((sav_ch=='y')||(sav_ch=='Y'))
6    {
```

```
7    while(flag1==1)
8        {
9        printf("\nEnter Saving type\t:");
10       gets(saving_type);
11       for(i=0;i<strlen(saving_type);i++)
12           {
13            if(((toascii(saving_type[i])>= 65) && (toascii(saving_type[i])
              <= 122)) || (toascii(saving_type[i]) == 32))
14            {
15            flag1=0;
16            }
17           else
18            {
19           printf("\nSaving type can contain only charcter Error at posi-
              tion number %d",i);
20            flag1=1;
21            break;
22            }
23           }
24           if((strlen(saving_type)<3)||(strlen(saving_type)>20))
25           {
26           printf("\nPlease enter between 3 to 20 characters ");
27           flag1=1;
28           }
29       }
30       while(flag2==1)
31       {
32       printf("\nEnter Amount\t:");
33       scanf("%f",&amount);
34       if(amount>0)
35            {
36            flag2=0;
37            }
38           else
39            {
40            printf("\nAmount cannot be less than or equal to 0");
41            flag2=1;
42            }
43       }
44       printf("\n\nPress any key to proceed");
45       getch();
46       clrscr();
47       patt("SAVING Details");
48       printf("\nSAVING TYPE\t:%s",saving_type);
49       printf("\nAMOUNT\t\t\t:%f",amount);
50       total=total+amount;
51       printf("\nDo you want to enter more(y for yes)\t:");
52       sav_ch=getche();
53       flag1=1;
```

```
54        flag2=1;
55        }
56        printf("\nTotal\t\t:%f",total);
57        return(total);
58    }
```

DD Graph for Module savings()

Cyclomatic Complexity of savings()

1. $V(G) = e - n + 2P$
   $= 29 - 23 + 2$
   $= 8$
2. $V(G) =$ Number of predicate nodes + 1
   $= 7 + 1$
   $= 8$
3. $V(G) =$ No. of regions
   $= 8$

Independent Paths of savings()

Since the cyclomatic complexity of the graph is 8, there will be 8 independent paths in the graph, as shown below:

1. $N_1\ N_2\ N_{23}$
2. $N_1\ N_2\ N_3\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
3. $N_1\ N_2\ N_3\ N_4\ N_5\ N_6\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
4. $N_1\ N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_8\ N_{10}\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
5. $N_1\ N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_8\ N_9\ N_{11}\ N_6\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
6. $N_1\ N_2\ N_3\ N_4\ N_5\ N_6\ N_{12}\ N_{13}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
7. $N_1\ N_2\ N_3\ N_4\ N_{15}\ N_{16}\ N_{17}\ N_{19}\ N_{20}\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
8. $N_1\ N_2\ N_3\ N_4\ N_{15}\ N_{16}\ N_{17}\ N_{18}\ N_{20}\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$

Test Case Design on savings() from the list of Independent Paths

Test Case ID	Inputs			Expected Output	Independent path covered by Test Case
	Saving Type	Amount	Enter more?		
1	NSC	5000		Saving type NSC Amount 5000	1, 2, 3, 5, 8
			y		
	PPF	1200		Saving type PPF Amount 1200	
			n	Total 6200	
2	123			Saving type can contain only character	2, 3, 4

3	PF			Please enter between 3 to 20 characters	2, 3, 6
4	PPF	0		Amount cannot be less than or equal to 0.	2, 3, 7

Data Flow Testing on module savings()

Definition and Usage nodes

Variable	Defined At	Used At
saving_type	10	11, 13, 24, 48
amount	33	34, 49, 50
total	2, 50	50, 56, 57
flag1	3, 15, 20, 27, 53	7
flag2	3, 36, 41, 54	30
i	11	11, 13, 19
sav_ch	4, 52	5

du and dc paths

Variable	du Path(beg-end)	dc?
saving_type	10–11	Yes
	10–13	Yes
	10–24	Yes
	10–48	Yes
amount	33–34	Yes
	33–49	Yes
	33–50	Yes
total	2–50	No
	50–50	No
	50–56	Yes
	50–57	Yes
flag1	3–7	Yes
	15–7	No
	20–7	No
	27–7	Yes
	53–7	Yes
flag2	3–30	Yes
	36–30	Yes
	41–30	Yes
	54–30	Yes

i	11–11	No
	11–13	Yes
	11–19	Yes
sav_ch	4–5	Yes
	52–5	Yes

Basis Path Testing on module tax_ded()

Source code of tax_ded()

```
float  tax_ded()
     {
1    char challan[20];
2    int flag=1,flag1=1,flag2=1,flag3=1,flag4=1,i,dd,mm,yyyy,choice='y';
3    float amount_tax,tot_amount_tax=0,amount_tds,tot;

4    while((choice=='y')||(choice=='Y'))
5    {   flag=1,flag1=1,flag2=1,flag3=1,flag4=1;
6        while(flag4==1)
7        {
8        printf("\nEnter amount of Tax deposited\t:");
9        scanf("%f",&amount_tax);
10           if(amount_tax<0)
11            {
12            printf("\nAMOUNT CANNOT BE NEGATIVE");
13            flag4=1;
14            }
15           else
16            {
17            flag4=0;
18            }
19       }
20       while(flag3==1)
21       {
22           flag3=0;
23           printf("\nEnter DATE:\t:");
24           printf("\n\tEnter day(dd)\t:");
25           scanf("%d",&dd);
26            if((dd>31)||(dd<=0))
27            {
28            flag3=1;
29            printf("\nWrong date");
30            }
```

```
31          printf("\n\tEnter month\t:");
32          scanf("%d",&mm);
33          if((mm<=0)||(mm>12)||((mm==2)&&(dd>29)))
34           {
35           printf("\nWrong date/month");
36           flag3=1;
37           }
38          printf("\n\tEnter year\t:");
39          scanf("%d",&yyyy);
40          if((yyyy<2009)||(yyyy>2099))
41           {
42           printf("\nPlease enter a year between 2009 and 2099");
43           flag3=1;
44           }
45      }
46      while(flag1==1)
47      {
48          printf("\nEnter Challan Number\t:");
49          gets(challan);
50          for(i=0;i<strlen(challan);i++)
51          {
52          if(((toascii(challan[i]) >= 65) && (toascii(challan[i]) <=
             122)) || (toascii(challan[i]) == 32) || ((toascii(challan[i])
             >= 48) && (toascii(challan[i]) <= 57)))
53           {
54           flag1=0;
55           }
56          else
57           {
58           printf("\nChallan Number can contain only charcter Error at
                position number %d",i);
59           flag1=1;
60           break;
61           }
62          }//end of for
63          if((strlen(challan)<5)||(strlen(challan)>20))
64          {
65          printf("\nPlease enter correct Challan Number having 5 to 20
                characters");
66          flag1=1;
67          }
68      }
69      printf("\nAMOUNT OF TAX DEPOSITED\t:%f",amount_tax);
```

