Software Testing

Principles and Practice

First Edition/Online Version

Stephen Brown, Joe Timoney, Tom Lysaght, Deshi Ye

Software Testing
Principles and Practice

First Edition

A collaboration between the Department of Computer Science, NUI Maynooth, Ireland, and the College of Computer Science, Zhejiang University, China

Authorized English language edition entitled “Software Testing: Principles and Practice” by Stephen Brown, Joe Timoney, Tom Lysaght, Deshi Ye.

English language edition published by China Machine Press.

This edition is manufactured in the People’s Republic of China, and is authorized for sale and distribution Worldwide.

Preface

This book is based on a series of lectures given at the National University of Ireland, Maynooth and Zhejiang University. It provides a textbook for a number of courses, describing the fundamentals of software testing. The material has been developed over the past ten years, and reflects both the experiences from 20 years in industry from one of the authors, and the authors' joint experiences in lecturing.

There is no one standard textbook on software testing, and this book is the result of many years of extracting and interpreting test techniques from a wide and varied number of sources. These include testing classics such as The Art of Software Testing by Myers, Software Testing by Roper, and Testing Object-Oriented Systems by Binder; standard textbooks such as Software Engineering by Pressman and Ince, and Software Engineering by Somerville; software process books such as Software Testing in the Real World by Kit and extreme Programming explained by Beck; and ISO and IEEE standards related to software quality and testing.

Software testing is a challenging task – it is as important for businesses and government as it is for research institutions. It is still as much an art as a science: there are no accepted standards or norms for applying the various techniques, and interpretation is required. There is no well established research on the effectiveness of different approaches. The techniques are easy to understand, but generally difficult to apply to real-world software. By providing extensive worked examples, this book aims to provide a solid basis for both understanding, and applying, various test techniques

序言

本书的内容基于爱尔兰国立大学梅努斯和浙江大学的一系列课程讲稿。该书覆盖了软件测试的基本原理，可以作为许多课程的参考教材。本书的内容历经十年发展，融合了其中一位作者近二十年的工业界经验及两位作者的教学经验。

在软件测试领域，目前还没有统一的标准教科书，而本书是通过对各种不同的软件测试技术进行多年的提炼、阐释而形成。这些测试技术的来源包括一些经典的测试书籍如 Myers 的《软件测试的艺术》、Roper 的《软件测试》、Binder 的《面向对象系统测试》；标准的软件工程教材如 Pressman 和 Ince 的《软件工程》、Somerville 的《软件工程》；软件过程类书籍如 Kit 的《现实世界中的软件测试》、Beck 的《极限编程》以及软件质量及测试相关的 ISO 和 IEEE 标准。

软件测试是一项具有挑战性的任务，它对科研机构、企业及政府具有同等的重要性。软件测试既是门科学也是门艺术。针对如何应用不同的软件测试技术，目前还没有广泛认同的标准，因此对各种软件测试技术的阐释是必须的。针对各种不同技术的有效性，目前还没有成熟的研究成果。总的来说，各种测试技术容易理解，但如何将其应用到软件产品中是困难的。本书旨在通过广泛丰富的实例，为读者对理解和如何应用各种不同的软件测试技术提供一个坚实的基础。

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Software Industry

The software industry has come a long way since its beginnings in the 1950's. The independent software industry was essentially born in 1969 when IBM announced that they would stop treating their software as a free add-on to its computers, and instead would sell software and hardware as separate products. This opened up the market to external companies that could produce and sell software for IBM machines.

Software products for mass consumption arrived in 1981 with the arrival of PC-based software packages. Another dramatic boost came in the 1990's with the arrival of the World Wide Web, and in the 2000's with mobile devices. In 2010 the Top 500 companies in the global software industry had revenues of $492 billion. The industry is extremely dynamic and continually undergoing rapid change as new innovations appear. Unlike some other industries, for example transportation, it is still in many ways an immature industry. It does not, in general, have a set of quality standards that have been gained through years of hard-won experience.

Numerous examples exist of the results of failures in software quality and the costs it can incur. Famous incidents include the failure of the European Space Agency's Ariane 5 rocket, the Therac-25 radiation therapy machine, and the loss of the Mars Climate observer in 1999. A study by the US Department of Commerce's National Institute of Standards and Technology in 2002 estimated that the cost of faulty software to the US economy was up to $59.5 billion per year.