```
70      printf("\nCHALLAN NUMBER\t\t:%s",challan);
71      printf("\nDATE\t\t\t:%d/%d/%d",dd,mm,yyyy);
72      flag=1;
73      while(flag)
74      {
75      printf("\nDo you want to enter more ?(y/n)");
76      choice=getche();
77      if((choice=='y')||(choice=='Y')||(choice=='N')||(choice=='n'))
78          {
79          flag=0;
80          }
81          else
82          {
83          printf("\nPlease Enter y or n");
84          flag=1;
85          }
86      }//end of inner while
87  tot_amount_tax=tot_amount_tax+amount_tax;
88  }
89      while(flag2==1)
90      {
91      printf("\nEnter amount of TDS \t:");
92      scanf("%f",&amount_tds);
93      if(amount_tds<0)
94          {
95          printf("\nTDS AMOUNT CANNOT BE NEGATIVE");
96          flag2=1;
97          }
98      else
99          {
100         flag2=0;
101         }
102     }
103     printf("\n\nPress any key to proceed");
104     getch();
105     clrscr();
106     patt("TAX DEDUCTION");
107     printf("\nAMOUNT OF TAX DEPOSITED\t:%f",tot_amount_tax);
108     printf("\nAMOUNT OF TDS\t:%f",amount_tds);
109     tot=tot_amount_tax+amount_tds;
110     printf("\nTotal\t\t:%f",tot);
111     return(tot);
112     }
```

DD Graph for Module tax_ded()

Cyclomatic Complexity of tax_ded()

1. $V(G) = e - n + 2P$
   $= 61 - 47 + 2$
   $= 16$
2. $V(G) =$ Number of predicate nodes + 1
   $= 15 + 1$
   $= 16$
3. $V(G) =$ No. of regions
   $= 16$

Independent Paths of tax_ded()

Since the cyclomatic complexity of the graph is 16, there will be 16 independent paths in the graph as shown below:

1. $N_1\ N_2\ N_{41}\ N_{47}$
2. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
3. $N_1\ N_2\ N_3\ N_4\ N_5\ N_6\ N_8\ N_9\ N_4\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
4. $N_1\ N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_9\ N_4\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
5. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{11}\ N_{12}\ N_{14}\ N_{15}\ N_{17}\ N_{18}\ N_{20}\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
6. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{11}\ N_{12}\ N_{13}\ N_{14}\ N_{15}\ N_{17}\ N_{18}\ N_{20}\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
7. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{11}\ N_{12}\ N_{13}\ N_{14}\ N_{15}\ N_{16}\ N_{17}\ N_{18}\ N_{20}\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
8. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{11}\ N_{12}\ N_{13}\ N_{14}\ N_{15}\ N_{16}\ N_{17}\ N_{18}\ N_{19}\ N_{20}\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
9. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{21}\ N_{22}\ N_{23}\ N_{29}\ N_{31}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
10. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{21}\ N_{22}\ N_{23}\ N_{29}\ N_{30}\ N_{31}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
11. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{21}\ N_{22}\ N_{23}\ N_{24}\ N_{25}\ N_{27}\ N_{28}\ N_{23}\ N_{29}\ N_{31}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
12. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{21}\ N_{22}\ N_{23}\ N_{24}\ N_{25}\ N_{26}\ N_{28}\ N_{23}\ N_{29}\ N_{31}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
13. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{34}\ N_{35}\ N_{37}\ N_{38}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
14. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{34}\ N_{35}\ N_{36}\ N_{38}\ N_{33}\ N_{34}\ N_{35}\ N_{37}\ N_{38}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
15. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{42}\ N_{43}\ N_{45}\ N_{46}\ N_{41}\ N_{47}$
16. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{42}\ N_{43}\ N_{44}\ N_{46}\ N_{41}\ N_{47}$

Test Case Design on tax_ded() from the list of Independent Paths

Test Case ID	Inputs							Expected Output	Independent path covered by Test Case
	Tax amount	day	month	year	Challan number	Enter more?	TDS		
1	5000	12	12	2009	Dfert56			AMOUNT OF TAX DEPOS-ITED 5000 CHALLAN NUMBER Dfert56 DATE 12 12 2009	1, 2, 3, 5, 9, 12, 14, 15
						Y			
	2000	13	12	2009	Hgyt65			AMOUNT OF TAX DEPOS-ITED 2000 CHALLAN NUMBER Hgyt65 DATE 13 12 2009	
						n			
							1200	AMOUNT OF TAX DEPOS-ITED 7000 AMOUNT OF TDS 1200 Total 8200	
2	–200							AMOUNT CANNOT BE NEGATIVE	2, 3, 4
3	3000	32/0/-1						Wrong date	2, 3, 6
4	3000	31	-1/0/13					Wrong date/ month	2, 3, 7
5	3000	29	2					Wrong date/ month	2, 3, 7
6	3000	31	5	2008				Please enter a year between 2009 and 2099	2, 3, 8

7	4000	31	5	2009	!@#$			Challan number can contain only characters	2, 3, 5, 11
8	4000	31	5	2009	rty			Please enter correct challan number having 5 to 20 characters	2, 3, 5, 10
9	4000	31	5	2009	Rty456	n	–234	TDS AMOUNT CANNOT BE NEGATIVE	2, 3, 5, 12, 14, 16
10	4000	31	5	2009	Rty456	t		Please enter y or n	2, 3, 5, 13

Data Flow Testing on module tax_ded()

Definition and Usage nodes

Variable	Defined At	Used At
challan	49	50, 52, 63, 70
flag	2, 5, 72, 79, 84	73
flag1	2, 5, 54, 59, 66,	46
flag2	2, 5, 96, 100	89
flag3	2, 5, 22, 28, 36, 43	20
flag4	2, 5, 13, 17	6
i	50	50, 52, 58
dd	25	26, 33
mm	32	33
yyyy	39	40
choice	2, 76	77
amount_tax	9	10, 69, 87
tot_amount_tax	3, 87	87, 107, 109
amount_tds	92	93, 108, 109
tot	109	110, 111

du and dc paths

Variable	du Path(beg-end)	dc?
challan	49–50	Yes
	49–52	Yes
	49–63	Yes
	49–70	Yes
flag	2–73	No
	5–73	No
	72–73	Yes
	79–73	Yes
	84–73	Yes
flag1	2–46	No
	5–46	Yes
	54–46	No
	59–46	No
	66–46	Yes
flag2	2–89	No
	5–89	Yes
	96–89	Yes
	100–89	Yes
flag3	2–20	No
	5–20	Yes
	22–20	No
	28–20	No
	36–20	No
	43–20	Yes
flag4	2–6	No
	5–6	Yes
	13–6	Yes
	17–6	Yes
i	50–50	No
	50–52	Yes
	50–58	Yes
dd	25–26	Yes
	25–33	Yes
mm	32–33	Yes
yyyy	39–40	Yes
choice	2–77	No

	76–77	Yes
amount_tax	9–10	Yes
	9–69	Yes
	9–87	Yes
tot_amount_tax	3–87	No
	3–107	No
	3–109	No
	87–87	No
	87–107	Yes
	87–109	Yes
amount_tds	92–93	Yes
	92–108	Yes
	92–109	Yes
tot	109–110	Yes
	109–111	Yes

Basis Path Testing on module male_tax()

Source code of male_tax()

```
float male_tax(float taxable)
        {
        float t;

1       if(taxable<=150000)
2               {
3               return 0;
4               }
5       else if((taxable>150000)&&(taxable<=300000))
6               {
7               t=10*taxable/100;
8               }
9       else if((taxable>300000)&&(taxable<=500000))
10              {
11              t=20*taxable/100;
12              }
13      else
14              {
15              t=30*taxable/100;
16              }
17      return t;
18    }
```

DD Graph for Module male_tax()

Cyclomatic Complexity of male_tax()

1. $V(G) = e - n + 2P$
   $= 8 - 6 + 2$
   $= 4$
2. $V(G) =$ Number of predicate nodes + 1
   $= 3 + 1$
   $= 4$
3. $V(G) =$ No. of regions
   $= 4$

Independent Paths of male_tax()

Since the cyclomatic complexity of the graph is 4, there will be 4 independent paths in the graph, as shown below:

1. $N_1\ N_2\ N_6$
2. $N_1\ N_3\ N_6$
3. $N_1\ N_4\ N_6$
4. $N_1\ N_5\ N_6$

Test Case Design on male_tax() from the list of Independent Paths

Test Case ID	Input Taxable income	Expected Output	Independent path covered by Test Case
1	147000	return 0	1
2	123000	return 12300	2
3	340000	return 68000	3
4	620000	return 186000	4

Basis Path Testing of module fem_tax()

Source code of fem_tax()

```
float fem_tax(float taxable)
        {
        float t;

1       if(taxable<=180000)
2               {
3               return 0;
4               }
5       else if((taxable>180000)&&(taxable<=300000))
6               {
7               t=10*taxable/100;
8               }
9       else if((taxable>300000)&&(taxable<=500000))
10              {
11              t=20*taxable/100;
12              }
13      else
14              {
15              t=30*taxable/100;
16              }
17      return t;
18    }
```

DD Graph for Module fem_tax()

Cyclomatic Complexity of fem_tax()

1. $V(G) = e - n + 2P$

$$= 8 - 6 + 2$$

$$= 4$$

2. V(G) = Number of predicate nodes + 1
   = 3 + 1
   = 4
3. V(G) = No. of regions
   = 4

Independent Paths of fem_tax()

Since the cyclomatic complexity of the graph is 4, there will be 4 independent paths in the graph, as shown below:

1. $N_1\ N_2\ N_6$
2. $N_1\ N_3\ N_6$
3. $N_1\ N_4\ N_6$
4. $N_1\ N_5\ N_6$

Test Case Design on fem_tax() from the list of Independent Paths

Test Case ID	Input Taxable income	Expected Output	Independent path covered by Test Case
1	180000	return 0	1
2	270000	return 27000	2
3	430000	return 86000	3
4	620000	return 186000	4

Basis Path Testing of module senior_tax()

Source Code of senior_tax()

```
float senior_tax(float taxable)
        {
        float t;

1       if(taxable<=225000)
2               {
3       return 0;
4               }
5       else if((taxable>225000)&&(taxable<=300000))
6               {
7               t=10*taxable/100;
8               }
9       else if((taxable>300000)&&(taxable<=500000))
10              {
```

```
11              t=20*taxable/100;
12              }
13      else
14              {
15              t=30*taxable/100;
16              }
17      return t;
18   }
```

DD Graph for Module senior_tax()

Cyclomatic Complexity of senior_tax()

1. $V(G) = e - n + 2P$
   $= 8 - 6 + 2$
   $= 4$
2. $V(G) =$ Number of predicate nodes + 1
   $= 3 + 1$
   $= 4$
3. $V(G) =$ No. of regions
   $= 4$

Independent Paths of senior_tax()

Since the cyclomatic complexity of the graph is 4, there will be 4 independent paths in the graph as shown below:

1. $N_1\ N_2\ N_6$
2. $N_1\ N_3\ N_6$
3. $N_1\ N_4\ N_6$
4. $N_1\ N_5\ N_6$

Test case design on senior_tax from the list of independent paths

Test Case ID	Input Taxable income	Expected Output	Independent path covered by Test Case
1	225000	return 0	1
2	228000	return 22800	2
3	430000	return 86000	3
4	620000	return 186000	4

Basis Path Testing of Module gtd()

Source code of gtd()

```
void gtd(float gross_income,float net_deductions,float tax_ded)
        {
1       float taxable_income,tax,edu_cess,surcharge=0,difference,total_tax;
2       taxable_income=gross_income-net_deductions;

3       printf("\nTaxable Income\t\t:%f",taxable_income);
4       if((s=='m')||(s=='M'))
5               {
6               tax=male_tax(taxable_income);
7               printf("\nTax\t\t\t:%f",tax);
8               }
9       else if((s=='f')||(s=='F'))
10              {
11              tax=fem_tax(taxable_income);
12              printf("\n\Tax\t\t\t:%f",tax);
13              }
14      else if((2009-yr1)>65)
15              {
16              tax=senior_tax(taxable_income);
17              printf("\nTax\t\t\t:%f",tax);
18              }
19      if(tax>1000000)
20              {
21              surcharge=10*tax/100;
22              printf("\nSurcharge\t\t:%f",surcharge);
23              }
24      edu_cess=3*tax/100;
25      printf("\nEducational cess\t:%f",edu_cess);
26      total_tax=tax+surcharge+edu_cess;
27      printf("\nTax payable\t\t:%f",total_tax);
28      printf("\nTotal tax deducted\t:%f",tax_ded);
```

```
29      difference=total_tax-tax_ded;
30      if(difference>0)
31              {
32              printf("\nTax payable\t\t:%f",difference);
33              }
34      else if(difference<0)
35              {
36              printf("\nRerfundable\t%f",-1*difference);
37              }
38      else
39              {
40              printf("\nNo tax");
41              }
42   }
```

DD Graph for Module gtd()

Cyclomatic Complexity of gtd()

1. $V(G) = e - n + 2P$
   $= 17 - 13 + 2$
   $= 6$
2. $V(G) =$ Number of predicate nodes + 1
   $= 5 + 1$
   $= 6$
3. $V(G) =$ No. of regions
   $= 6$

Independent Paths of gtd()

Since the cyclomatic complexity of the graph is 6, there will be 6 independent paths in the graph as shown below:

1. $N_1\ N_2\ N_3\ N_6\ N_8\ N_9\ N_{10}\ N_{13}$
2. $N_1\ N_2\ N_4\ N_6\ N_8\ N_9\ N_{10}\ N_{13}$
3. $N_1\ N_2\ N_5\ N_6\ N_8\ N_9\ N_{10}\ N_{13}$
4. $N_1\ N_2\ N_3\ N_6\ N_7\ N_8\ N_9\ N_{10}\ N_{13}$
5. $N_1\ N_2\ N_3\ N_6\ N_7\ N_8\ N_9\ N_{11}\ N_{13}$
6. $N_1\ N_2\ N_3\ N_6\ N_7\ N_8\ N_9\ N_{12}\ N_{13}$

Test case design on gtd() from the list of independent paths

Test Case ID	Inputs					Expected Output	Independent path covered by Test Case
	Gross_ income	Net_ deductions	Tax_ded	Status	Age		
1	320000	2000	1000	male	34	Tax 30000 Educational cess 900 Tax payable 30900 Total Tax deducted 1000 Tax payable 29900	1
2	340000	2000	1000	female	34	Tax 32000 Educational cess 960 Tax payable 32960 Total Tax deducted 1000 Tax payable 31960	2
3	420000	2000	1000	male	67	Tax 84000 Educational cess 2520 Tax payable 86520 Total Tax deducted 1000 Tax payable 85520	1, 3

4	12000000	2000	1000	male	34	Tax 3629400 Surcharge 362940 Educational cess 108882 Tax payable 4101222 Total Tax deducted 1000 Tax payable 4100222	1, 4
5	320000	2000	32000	male	34	Tax 30000 Educational cess 900 Tax payable 30900 Total Tax deducted 32000 Refundable 1100	1, 5
6	320000	2000	30900	male	34	Tax 30000 Educational cess 900 Tax payable 30900 Total Tax deducted 30900 No tax	1, 6

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

References

1. Edward Kit, Software Testing in the Real World, Pearson Education Pte. Ltd., Delhi, 2004
2. G.J. Myers, The Art of Software Testing, John Wiley & Sons, 1979
3. Pankaj Jalote, CMM in Practice, Pearson Education Pte. Ltd., Delhi, 2004
4. William E. Perry, Effective Methods for Software Testing, John Wiley & Sons (Asia) Pte. Ltd., Second Edition, 2003
5. Boehm B., Software Engineering Economics, Prentice Hall, 1981
6. Wallace, D.R. and R.U. Fujii, "Software Verification and Validation: An Overview", IEEE Software, May 1989
7. Roger S. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill International Edition, Fifth Edition, 2001
8. K.K. Aggarwal, Yogesh Singh, Software Engineering, New Age International (P) Ltd., 2001
9. Boris Beizer, Software Testing Techniques, Dreamtech Press, Second Edition, 2004
10. J.M. Voas, "PIE: A Dynamic Failure based technique", IEEE Trans. On Software Engg., Vol. 18, No. 8, Aug. 1992, pp717-727
11. James A. Whittalcer, "What is Software Testing? And why is it so hard?", IEEE Software, Ja/Feb 2000, pp 70-79
12. Elfriede Dustin, Effective Software Testing: 50 specific ways to improve your testing, Pearson Education Pte. Ltd., Delhi, 2004
13. Renu Rajani, Pradeep Oak, Software Testing: Effective Methods, Tools and Techniques, Tata McGraw-Hill Publishing, 2004
14. Pankaj jalote, An Integrated approach to Software Engineering, Narosa Publishing House, Second Edition, 2003
15. Manna Z., and Waldinger R., "The Logic of Complete Programming", IEE Trans. On Software Engg., 4:199-229. 1978
16. Richard Fairley, Software Engineering Concepts, Tata McGraw-Hill, 2003
17. Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli, Fundamentals of Software Engineering, Prentice Hall of India, 1999
18. IEEE Recommended Practice for Software requirements Specifications IEEE Std 830-1998 (Revision of IEEE-Std 830-1993)
19. IEEE Recommended Practice for Software Design Descriptions, IEEE Std 1016-1998
20. Software Testing: A Craftsman's Approach, CRC Press, Second Edition, 2002
21. ANSI/IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, IEEE Computer Society Press
22. Louise Tamres, Introducing Software Testing, Pearson Education Pte. Ltd., Delhi, 2004
23. Kenneth H. Rosen, Discrete Mathematics and its applications, Tata McGraw-Hill, Forth Edition
24. T. McCabe, "A Complexity Measure", IEEE Trans. On Software Engg., 2(4), 1976, pp308-320
25. Fabrizio Riguzzi, "A Survey of Software Metrics", DEIS Tech. Report No. DEIS-LIA-96-010, July 1996
26. Basili V.R., Selby R.W., "Comparing the effectiveness of Software Testing Strategies", IEEE Trans. On Software Engg., SE-13(12), 1278-96, 1987
27. Gilb T., Graham D., Software Inspection, Addison Wesley, 1993

28. Fagan M.E., "Advances in Software Inspections", IEEE Trans. On Software Engg., SE-12(7), 744-51
29. Ian Sommerville, Software Engineering, Pearson Education Pte. Ltd., Delhi, Sixth Edition, 2004
30. N.H. Petschenik, "Practical Properties in Software Testing", IEEE Software 2(5):18-23
31. G.W. Jones, Software Engineering, Jon Wiley and Sons, 1990
32. Norman E. Fenton, Shari Lawrence Pfleeger, Software Metrics: A Rigorous and Practical Approach, Brooks/Cole, Second Edition
33. David Garmus, David Herron, Function Point Analysis: Measurement Practices for Successful software projects, Addison Wesley Information Tech. series
34. Paul Goodman, Practical Implementation of Software Metrics, McGraw-Hill, 1993
35. Ilene Burnstein, Practical Software Testing: A Process oriented approach, Springer-Verlog, New York, 2003
36. Grady Booch, Object –Oriented Analysis and Design with Applications, Pearson Education, Second Edition, 2000
37. Wendy Boggs, Michael Boggs, Mastering UML with Rational Rose, BPB Publications, 2002
38. M.J. Harrold, John D. McGregor and Kevin J. Fitzpatrick, "Incremental Testing of Object-Oriented Class Structures", Proceedings of 14th International Conference on Software Engg., 1992
39. P.B. Crisby, Quality is Free: The Art of making Quality Certain, McGraw-Hill, New York, 1979
40. J.M. Juran and F.M. Gryna, Quality Planning and Analysis: Product Development through use, McGraw-Hill, NewYork, 1970
41. W. Humphrey, A Discipline for Software Engineering, Addison Wesley, MA, 1995
42. Stephan H. Kan, Metrics and Model in Software Quality Engineering:, Pearson Education, 2003
43. M.E. Fagon, "Design and Code Inspections to reduce errors in program development" IBM System Journal, (3):182-211, 1976
44. I.S.G. of Software Engg. Terminology, IEEE Standards Collection, IEEE Std 610.12-1990, Sep. 1990
45. Akira K. Onoma, Wei-Tek Tsai, Mustafa Poonawal and Hiroshi Suganuma, "Regression Testing in an Industrial Environment", Communication of the ACM, 41(5):81-86, 1998
46. Hiralal Agrawal, Joseph R. Horgan et al., "Incremental Regression Testing", IEEE Conference on Software maintenance, Montreal Canal, 1993
47. H. Lenng and L. White, "Insights into Regression Testing", Proceedings of the conference on Software maintenance, 1989
48. Cem Kaner, Jack Falk, Hung Quoc Nguyen, Testing Computer Software, Second Edition, Wiley India, 2007
49. B. Beizer, Software Testing and Quality Assurance, Von Nostrand Reinhold, New York, 1984
50. Rothermel and M.J. Harrold, "A Framework for evaluating regression test selection techniques", Proceedings of 16th International Conference on Software Engineering, IEEE Computer Soceity Press, May 1994
51. Rothermel and M.J. Harrold, "Analyzing regression test selection techniques", IEEE Transactions on Software Engineering, 22(8):529-551, Aug. 1996
52. Rothermel and M.J. Harrold, "Empirical studies of a safe regression test selection technique", IEEE Transactions on Software Engineering, 24(6):401-419, June 1998
53. K. Fischer, F. Raji, A. Chruscicki, " A Methodology for retesting modified software", Proceedings of National Tele. conference , B-6-3, pp1-6, 1981
54. M. Harrold and M. Soffa, "An incremental approach to unit testing during maintenance" Proceedings of the conference on Software maintenance, pp 362-367, 1988
55. G. Rothermel and M. Harrold, "A safe, efficient regression test selection technique", ACM Transactions on Software Engg. And Methodology, 6(2):173-210, 1997
56. IEEE/ANSI (1983), IEEE Standard for Software Test Documentation, (Reaff. 1991), IEEE Std 829-1983
57. D.N. Card, R.L. Glass, Measuring Software Design Quality, Prentice-Hall, 1990
58. G. Rothermel, R.H. Untch, Chengyun Chu, M.J. Harrold, "Prioritizing test cases for Re-

gression Testing", IEEE Trans. On Software Engg., Vol. 27, Issue 10, Oct 2001

59. Paulk, M, Curtis B., Chrissis M.B. and Weber, C., *Capability Maturity Model for Software*, Software Engineering Institute, Carnegie Mellon University, 1993
60. Burnstein I., Suwanassart T., and Carlson C., "Developing a Testing Maturity Model. Part 1", *Crosstalk, Journal of Defense Software Engineering*, 9, no. 8, 21–24, 1996, also available on http://www.stsc.hill.af.mil/-crosstalk/1996/aug/developi.asp
61. Burnstein I., Suwanassart T., and CarlsonC., "Developing a Testing Maturity Model. Part 2", *Crosstalk, Journal of DefenseSoftware Engineering*, 9, no. 9, 19–26, 1996, also available on http://www.stsc.hill.af.mil/-crosstalk/1996/sep/developi.asp
62. Gelperin D. and Hetzel B., "The Growth of Software Testing", Communications of the ACM, 31, no. 6, 687–695, 1998
63. Paulk, M., C. Weber, B. Curtis, and M. Chrissis, *The Capability Maturity Model Guideline for Improving the Software Process*, Addison-Wesley, Reading, Mass., 1995
64. Thayer, R., "Software Engineering Project Management, A Top Down View," *IEEE Tutorial, Software Engineering Project Management,* R. Thayer, ed., IEEE Computer Society Press, Los Alamitos, Calif., 1990, pp. 15-53
65. Jef Jacobs, Jan van Moll, and Tom Stokes, "The Process of Test Process Improvement", XOOTIC MAGAZINE, Nov. 2000
66. S.L. Pfleeger, Software Engineering:Theory and Practice, Pearson Education, Second Edition, 2003
67. Firesmith, Donald G. "Testing Object-Oriented Software," published in Proceedings of TOOLS, 19 March 1993
68. David et al. "Developing an Object-Oriented Software Testing Environment", Communications of the ACM, Vol. 38, No. 10, pp. 75-87, Oct. 1995
69. Booch, G., *Object Oriented Design With Applications*, Benjamin Cummings, 1991,
70. Harrison, R., *Abstract Data Types in Modula-2*, J. Wiley & Sons, 1989