However, many participants in the industry do apply quality models and measures to the processes through which their software is produced. Software Testing is an important part of the Software Quality assurance process, and is an important discipline within Software Engineering. It has an important role to play throughout the software development lifecycle, whether being used in a Verification and Validation context, or as part of an actual test-driven software development process such as Extreme programming.

Software Engineering as a discipline grew out of the "Software Crisis". The term Software Crisis was first used at the end of the 1960's but it really began to have meaning through the 1970's as the software industry was growing. This reflected the increasing size and complexity of software projects combined with the lack of formal procedures for managing such projects.

This organizational poverty resulted in a number of problems:

- Projects were running over-budget
- Projects were running over-time
- The Software products were of low quality
- The Software products often did not meet their requirements
- Projects were chaotic
- Software maintenance was very difficult

If the software industry was to keep growing, and the use of software was to become more widespread, this situation could not continue. The solution was to be in formalizing the roles and responsibilities of software engineering personnel. These software engineers would plan and document in detail the goals of each software project and how it was to be carried out, they would manage the process via which the software code would be created, and they would ensure that the end result had attributes to show that it was a quality product. This relationship between quality management and software engineering meant that software testing would be integrated into its field of influence. Moreover, the field of software testing was also going to have to change if the industry wanted to get over the "Software Crisis".

While the difference between debugging a program and testing a program was recognized by the 1970's, it was only from this time on that testing began to take a significant role in the production of software. It was to change from being an activity that happened at the end of the product cycle, to check that the product worked, to an activity that takes place throughout each stage of development, catching faults as early as possible. A number of studies comparing the cost of early vs late defect detection have all reached the same conclusion: the earlier the defect is caught, the less the cost of fixing it.

The progressive improvement of software engineering practices has led to a significant improvement in software quality. The short-term benefits of software testing to the business include improving the performance, interoperability and conformance of the software products produced. In the longer term, testing reduces the future costs, and builds customer confidence.

Nowadays, many software development processes have integrated software testing within their activities. Furthermore, processes such as "Test Driven Development" use testing to lead the code development.

1.2 Software Testing and Quality

Proper software testing procedures reduce the risks associated with software development. This is because modern software programs are very complex, having thousands of lines of code. And they are often attempting to solve a problem that has been defined in very abstract terms, described as a vague set of requirements lacking in exactness and detail. Quality problems are further compounded by external pressures on developers, from business owners, imposing strict deadlines and budgets in order to reduce the time to market and associated production costs. However, these pressures can result in inadequate software testing procedures, leading to reduced quality. Poor quality leads to more failures, increased development costs, and delays in attaining product stability. More severe outcomes for a business can be a loss in reputation, combined with market share and even having to face legal claims.

International Standard ISO 9126[1] *Software Engineering – Product Quality* is structured around six main attributes. These are given with their characteristics in Table 1.1.

[1] Recently updated in the ISO 25000 family of standards

Table 1.1: Software Quality Attributes ISO 9126

Attribute	**Characteristics**
Functionality	Suitability, accurateness, interoperability, compliance, security
Reliability	Maturity, fault tolerance, recoverability
Usability	Understandability, learnability, operability
Efficiency	Time behavior, resource behavior
Maintainability	Analysability, changeability, stability, testability
Portability	Adaptability, installability, conformance, replaceability

Attributes that can me measured objectively, such as performance and functionality, are easier to test than those which require a subjective opinion, such as learnability and installability.

1.3 Errors, Faults and Failures

Leaving aside the broader relationship between software testing and the attributes of quality for the moment, the most common application of testing is to search for defects present in a piece of software and/or verify that particular defects are not present in that software. The term software defect is often expanded into three classes: errors, faults, and failures.

1. **Errors**: these are mistakes made by software developers. They exist in the mind, and can result in one or more faults in the software.
2. **Faults**: these consist of incorrect material in the source code, and can be the product of one or more errors. Faults can lead to failures during program execution.
3. **Failures**: these are symptoms of a fault, and consist of incorrect, or out-of-specification behaviour by the software. Faults may remain hidden until a certain set of conditions are met which reveal them as a failure in the software execution.