71. C. D. Turner and D. J. Robson, "Guidance for the Testing of Object-Oriented Programs", Technical Report No: TR 2/93 Computer Science Division School of Engineering and Computer Science (SECS) University of Durham Durham, England16 April, 1993
72. T. J. Ostrand and M. J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests," Comm. ACM, vol. 31, no. 6, pp. 676–686, 1988.
73. P. Wegner and S. B. Zdonik, "Inheritance as an incremental modification mechanism or what like is and isn't like," Proceedings of ECOOP'88, pp. 55-77, Springer-Verlag, 1988
74. Arthur H.W., Thomas J. McCabe, "Structured Testing: A Testing Methodology using the Cyclomatic Complexity metric", NIST special publication 500-235, sep. 1996
75. P. Jorgensen and C. Erickson, Object-oriented integration testing, *Communications of the ACM*, 37(9):30-38, Septenber 1994
76. J. Overbeck, Integration Testing for Object Oriented Software, PhD thesis, Vienna University of Technology, 1994.
77. S. Kirani, Specification and Verification of Object-Oriented Programs, PhD thesis, University of Minnesota, Minneapolis, Minnesota, December 1994
78. P. Jorgensen and C. Erickson, "Object-oriented integration testing", Communicationsof the ACM, 37(9):30.38, September 1994
79. D. Gelperin and B. Hetzel, "The Growth of Software Testing", Communications of the ACM, Volume 31 Issue 6, June 1988, pp. 687-695
80. Lu Luo, "Software Testing Techniques: Technology Maturation and Research Strategies", Institute for Software Research International, Class Report for 17-939A
81. Software Testing 3.0:The Continuing Evolution of Software Testing, LogiGear Corporation white paper January 2008
82. Bret Pettichord, Schools of Software Testing, March 2007, www.pettichord.com
83. James Bach, Explaining testing anybody, www.satisfice.com
84. Edward Miller, "Introduction to software testing technology", In Tutorial: Software Testing & Validation Techniques, pages 4-16, IEEE Computer Society Press, 1981.
85. Cem Kaner, "Exploratory Testing", Quality Assurance Institute Worldwide Annual Software

Testing Conference, Orlando, FL, November 2006

86. l.H. Fenton, "Response to the SHARE software service task force report, IBM Corp., Kingston, NY, March 6, 1984
87. Parnas, D. L. and Weiss, D. M., "Active Design Reviews: Principles and Practices," Proceedings of the Eighth International Conference on Software Engineering, August, 1985
88. Johnson, P. M., "An Instrumented Approach to Improving Software Quality Through Formal Technical Review", Proceedings of the 16th International Conference on Software Engineering (ICSE-16), May 1994
89. Gilb, T. and Graham, D., "Software Inspection", Addison-Wesley, 1993
90. Macdonald, F. and Miller, J., "Modeling Software Inspection Methods for the Application of Tool Support", December, 1995
91. Martin, J. and Tsai, W. T., "N-Fold Inspection: A Requirements Analysis Technique", Communications of the ACM, V. 33, N. 2, February, 1990, pp. 225-232
92. Knight, J. C., and Myers, E. A., "An Improved Inspection Technique", Communications of the ACM, V. 36, N. 11, November, 1993, pp. 51-61
93. Porter, A. & Votta, L.G., "An experiment to assess different defect detection methods for software requirements inspections", In Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, Los Alamitos: IEEE Computer Society Press, 103-112, May 16-21, 1994
94. Basili, V.R., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Soerumgaard, S. & Zelkowitz M., "The empirical investigation of perspective-based reading", Empirical Software Engineering 1(2), 133-164, 1996
95. Laitenberger, O. & DeBaud, J.-M., "Perspective-based reading of code documents at Robert Bosch GmbH", Information and Software Technology 39(11), 781-791, 1997
96. Thelin, T., Runeson, P. & Regnell, B., "Usage-based reading - An experiment to guide reviewers with use cases" Journal of Information and Software Technology 43(15), 925-938, 2001
97. Thelin, T., Runeson, P. & Wohlin, C., "An experimental comparison of usage-based and checklist-based reading", IEEE Transactions on Software Engineering 29(8), 687-704, 2003
98. Thelin, T., Runeson, P., Wohlin, C., Olsson, T. & Andersson, C., "Evaluation of usage-based reading - Conclusions after three experiments", Empirical Software Engineering 9(1-2), 77-110, 2004
99. Dunsmore A., Roper M. & Wood M., "Systematic object-oriented inspection technique", In Proceedings of the 23rd International Conference on Software Engineering, Toronto, Canada, May 12-19. Washington: IEEE Computer Society, 123-144, 2001
100. Dunsmore A., Roper M. & Wood M., "Further investigations into the development and evaluation of reading techniques for object-oriented code inspection", In Proceedings of the 24th International Conference on Software Engineering, Orlando, Florida, May 19-25. New York: ACM Press, 47-57, 2002
101. Dunsmore, A., Roper, M. & Wood, M., "The development and evaluation of three diverse techniques for object-oriented code inspection", IEEE Transactions on Software Engineering 29(8), 677-686, 2003
102. Kelly, D. & Shepard, T., "Task-directed software inspection", Journal of Systems and Software 73(2), 361-368, 2004
103. Cheng, B. and Jeffrey, R., "Comparing Inspection Strategies for Software Requirements Specifications", Proceedings of the 1996 Australian Software Engineering Conference, pages 203-211, 1996
104. Basili, V., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sorumgard, S., and Zelkowitz, M., "The Empirical Investigation of Perspective-based Reading", Journal of Empirical Software Engineering, 2(1):133-164, 1996
105. Laitenberger, O., "Cost-Effective Detection of Software Defects with Perspective-based Inspection", PhD-Thesis, University of Kaiserslautern, ISBN 3-8167-5583-6, 2000
106. Edward Yourdon, Structured Walkthroughs, Yourdon Press, 4th edition, 1989
107. Laitenberger, O., "Studying the effects of code inspection and structural testing on software quality", Proceedings of ninth International Symposium o Software Relibility Engineering, 1998

108. W. Humphrey, A Discipline for Software Engineering, Addison-Wesley, Reading, MA, 1995
109. J. Rakos, Software Project Management for small to medium-sized projects, Prentice Hall, Englewood Cliffs, NJ, 1990
110. Y. Chernak, "Validating and improving test case effectiveness", IEEE Software, Vol. 16, No.1, pp. 81-86, 2001
111. Drs Eric P W M Van Veenendaal CISA, Ton Dekkers, "Test Point Analysis: A Method for Test Estimation", Published in Project Control for Software Quality, Shaker Publishing BV, Maastricht, Netherlands, 1999
112. S. R. Rakitin. Software Verification and Validation for Practitioners and Managers. Artech House Inc. Boston-London, 2001.
113. QAI Consulting Organization. Emphasizing Software Test Process Improvement, http://www.qaiindia.com/Resources_Art/journal_emphasizing.htm , September 2006.
114. M. L. Hutcheson. Software Testing Fundamentals: Methods and Metrics. John Willey & Sons, 2003
115. K. Iberle, S. Bartlett. Estimating Tester to Developer Ratios (or Not). Hewlett-Packard and STEP Technology, www.kiberle.com/pnsqc1/estimate.doc, November 2006
116. R. S. Pressman. Software Engineering – A Practitioner's Approach. Sixth Edition, McGraw Hill Education Asia, 2005.
117. R. D. Craig, S. P. Jaskiel. Systematic Software Testing. Artech House Publishers, Boston-London, 2002.
118. B. Berger. Evaluating Test Plans with Rubrics. International Conference on Software Testing Analysis and Review, 2004
119. P. Piwowarski, M. Ohba, J. Caruso. Coverage Measurement Experience During Function Test. International Business Machines Corporation. In IEEE Software Engineering Proceedings, 1993.
120. M. Marre´, A. Bertolino. Using Spanning Sets for Coverage Testing. In IEEE Transactions on Software Engineering, Volume 29, Number 11, November 2003.
121. Afzal, W., Torkar, R., "Incorporating Metrics in an Organizational Test Strategy", Proc of IEEE International Conference on Software Testing Verification and Validation Workshop, April 2008
122. David Garmus, David Herron, Function Point Analysis: Measurement Practices for Successful Software Projects, Addison Wesley, 2001
123. www.ifpug.org
124. R. S. Pressman. Software Engineering – A Practitioner's Approach. McGraw Hill Education Asia, 2005.
125. W. Dijkstra. Structured Programming. In J.N.Buxton and B.Randell (eds.), Software Engineering Techniques, Brussels, Belgium, NATO Science Committee, 1970
126. E. Miller. The Philosophy of Testing. In Program Testing Techniques, IEEE Computer Society Press, 1977
127. M. Shaw, "Abstraction techniques in Modern Programming Languages", IEEE Software, vol. 1(4), 1984
128. J.Musa, Software Reliability, New York, NY: McGraw-Hill, 1999
129. J.Musa, "Software-Reliability-Engineered Testing", IEEE Computer, vol. 29, pp. 61-68, Nov. 1996
130. H. Agarwal, J.R. Horgan, E.W. Krauser, S. London, "Incremental regression testing", IEEE International Conference on Software Maintenance, pp. 348-357, 1993
131. T. Gyimothy, A. Beszedes, I. Forgacs, "An efficient relevant slicing method for debugging", ACM/SIGSOFT Foundations of Software Engineering, pp. 303-321, 1999
132. B. Korel, J.Laski, "Algorithmic software fault localization", Annual Hawaii International conference on system sciences", pp. 246-252, 1991
133. S. Elbaum, A. Malishevsky, G. Rothermel, "Prioritizing test cases for regression testing", Proceedings of International symposium on software testing and analysis, pp. 102-112, Aug. 2000
134. G. Rothermel, R. Untch, C. Chu, M.J. Harrold, "Test case Prioritization: an empirical study", Proccedings of International conference on software maintenance, pp. 179-188, Aug. 1999
135. S. Elbaum, A Malishevsky, G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization", Proceedings of 23rd International conference on software engineering, Ontario, Canada, May 2001

136. H. Srikanth, L. Williams, J. Osborne, "Towards the Prioritization of system test cases" North Carolina State University TR-2005-44, 2005
137. R.A. DeMillo, R.J. Lipton, F.G. Sayward, "Hints on Test Data Selection: Help for the practicing programmer", Computer, 11(4):34-41, April 1978
138. R.G Hamlet, "Testing programs with the aid of a compiler", IEEE transactions on Software engineering, SE-3(4):279-290, July 1977
139. Dennis Jeffrey, Neelam Gupta, "Test Case Prioritization using Relevant Slices", Proceedings of 30th Annual International Computer software and applications conference (COMPSAC '06), vol. 1, pp. 411-420, 2006
140. Koomen, T. and Pol M., Test Process Improvement: A practical step-by step guide to structured testing, ACM Press, London, England, 1999
141. Sog04 TPI home pages, Sogeti Nederland B.V., 2004. http://www.sogeti.nl/index.html?/iospagina.cfm?uNr=150
142. TMa04 TMap home pages, TMap - Sogeti Nederland B.V., 2004, http://www.tmap.net
143. www.gerrardconsulting.com
144. Ericson, T., Subotic, A. and Ursing, S., "Towards a Test Improvement Model", Proceedingsof the Fourth European Conference on Software Testing, Analysis & Review, Amsterdam, December 2-6, 1996
145. Jari Andersin, "TPI – a model for Test Process Improvement", Seminar on Quality Models for Software Engineering, Department of Computer Science UNIVERSITY OF HELSINKI, Helsinki, 5th October 2004
146. Capers, Jones, Applied software measurement, McGraw-Hill, 1996
147. Ghulam Mustafa, Abad Ali Shah, et al, "A Strategy for testing of web based software", Information Technology Journal, 6(1):74-81, 2007
148. Dart, S., Containing the Web Crisis Using Configuration Management, Proc ICSE Workshop on WebEngineering, 1999. http://fistserv.macarthur.uws.edu.au/san/icse99-webe/
149. Murugesan, S. et al. "Web engineering: A New Discipline for Development of Web based systems" In Proceedings of the First ICSE Workshop on Web Engineering, Los Angeles (pp. 1-9), 1999
150. Deshpande, Y. et al., " Web engineering", Journal of Web Engineering, 1(1), 3-17, 2002
151. Deshpande, Y., Ginige, A., Murugesan, S., & Hansen, S., "Consolidating Web engineering as a discipline.", SEA Software, (April), 32-34, 2002
152. Deshpande, Y., & Hansen, S., "Web engineering: creating a discipline among discipline", IEEE Multimedia, (April - June), 82-87, 2001
153. Deshpande Y., Olsina, L., & Murugesan, S., "Web engineering.", Report on the Third ICSE Workshop on Web Engineering, ICSE2002, Orlando, FL, USA, 2002.
154. Ginige, A., & Murugesan, S., "Web engineering: An introduction", IEEE Multimedia, 8(1), 14-18, 2001
155. Ginige, A. & Murugesan, S., "The essence of Web engineering: Managing the diversity and complexity of Web application development", IEEE Multimedia, 8(2), 22-25, 2001
156. Ginige, A., & Murugesan, S., "Web engineering: A methodology for developing scalable, maintainable Web applications", Cutter IT Journal, 14(7), 24-35, 2001
157. HENNICKER R. and KOCH N., "A UML-based Methodology for Hypermedia Design", In Proceedings of the Unified Modeling Language Conference, UML´2000, Evans A. and Kent S., Eds. LNCS 1939, Springer Verlag, 410-424, 2000
158. KOCH N. , Software Engineering for Adaptive Hypermedia Applications, PhD. Thesis, Reihe Softwaretechnik 12, Uni-Druck Publishing Company, Munich, 2001
159. KOCH N., KRAUS A. and HENNICKER R., "The Authoring Process of the UML-based Web Engineering Approach", In First International Workshop on Web-Oriented Software Technology IWWOST'2001, Valencia, 2001
160. KRAUS A. and KOCH N. , "Generation of Web Applications from UML Models using an XML Publishing Framework", Proceeding of the Integrated Design and Process Technology Conference, IDPT'2002, Pasadena, 2001
161. Koch, Andreas Kraus, "The Expressive Power of UML-based Web Engineering", Ludwig-Maximilians-Universität München. Germany http://www.pst.informatik.uni-muenchen.de/personen/kochn/IWWOST02-koch-kraus.pdf

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

APPENDICES

Appendix

A

Answers to Multiple Choice Questions

Chapter 1

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

Chapter 2

1. (b)	2. (a)	3. (d)	4. (d)	5. (c)
6. (a)	7. (c)	8. (d)	9. (b)	10. (a)
11. (b)	12. (a)	13. (c)	14. (b)	15. (b)

Chapter 3

1. (c)	2. (c)	3. (a)	4. (b)	5. (c)
6. (d)	7. (b)	8. (d)	9. (d)	10. (d)
11. (a) and (c)	12. (c)	13. (b)		

Chapter 4

1. (b)	2. (a)	3. (d)	4. (b)	5. (c)
6. (b)	7. (a)	8. (c)	9. (d)	10. (c)
11. (a)				

Chapter 5

1. (b)	2. (b)	3. (a)	4. (b)	5. (c)
6. (d)	7. (c)	8. (c)	9. (b)	10. (c)
11. (b)	12. (a)	13. (b)	14. (a)	15. (d)

Chapter 6

1. (a)	2. (b)	3. (b)	4. (d)	5. (a)
6. (b)	7. (d)	8. (d)	9. (b)	10. (a)
11. (a) and (c)	12. (b)	13. (c)	14. (b)	15. (d)
16. (a)	17. (c)	18. (c) and (d)	19. (d)	20. (d)

Chapter 7

1. (b)	2. (a)	3. (c)	4. (c)	5. (d)
6. (a)	7. (c)	8. (a)	9. (a)	10. (b)
11. (d)	12. (c)	13. (a)	14. (a) and (c)	15. (b)

16. (b)	17. (c)	18. (c)	19. (d)	20. (d)
21. (b)	22. (a)	23. (c)	24. (c)	

Chapter 8

1. (b)	2. (a)	3. (b)	4. (c)	5. (b)
6. (a)	7. (b)	8. (a) & (b)	9. (c)	10. (a)
11. (b)	12. (d)	13. (c)	14. (a)	15. (b)
16. (d)				

Chapter 9

1. (a), (c) and (d)	2. (b)	3. (c)	4. (a) and (b)	5. (a)
6. (c)	7. (b)	8. (a)	9. (b)	10. (c)
11. (d)	12. (a)	13. (d)	14. (a) and (b)	

Chapter 10

1. (a)	2. (a)	3. (a)	4. (b)	5. (c)
6. (b)	7. (a)			

Chapter 11

1. (c)	2. (b)	3. (a)	4. (b)	5. (c)
6. (a)	7. (a)	8. (b)	9. (c)	10. (d)
11. (a)	12. (b)			

Chapter 12

1. (a)	2. (b)	3. (a)	4. (b)	5. (a)
6. (b)	7. (a)	8. (c)	9. (c)	

Chapter 13

1. (d)	2. (b)	3. (a)	4. (b)	5. (a)
6. (c)	7. (a)	8. (d)	9. (a)	10. (b)
11. (c)	12. (a)	13. (b)		

Chapter 14

1. (b)	2. (d)	3. (a)	4. (c)	5. (d)
6. (b)	7. (b)	8. (b)	9. (d)	10. (c)
11. (a)				

Chapter 15

1. (a)	2. (c)	3. (a)	4. (d)	5. (c)
6. (b)	7. (c)	8. (a)	9. (b)	

Chapter 16

1. (b)	2. (a)	3. (c)	4. (a)	5. (d)
6. (c)	7. (a)	8. (b)	9. (b)	10. (a)
11. (c)	12. (d)	13. (d)	14. (d)	15. (c)
16. (a)	7. (b)	18. (a) and (d)	19. (b)	20. (b)

Chapter 17

1. (b)	2. (c)	3. (a)	4. (a)	5. (b)
6. (a)	7. (b)	8. (d)	9. (a)	10. (a)
11. (b)	12. (c)	13. (a)	14. (b)	15. (a) and (b)

Chapter 18

1. (b)	2. (b)	3. (a)	4. (b)	5. (d)
6. (c)	7. (a)			

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Appendix

B

Software Requirement Specification (SRS) Verification Checklist

S.No.		Y/N/NA	Remarks
1.	**Overview and System Functional Flow**		
	Is the overview of high-level system mentioned?		
	Is the system's functional flow clearly and completely described?		
	Has the high-level functionality of system depicted in a diagram like DFD level 0?		
	Do the high-level diagrams depict internal and external interfaces and data flows?		
	Are the software functions mentioned at a high-level with the viewpoint of operational system?		
	Are the users of the software including other systems interacting with it recognized?		
2.	**Functional Requirements**		
	Are the requirements traceable to customer?		
	Are the requirements defined separately?		
	Are the functionalities given proper name and unique number/ID?		
	Does each function fully define its purpose and scope?		
	Does each function mention required inputs and outputs?		
	Do the requirements specify any method for exception handling or handling any alternate functional logic flow?		
	Does each functional requirement mention the acceptance criteria, if any?		
	Have all the constraints (software / hardware) related to any functional requirement mentioned?		
	Are the functional requirements prioritized and given an identifier to indicate its importance?		
3.	**Non-functional (Performance, security, quality) requirements**		
	Is the acceptable response time for each functionality mentioned?		
	Is the number of maximum users on the system mentioned?		
	Is the number of transactions / online transactions per user per unit time mentioned?		

S.No.		Y/N/NA	Remarks
	Is the peak data flow volume mentioned?		
	Is the maximum size of the data transaction specified?		
	Is the upper and lower control limits (tolerance criteria) specified?		
	Is it mentioned what will be done by the system when it fails on exceeding its capacity?		
	Are all the physical and operational security and safety requirements, if any, mentioned?		
	Are quality requirements like reliability, portability, maintainability, etc. mentioned?		
	Are quality requirements quantified with the acceptable limit?		
	Are the non-functional requirements prioritized and given an identifier to indicate its importance?		
4.	**Interface Requirements**		
	Are all inputs to the system specified, including their source, accuracy, range of values, and parameters?		
	Are all screen formats specified?		
	Are all report formats specified?		
	Are all interface requirements between hardware, software, personnel, and procedures included?		
	Are all communication interfaces specified?		
5.	**Resource Requirements**		
	Are all hardware configuration requirements mentioned?		
	Are all software configuration requirements mentioned?		
	Are all memory requirements mentioned?		
	Are all software, hardware, memory requirements for interfaced systems mentioned?		
6.	**Correctness**		
	Is each requirement in scope for the project?		
	Is each requirement free from content and grammatical errors?		
	Does the requirement make the clear understanding?		
	Is the requirement realistic under mentioned constraints?		
7.	**Ambiguity**		
	Has every requirement only one interpretation?		
	Is each characteristic of the final product described using a single unique term?		
	Is any requirement conflict with other requirement?		
8.	**Consistency**		
	Is there any real world objects conflict such that as one specification recommends mouse for input and another recommends ball tab?		
	Is there any logical conflict between two specified actions?		

S.No.		Y/N/NA	Remarks
	Are there any requirements that describe two or more actions that conflict temporally?		
	Are there any requirements describing the same object that conflict with other requirements with respect to terminology?		
	Is there any conflict between the specified requirement and project goals?		
	Are the timing and memory limits compatible with hardware constraints?		
9.	**Completeness**		
	Is there any missing requirement?		
	Are requirements in terms of input, output, functional, non-functional, interfaces complete?		
	Is there any additional predicted capability for the system?		
	Are all requirements written at an appropriate level of detail?		
	Are assumptions and dependencies explicitly mentioned?		
	Are all sections, figures and tables labeled appropriately?		
	Are all figures and tables referenced appropriately?		
	Has every acronym, constant, variable, etc. been defined in the data dictionary?		
	Is a dictionary for all data elements provided?		
	Is the data dictionary complete in all manners?		
	Are all documents to be referenced listed?		
	Are all units of measure needed defined?		
10.	**Traceability**		
	Is each functional requirement traceable to system functionality?		
	Has each requirement a unique name or reference number in all the documents?		
11.	**Standards Compliance**		
	Is established standard or guidelines mentioned to be followed for the document?		
	Does the document format conform to the specified standard/guideline?		
	Are the standards and naming conventions followed according to the standards throughout the document?		