1.3.1 Software Faults

It is helpful to have a classification of the types of faults. The classification can be used for a number of purposes:

1. when analysing/designing/coding software, where is acts as a checklist of faults to avoid
2. when developing software tests, where it acts as a guide to likely faults
3. when undergoing software process evaluation or improvement, where it provides input data

There are a number of different ways to categorise these software faults – but no single, accepted standard, as the significance and frequency of faults can vary depending on the circumstances. One representative scale [2] identifies the following ten fault types:

Algorithmic These occur when a unit of the software does not produce an output corresponding to the given input under the designated algorithm

Syntax These occur when code is not in conformance to the programming language specification

[2] Pfleeger and Atlee, *Software Engineering: Theory and Practice*

Computation and Precision These occur when the calculated result using the chosen formula does not conform to the expected accuracy or precision

Documentation Incomplete or incorrect documentation will lead to Documentation faults

Stress or Overload These happen when data structures are filled past their specific capacity whereas the system characteristics are designed to handle no more than a maximum load planned under the requirements

Capacity and Boundary These occur when the system produces an unacceptable performance because the system activity may reach its specified limit due to overload

Timing or Co-ordination These are typical of real time systems when the programming and coding are not commensurate to meet the co-ordination of several concurrent processes or when the processes have to be executed in a carefully defined sequence

Throughput or Performance This is when the developed system does not perform at the speed specified under the stipulated requirements

Recovery This happens when the system does not recover to the expected performance even after a fault is detected and corrected

Standards and Procedure These occur when a team member does not follow the standards deployed by the organization which will lead to the problem of other members having to understand the logic employed or to find the data description needed for solving a problem

A study by Hewlett Packard on the frequency of occurrence of various fault types, found that 50% of the faults were either Algorithmic or Computation and Precision.

1.3.2 Software Failures

Classifying the severity of failures that result from particular faults is more difficult because of their subjective nature, particularly with failures that do not result in a program crash. One user may regard a particular failure as being very serious, while another may not feel as strongly about it. Table 1.2 shows an example of how failures can be classified by their severity.

Table 1.2: Classification of Software Failures

Failure Severity Level	**Behaviour**
1	Failure causes a system crash and the recovery time is extensive; or failure causes a loss of function and data and there is no workaround
2	Failure causes a loss of function or data but there is manual workaround to temporarily accomplish the tasks
3	Failure causes a partial loss of function or data where the user can accomplish most of the tasks with a small amount of workaround
4	Failure causes cosmetic and minor inconveniences where all the user tasks can still be accomplished

Software failures demonstrate a characteristic pattern, but it is different from that for hardware failures. A curve for the failure rate of software is shown in Figure 1.1.

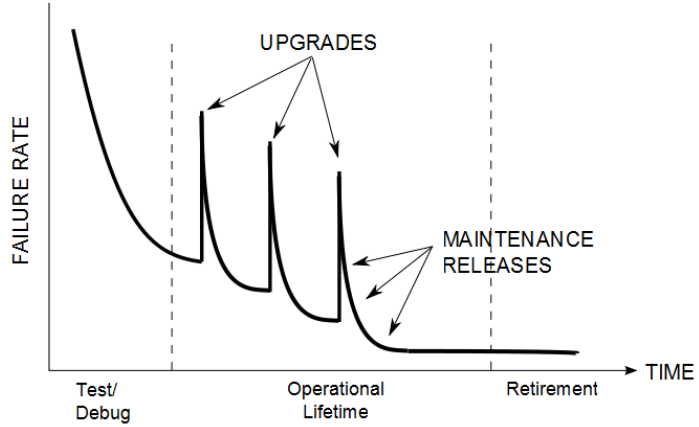


Figure 1.1: Failure Rate for a Software Product over its Lifecycle

Unlike hardware, software does not physically wear out, so it does not show the same "bathtub curve" typical of increasing failures over time. However, it is subject to ongoing changes even after release, and it is also subject to changes in the external software environment (such as an operating system upgrade). These changes can introduce new faults that did not exist in the original version, or expose latent faults that did exist. The first internal version for test and debug has a high failure rate. But as the faults are found and corrected, the failure rate decreases, until it reaches an acceptable level for release. Subsequent changes to introduce new features (upgrades) lead to increases in the failure rate, until these faults are subsequently fixed (maintenance releases). Once the software is no longer supported, the failure rate levels off, until the software product is taken off the market.

1.3.3 Need for Testing

There are a number of reasons why software has faults, or is perceived to have faults:

- It is difficult to collect the user requirements correctly.
- It is difficult to specify the required software behaviour correctly.
- It is difficult to design software correctly.
- It is difficult to implement software correctly.
- It is difficult to modify software correctly.

There are two engineering approaches to developing correct systems: one is *forward* engineering, the other is based on *feedback*.