Y = Yes, N = No, NA = Not applicable

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Appendix

C

High Level Design (HLD) Verification Checklist

S.No.		Y/N/NA	Remarks
1.	**Data Design**		
	Have the sizes of data structure been estimated appropriately?		
	Is there any provision of overflow in a data structure?		
	Are the data formats consistent with the requirements?		
	Is the data usage consistent with its declaration?		
	Are the relationships correct among data objects in data dictionary?		
	Are databases and data warehouses consistent with requirements in SRS?		
	Are the data structure names meaningful?		
	Is the database organization and content specified?		
	Is the data model designed correct in all manners?		
	Are the database design rules listed?		
	Has the database model been described?		
	Are the logical data models of database, if any written clearly?		
2.	**Architectural Design**		
	Does the architecture consider every functional and non-functional requirements specified in SRS?		
	Is the architecture correct and unambiguous?		
	Is the functionality of sub-system traceable to the system functionality in SRS?		
	Is the hierarchy of subsystems correct?		
	Is dependency and interfaces between subsystems correctly designed?		
	Does the design consider all the constraints mentioned in SRS?		
	Is the coupling between subsystems low?		
	Is the cohesion between subsystems high?		
	Is the design flexible enough for future extensions to the program?		
	Does the architecture consider all exception handling features?		
	Are all input-outputs for a subsystem correct and complete?		
3.	**Interface Design**		
	Have the possible interfaces been identified?		

S.No.		Y/N/NA	Remarks
	Are the interfaces clear and well-defined?		
	Are the interfaces between modules according the to architecture design?		
	Is the required data passed at each interface?		
	Are the interfaces between software and other non-human producer and consumer of information correct?		
	Are the interfaces between the user and software system correct?		
	Are the interfaces consistent?		
	Are the response time for all the interfaces within required ranges?		
	Is the representation of Help in its desired manner?		
	Does the user return to the normal interaction from Help?		
	Do the error messages clarify the problem?		
	Does the message provide constructive advice for recovering from the error?		
	Is the mapping between every menu option and corresponding command for typed command interaction correct?		

Y = Yes, N = No, NA = Not applicable

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Appendix

D

Low Level design (LLD) Verification Checklist

S.No.	Criteria	Y/N/NA	Remarks
1.	Is HLD traceable to LLD?		
2.	Is high level and low level abstraction consistent?		
3.	Is every subsystem detailed enough to be operational?		
4.	Does the pseudo-code of every subsystem follow the guidelines and syntax rules of a program design language (PDL)?		
5.	Does the algorithm of every subsystem consistent with its defined functionality?		
6.	Does the design notation for LLD support the development of modular software?		
7.	Does the design notation for LLD support the means for interface specification?		
8.	Is the design notation able to represent local and global data?		
9.	Is the design notation able to modify the design representation easily?		
10.	Is the logic in each algorithm clear, correct, and complete?		
11.	Has the data been properly defined and initialized?		
12.	Are all the defined data used somewhere?		
13.	Is every detail of data available?		
14.	Is the data used consistently throughout the module and module interface?		
15.	Are any variables missing?		
16.	Have all interfaces been correctly considered and implemented in LLD?		
17.	Do the subsystems accept all data within the allowable range?		
18.	Is the data converted according to the correct format?		
19.	Has any consideration been given to the effects of round-off or truncation?		
20.	Are the indices valid in the subsystem pesudocode?		
21.	Is there any infinite loop in the subsystem pesudocode?		
22.	Does the design address arithmetic overflow and underflow?		
23.	Are the physical data models of database, if any, written clearly?		

Y = Yes, N = No, NA = Not applicable

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Appendix

E

General Software Design Document (SDD) Verification Checklist

S.No.	Criteria	Y/N/NA	Remarks
1.	Does the design allow addition of more parameters, for example, number of users?		
2.	Does the system allow addition of new data types?		
3.	What is the impact of adding new databases?		
4.	Can the design cope with technological changes?		
5.	Does every design decision documented in SDD have only a single interpretation?		
6.	Is the SDD consistent with higher-level documents?		
7.	Is the SDD consistent with documents and models at the same level?		
8.	Is the SDD internally consistent in that all design decisions that it contains are compatible?		
9.	Have all the design standards been followed?		
10.	Does SDD specify all significant design decisions?		
11.	Is the Architecture design supported with a diagram?		
12.	Have the actual deployment environment of the architecture in terms of tools, databases, and actual software used been mentioned?		

Y = Yes, N = No, NA = Not applicable

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Appendix

F

Generic Code Verification Checklist

S.No.		Y/N/NA	Remarks
1.	**Data Declaration**		
	Are all data considered in design declared in coding?		
	Are variable type and dimensions correctly declared?		
	Do all variables have proper type consistency?		
	Are declared variables initialized also? If so, are they correctly defined?		
	Are there any variables having same or similar name with any other library function or reserved keywords?		
	Is there any variable declared, but not used?		
	Are all the pointer variables declared correctly?		
	Are all the global variables declared correctly?		
	Is Boolean variable declared correctly (if supported)?		
	Is Boolean variable declared directly (if not supported by language directly)?		
	Are the variables of user-defined type declared correctly?		
2.	**Data Reference**		
	Is there any variable referenced, but not declared?		
	Is there any variable referenced, but not initialized?		
	Are the subscript values for array references within the specified limits?		
	Are the subscript values for array references of the type integer?		
	If a variable is passed as parameter in procedure, is it correctly and consistently referred?		
	Are the pointer variables correctly referenced?		
	Is Boolean variable referenced correctly?		
	Are the variables of user-defined data type referenced correctly?		
3.	**Interfaces**		
	Is the order of parameters same in prototype, calling module, and called module?		
	Do the types of parameters match in prototype, calling module, and called module?		

S.No.		Y/N/NA	Remarks
	Do the sizes of parameters match in prototype, calling module, and called module?		
	Does the return type of module match in prototype, calling module, and called module?		
	Does the called module after computation return the same type of data as expected?		
	Are the interfaces being implemented traceable to SDD and SRS?		
	Are the global variables referenced consistently?		
	Are the library files included correctly?		
	Are the called modules return to the calling module?		
4.	**Computation**		
	Are the data types of variables used in computation inconsistent?		
	Are the lengths of variables used in computation same?		
	Is the data type on both sides of computation same?		
	Is there any computation having divisor as zero?		
	Are the order of evaluation in computation correct?		
	Are the order of precedence of operators correct?		
	Are mixed-mode computations correct?		
	Is loop index variable used correctly?		
	Are bitwise operators used correctly in the computation?		
	Are logical operators used correctly in the computation?		
	If there is a computation such that on the right side, there is a module call and the result returned is stored in the variable on the left side, then is the data type of variable on the left side of computation same as of the data type of value returned by the module call?		
	Are special operators used correctly in the computation?		
5.	**Comparison**		
	Are comparison operators used correctly?		
	Do the types of variables, for which comparison is made, match?		
	Are logical expressions expressed correctly?		
	Are the operands of Boolean operator Boolean?		
6.	**Control Flow**		
	Is there any infinite loop?		
	Is there any loop which does not enter in its body?		
	Is there any loop which does not by-pass?		
	Is there any if-then-else/switch-case structure infinite?		
	Is there any if-then-else/switch-case structure which does not enter in its body?		
	Is there any if-then-else structure which does not by-pass?		
	Are the nested control statements used correctly?		
	How many nested controls are there?		

S.No.		Y/N/NA	Remarks
	Is the expression blank in a loop?		
	Is any required expression missing in a loop?		
	Is each loop controlled by a different index?		
	Is one loop completely embedded within the other in case of nested loops?		
	Is the expression missing in an if-then-else structure?		
	Is the type of expression in switch-case structure an integer?		
	Are all Case labels in switch-case structure unique?		
	Does the control come out from each Case label group?		
	Is there any 'goto' statement?		
	Does every switch-case contain a default label?		
7.	**Input/Output**		
	Are the input statements according to the specified format of language?		
	Are the specified input ranges correct in input statement format?		
	Are the output statements according to the specified format of language?		
	Are the specified output ranges to be displayed as output correct in output statement format?		
	Are the files declared correct?		
	Are the files opened before use?		
	Are the parameters in File Open specified correctly?		
	Are all opened files closed?		
	Are End of File conditions detected?		
	Are the texts displayed to user on the screen are meaningful and appropriate?		
	Is there any spelling or grammatical mistake in the texts displayed to the user on the screen?		
8.	**Modules/Sub-systems**		
	Is there any traceability between LLD and module code?		
	Are all modules coded which were considered in LLD?		
	Is there any module missing which was considered in LLD?		