The ideal software development project starts with the user, and ends with a correct implementation. The development is completely reliable: each activity is based on the outputs of the previous activity, and produced correct outputs. The end product matches its specification, and meets the user's needs (see Figure 1.2).

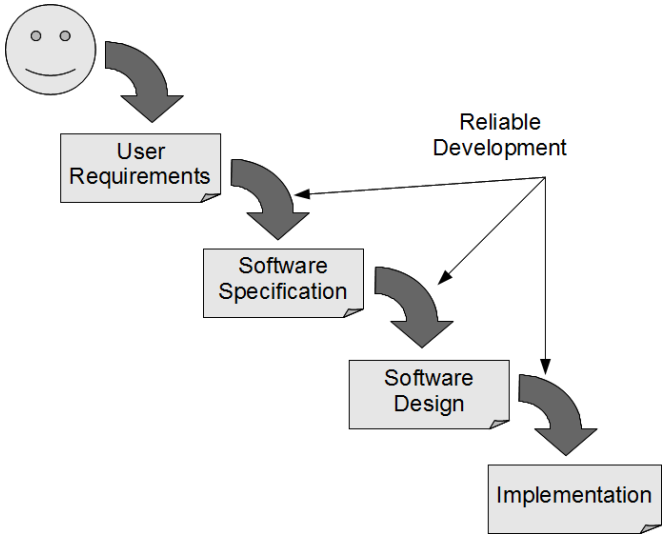


Figure 1.2: Ideal Project Progression using Forward Engineering

In practice, however, all the development steps are subject to mistakes and ambiguities, leading to non-ideal results. To resolve this, each of the steps must be subject to a check to ensure that it has been carried out properly, and to provide an opportunity to fix any mistakes before proceeding (see Figure 1.3).

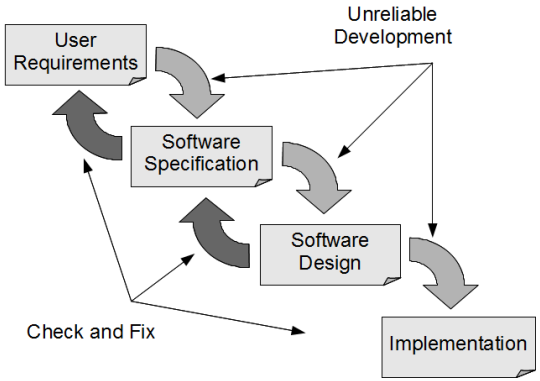


Figure 1.3: Realistic Project Progression including Checks

For software products, there are two forms of checking: checking that the output of each step meets its specifications (verification), and checking that the software meets the users needs (validation) – see Figures 1.4 and 1.5.