	Does the module's code match with the pseudo-code in the LLD?		
	Is there any logical mistake in the module code which contradicts its functional specification?		
	Is there any module which has not been referenced?		
	Is the implementation of module according to the language constraints complex? If so, can it be restructured with modifications in pseudo-code?		
9.	**General**		
	Are there comments put appropriately in the code?		
	Are the comments put for complex logic of the code?		
	Are the comments meaningful and understandable?		
	Do all source code files mention the function number corresponding to which it has been developed?		

S.No.		Y/N/NA	Remarks
	Are there proper indentations in the code?		
	Is there an 'end' brace for every 'begin' in all structures?		
	Is the code readable enough?		
	Can the compact code be replaced with the alternate code which is more readable and understandable?		
	Does the code adhere to coding standards?		
	Are there any leftover stubs or test drivers in the code?		
	Is the code consistent in style?		
	Is there any unreachable code?		
	Is there any redundant code which can be replaced by the component library module?		
	Have all the declared storage been used to their full limit or is there any storage space wastage?		
	Is there any inconsistency in code and comments?		

Y = Yes, N = No, NA = Not applicable

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Acknowledgements

I am thankful to God for making things possible at the right time always.

Big projects are not developed overnight. Some ideas always incubate in our subconscious and take a definite shape gradually. However, these ideas are not developed by their own; they take shape as a result of constant learning and interaction between individuals and great personalities. I would like to acknowledge these personalities who have inspired me directly or indirectly to work on this book.

I would like to express my sincere gratitude to my school-time teacher, Sh. Girish Kumar who had given me the base for my upcoming life at that time. He has always been a role model for me. Next, I would like to thank my Guru, Pandit Priyadutt Shastri for realizing life with a totally different viewpoint and caused a turning point in my life. The principles that I learnt from these two persons will always be a moral support for me in any project.

The technical roots behind writing this book date back to the days when I was working in the Central Research Laboratory (Bharat Electronics Ltd., Ghaziabad), where I learnt many practical techniques of software testing. But for the critical learning support of Sh. K. Johri (Scientist at Central Research Laboratory, Ghaziabad) I would not have learnt this discipline. Those learning have helped me a lot in writing this book. He always used to say, 'Welcome the bugs; do not hide them.' While explaining the psychology of software testing, I kept his saying in mind

I would like to express my profound gratitude to Dr A. K. Sharma, Chairman (Computer Engg.), YMCA University of Science and Technology, Faridabad, who showed me the path of research and technical writing during the research work performed under him. I am extremely grateful to all my colleagues with whom I discussed many issues. Many thanks to my students, Sandeep Rana, Anita, Harsh, InduBala, and all others for their contribution towards completing this book.

I am indebted to my family for their love, encouragement, and support throughout my education. I am also thankful for all the support received from my parents-in-law.

I owe a lot to my dear wife, Anushree, who made many compromises to let me finish this book. I am thankful for her never-ending patience, unconditional moral support, and peaceful environment at home. Without her friendship and love, this book would not have been completed. I express my heartfelt gratitude to my dear daughter, Smiti, for her love and encouragement.

Finally, I extend my gratitude to the editorial staff at Oxford University Press for their support.

Thanks to all of you!

Naresh Chauhan

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

INDEX

Index Terms	Links		
A			
abstraction	447		
abstract superclass	460		
acceptance criteria	291		
acceptance testing	81	197	245
alpha testing	244		
beta testing	246		
types of	246		
acceptance test plan	283	291	
active design reviews	165		
activity diagrams	449		
actors	448	454	468
adjusted function point	313		
adjustment factor	313		
AFP	313		
alpha testing	75		
entry criteria	246		
exit criteria	246		
anomalies	246		
APFD	365		
cost-cognizant	367		
architectural design	238		
assessment	273		
authentication	238		
authorization	435		
automated script development	429		

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		

automated testing 436
behavioral view 447
environmental view 448
guidelines 429
implementation view 448
structural view 447
user view 447
automated testing tools 429 430

B

backlog management index 389
back tracking 508
based applications 475
baseline version 256
basis path testing 138
beta testing 247
entry criteria 248
exit criteria 247
guidelines for 247
beta-versions 90
black-box testing 40 90 226
BMI 389
bottom-up integration 224 225
boundary value analysis 201
boundary value checking 35
brainstorm 222
breadth first integration 37 222
breakpoint 507
conditional 507
internal 507
temporary 507
unconditional 507

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		

bug classification 90
based on criticality 91
based on SDLC 421

C

call graph 227 228 229
capability maturity model 392
capacity testing 494
capture/playback tools 433
card-sorting technique 486
cause-effect graph 126
checklist 205
abstraction driven reading 205
function-point based scenarios 204
perspective-based reading 204
scenario-based reading 205
task driven reading 204
usage-based reading 192
checklists 67
class diagrams 448
classes 446 450
class-level testing 452
class-responsibility-collaboration 449
client-server 476 477
client-server model 475
cluster-level testing 452 464
CMM structure 392
code comprehension 432
code coverage 326
code traceability 292
coding 195
cohesion 136

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links	

collaboration diagrams	454	
commercial testing tools	437	
configuration testing	243	490
complexity analysis tools	432	
component diagrams	449	
concatenated loops	141	
confidentiality	238	
configuration/compatibility testing	490	
configuration management	435	
connection matrix	326	
content testing	487	
control flow	142	195
conventional testing	450	
coupling	355	
coverage	352	353
coverage analysis tools	433	
coverage criteria	262	
coverage identification problem	343	
coverage measures	284	
CPM	141	
CRC	449	
CRC model index card	449	
cross-site scripting	492	
customer problem metrics	388	
customer satisfaction metrics	388	

D

database servers	476	
data complexity	331	
data design	74	

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		
data flow	164	165	326
testing	164		
anomalies	165		
dataflow techniques	263		
data flow testing	166	352	
dynamic	169		
static	167		
terminology used in	166		
data-member	446		
DD graph	139		
dead code	288		
debuggers	510		
debugging	503	505	509
debugging techniques	506		
memory dump	506		
watch points	506		
decision node	139		
decision table	120		
extended entry	120		
limited entry	120		
test case design using	120		
decision table-based testing	119		
decision-to-decision-graph	139		
decomposition-based integration	219	228	
top-down	222		
types of	221		
decomposition tree	220		
defect age	323	341	
defect-arrival pattern during testing	389		
defect density	343	389	
defect density metrics	388		
defect fix time to retest	342		

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		
defect rate	374		
defect-removal efficiency	323		
defect removal percentage	321		
defect spoilage	324		
defect-tracking	438		
defect trend analysis	342		
definition-clear path	189		
definition node	222		
definition-use path	256		
delta build	256		
delta version	222		
deployment diagram	449		
depth first	214		
depth first integration	292		
design traceability	255		
development effort estimation	328		
development ratio method	328		
development testing	217		
DFDs	508		
distributed computing	476		
DMAIC	399		
DRE	389		
driver module	228		
drivers	214	322	362
dynamic binding	451		
dynamic contents	488		
dynamic slice	336		
dynamic testing	431		

E

earned value tracking	344		
effectiveness of test cases	323		

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links

EIF 310
encapsulation 446
equivalence classes 108
equivalence class testing 107
equivalent classes 108
    identification of 108
    valid 109
error 196
error detection efficiency 129
error guessing 195
error-prone modules 196
error-types 195
estimation of test cases 327
execution slice 361
exit criteria 321
external interface files 310

F

failure 33
failure impact 359
fan-in 332
fan-out 332
fault/defect/bug 33
fault-exposing-potential 358
fault-revealing test cases 263
FEP 358
finite state machine 115
fix backlog 389
fix response time 390
fix responsiveness 390
flow graph 139 197

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		

formal inspection 197
active design reviews 197
FTArm 197
Gilb inspection 197 200
Humphrey's inspection 197 201
N-fold inspections 197
phased inspection 197
structured walkthrough 205 333
FP 205
FPA 114
FSM 115
FTArm 219
functional decomposition 67 289
functional design 89
functional testing 232 289
function coverage 232 333
function point analysis 333
function points 81 309 335

G

Gantt charts 284
glass-box testing 135
goal question metric 318
GQM 318
graph matrix 156 157

H

Halstead product metrics 308
high-level design 67
hitting set 354
HTTP 239

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		

human testing 188

I

IBM rational SQA robot 438
IFPUG 194 309
incremental testing 221 461
information flow complexity 332
inheritance 447 451 460
inheritance testing 459
inherited methods 460
inspection process 191
  benefits of 194
  bug prevention 194
  bug reduction 194
  checklists 205 255
  cost of 196
  effectiveness of 196
  variants of 197
inspections 190 381
installation testing 82
integration strategies 452
integration testing 57 156 391
integration test plan 287
integrity 75
inter-class testing 464
inter-cluster testing 464
interface design 67
interface testing 485 489
internal design 310
internal logical files 390
internal testing 430
intrusive tools 436

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links			
J				
junction node	139			
junior test engineers	276			
K				
KLOC	308			
KPAs	392	394	395	
L				
leaf nodes	219			
lines of code	308			
loadRunner	438			
load testing	240	241	438	494
load tests	327			
LOC	306			
private	307			
public metrics	307			
logic coverage	137			
condition coverage	136			
criteria	137			
decision/condition coverage	137			
decision or branch coverage	138			
multiple condition coverage	136			
statement coverage	161			
loop testing	67			
M				
maintenance testing	156			
master schedule	284			
master test plan	283			

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links	

maturity levels	393	413
mean-time to failure	388	
measurement program	318	
measurements	305	
memory testing tools	433	
MEP	229	230
method-level testing	452	
method-message path	464	
metrics	304	319
middleware	476	
Miller's theorem	142	
minimization techniques	263	
MM-path	230	231
module	76	
cohesion	76	
coupling	174	
MTTF	388	
multi-tiered applications	474	
mutants	174	
primary	177	
process	175	
secondary	174	

N

navigation testing	489	490
neighborhood integration	228	
nested loops	162	
network-testing tools	434	
N-fold inspection	202	
non-repudiation	238	
N-tier	476	

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links			

O

object	446	450		
object diagrams	449			
object model	445			
object-oriented modeling	447			
object-oriented technology	445			
object-oriented testing	450			

P

pair-wise integration	228			
path-based integration	229			
message	230			
module execution path	229			
sink node	229			
source node	229			
path testing	140			
independent path	140			
terminology	140			
unit testing	156			
use of	159			
percent delinquent fixes	390			
performance testing	238	239	240	489
	493			
PERT	284			
phased inspection	202			
polymorphism	447	451	452	
PORT	365			
prioritization	354	356		
based on requirements	364			
operational profiles	360			
regression test suite	356			

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links	
prioritization (*Cont.*)		
system test suite	356	
techniques	356	
prioritization scheme	355	
prioritized test suite	365	
probability of errors	331	
problems per user month	388	
procedure	335	
process improvement	195	
process maturity	396	
process quality	374	375
product quality	374	
program length	308	
program monitors	431	
program vocabulary	308	
program volume	309	
progressive testing	255	
project management	379	
project-staff ratio method	329	
PUM	388	

Q

quality assurance	377	
quality control	377	421
quality cost	375	
failure costs	376	
prevention costs	379	
quality factors	378	
quality management	381	

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links			

R

rate of fault detection	355			
reading techniques	203			
ad hoc method	203			
checklists	203			
realistic-size databases	240			
recovery testing	235	236		
recurrence ratio	342			
redundant test cases	353			
regions	139			
regression bugs	259			
regression number	258			
regression testability	257	258		
regression testing	256	257	258	505
	509	510		
objectives of	258			
problem	260			
techniques	260			
types	259			
regression test prioritization	265			
regression test problem	259			
regression test selection	263			
regression test suite	257			
regressive testing	255			
relevant slice	364			
relevant slices	360			
reliability	380			
requirement coverage matrix	290			
requirements	67			
gathering	91			
specification	120			

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		

requirement traceability	292	326	
reviews	359	360	
risk analysis	325		

S

safe techniques	263		
sandwich integration testing	225		
scalability testing	494		
SDLC phases	66	212	429
security requirements	236		
security testing	237	238	491
security vulnerabilities	237		
selective retest technique	261		
sequence diagram	449		
severity of impact	359		
simple loops	162		
simulated testing	430		
sink nodes	229		
six sigma	399		
size metrics	308		
slicing technique	361		
dynamic slice	362		
execution slice	361		
relevant slice	364		
smoke test	290	324	
SMP	376		
software crisis	480		
software maintenance	261		
software measurement	305	307	

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		
software metrics	304	306	
classification	306		
computed metrics	306		
definition of	306		
line of code	306		
objective	306		
primitive	306		
subjective metrics	306		
software quality	374	378	384
software quality metrics	387	388	
software testing	5		
as a process	22		
economics of	39		
effective	16		
evolution of	5		
exhaustive	16		
goals of	10		
methodology	51		
model for	15		
myths and facts	8		
psychology for	13		
schools of	23		
STLC	46		
strategy	52		
techniques	57		
terminology	33		
testing definitions	14		
software tools	430		
spanning set	326		
specialized environments	443		
spoilage	324		
SQA	378		

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		
SQA activities	381		
SQA audit	382		
SQA group	421		
SQA models	390		
SQA team	381		
SQL injection	492		
SRS	239		
staff productivity	321		
state-based testing	457	458	
state chart diagrams	449		
state graph	115		
state table	116		
state table-based testing	114	116	
static contents	488		
static program analysers	431		
static testing	189	381	
benefits	58		
objectives	189		
types of	189		
static testing tools	431		
static test points	135	338	
STQM	397		
stress testing	241	494	
structural complexity	57		
structured walkthrough	322		
stubs	215	220	288
subclass	460		
superclass	459	460	
suspension criteria for testing	255		
SUT	321		
SVVP	213		
symptoms	505	508	

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		

system complexity 284
system-level testing 452
system testing 233 290 331
system test plan 289

T

task planning method 330
TCE 345
technical review 206
template class 451
test 34
  case 34
  design 47
  execution 49
  factors 52
  incident 35
  phase 52
  planning 47
  strategy matrix 52
  test oracle 35
  testware 35
testability 380
test automation 429
test case effectiveness metric 345
test case generator 433
test case prioritization 355
  coverage-based 356
  general test case prioritization 355
  types of 355
  version-specific 355
test case specifications 273
test coverage 326

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		
test data generator	433		
test design	215		
TestDirector	438		
TestStudio	434		
test driver	274		
test effort estimation	335		
test enablers	430		
test engineers	274	276	
tester productivity measures	344		
testers	275	276	
test execution and evaluation tools	433		
test harness	286	288	296
test incident report	43		
testing	55		
life cycle model	59		
measurement objectives	318		
methodologies	43		
principles	57		
tactics	58		
tools	57		
unit testing	274		
testing activity tools	432		
testing cost estimation	322		
testing efforts	327	332	
testing group	275		
testing improvement model	406		
testing metrics	341		
testing of OO classes	455		
feature-based testing of classes	455		
role of invariants in class testing	456		
state-based testing	457		
testing feature groups	456		

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		
testing of web-based systems	474	484	
testing techniques	295	417	
testing tools	429	430	431
categorization	431		
costs incurred	435		
selection	434		
test leader	273		
test log	273		
test management	275		
key elements of	273		
test management tools	434		
test manager	273		
test maturity matrix	409	412	413
test maturity models	404	406	413
test monitoring	274		
test organization	80	418	
test organization model	408		
test plan	80		
acceptance	80		
function	80		
integration	80		
system	277		
unit	277		
test plan	212	398	
components	282		
hierarchy	228		
test planning	295	417	
test point analysis	335		
test procedure method	295	329	330
test process	406		
test process management	412		
test process maturity	405		

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links			
test process maturity models	405	406		
test process optimization	421			
test result specifications	274			
test script language	437			
test sessions	292			
test specialist	273			
test specifications	222	262		
test suite	296	298	352	354
test suite prioritization	354			
test suites	353			
test summary report	308			
test team size	330			
test tools	338			
testware	339			
testware management	412			
TIM	406			
TMM	405			
assessment model	422			
components	413			
TMM levels	415			
token count	73			
TOM level	408			
top-down integration	222			
total quality management	73			
total test sessions	335			
TPA	397			
TPI	405			
TPI model	409			
TQM	232			
traceability	289			
backward	320			
forward	437			

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links			

U

UFP	312			
UML	447	466	467	468
UML-based modeling	481			
conceptual modeling	481			
configuration modeling	483			
navigation modeling	481			
presentation modeling	482			
task modeling	482			
web scenarios modeling	482			
UML-based OO testing	466			
UML diagrams	467			
UML modeling	448			
unadjusted function point	312			
unified modeling language	447			
unit testing	213	331		
unit test plan	286			
unit validation testing	78			
unit verification	163			
unstructured loops	242			
usability testing	166	486	487	
usage node	166			
computation	166			
predicate	449			
use-case diagram	448	454	467	
use-case model	448			
actors	448			
use-case diagrams	448			
use-case templates	454			

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		
V			
VAF	313		
validation	54	284	419
activities	79		
test plan	79		
validation activities	282	454	
validation plan	284		
validation test execution	212		
validation testing	66	284	
validation test plan	285		
value	313		
verification	70	77	282
activities	75		
of architectural design	75		
of data design	74		
of high-level design	76		
of low-level design	71		
of objectives	70		
of requirements	76		
of user-interface design	74		
verification test plan	285		
verify	77		
code	74		
high-level design	77		
low-level design	71		
V-testing model	56	69	
V&V diagram	285	378	
W			
walkthroughs	296	381	419
watch points	506		

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links			
WBS	280			
web applications	438	452	474	475
web-based applications	475			
web-based software	476			
web browsers	477			
web crisis	480			
web-enabled applications	475			
web engineering	480			
web page	475			
website	475			
web software	478			
web technology	475			
3-tier	475			
first generation/2-tier	475			
N-tier	475			
white-box testing	58	417		
WinRunner	437			
work breakdown structure	280	322		
worst-case testing method	92			

X

XSS	492

This page has been reformatted by Knovel to provide easier navigation.