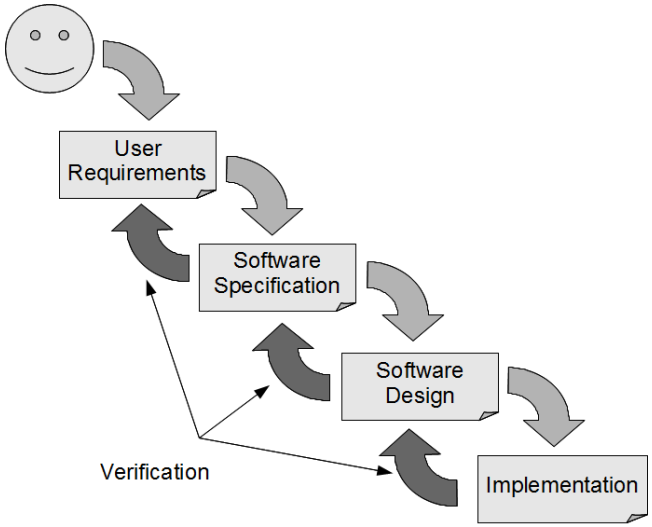


Figure 1.4: Verification in the Development Process

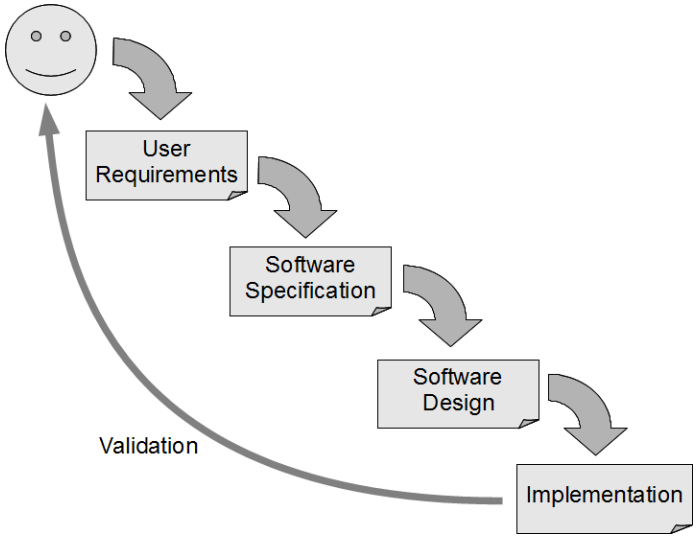


Figure 1.5: Validation in the Development Process

1.4 The Role of Specifications

Specifications play a key role in testing. In order to be tested, the correct behaviour of software must be known. This implies the need for detailed specifications (or software requirements).

To support thorough testing of the code, specifications must describe both *normal* and *error* behaviour. Normal behaviour is the expected ouput when the inputs are not in error. Error behaviour is the expected output when one or more inputs are in error, or will produce an error in processing.

Attempts have been made to develop test approaches based on "reasonable" behaviour of software – often deduced from the name of the method. This fails as not every developer and tester will have the same expectations of reasonable behaviour, especially for error cases. Some examples:

- If an invalid input is provided, should the method ignore it, return an invalid value, raise an exception, or write to an error log?
- If a temperature is calculated, should it be returned in degrees Centigrade (°C), Kelvin (°K), or Fahrenheit (°F)?
- Is zero to be treated as a positive number? negative number? or neither?
- Can an angle have a value of 0°? Can a shape have an area of 0 cm^2? Or a negative area?

In summary, in order to test software properly, specifications are necessary.

Note that often a tester will need to convert a specification from a "written English" form to one that is easier to create tests from (such as a table).

1.5 Overview of Testing

1.5.1 Testing in the Development Process

Software of any degree of complexity has three key characteristics:

1. User requirements, stating the needs of the software users.
2. A functional specification, stating what the software must do.
3. A number of modules, that are integrated together to form the final system.

Verifying each of these leads to a sequence of four key test activities:

Unit Testing An individual unit of software is tested to make sure it works correctly. This unit may be a single component, or a compound component formed from multiple individual components. For example, a component might be a function or method, a class, or a sub-system. The unit may also be a single graphical user interface component (such as a button), or a collection of them (e.g. a window). Unit testing almost invariably makes use of the *programming* interface of the unit. Test data is selected to ensure that the unit satisfies its specification.

Integration Testing Two or more units are tested to make sure they interoperate correctly. The testing may make use of the programming interface (typically for simple components), or possibly the *system* interface (for sub-systems). Test data is selected to exercise the interactions between the integrated units.

System Testing The entire software system is tested as a whole to make sure it works correctly, and subsequently to make sure it meets the user's needs/solves the user's problem. The testing uses the *system* interface: this may be a Graphical User Interface for an application (GUI), web interface for a web-server, network interface for a network server, etc. Test data is selected to ensure the system satisfies the specification.

Acceptance Testing The entire software system is tested as a whole to make sure it meets the user's needs/solves the user's problem. The testing uses the *system* interface: this may be a Graphical User Interface for an application (GUI), web interface for a web-server, network interface for a network server, etc. Test data is selected to ensure the software system meets the users needs or solves their problem.

1.5.2 Test Automation

Manual testing is slow, error-prone, and hard to repeat. Software testing needs to be fast, accurate, and repeatable. The answer is test automation. Unit tests are normally automated by default, when writing a test program to call a function of method it is natural to check the return value inline. System tests are more difficult to automate, but in general there is a shift from manual to automated system testing. Test automation requires that the correct result be known in advance – in manual testing, this can be left to the tester's judgement.

Test automation can include one or more of the following features:

- automated test execution
- automated collection of results
- automated evaluation of results
- automated report generation
- automated measurement of test coverage
- automated generation of test data

Automated tests can be collected into Test Suites, and the results automatically collated into a report. The report usually includes an overall summary (pass or fail);

statistics on the number of tests run, that failed, that passed, and that did not run properly; and a test incident report on each failure, providing exact details on why the test failed in order to assist in locating the fault.

1.6 The Theory of Testing

The goal of developing a theory of software testing is to be able to identify the ideal test – that is the minimum test data required to ensure that software works correctly for all inputs. The fundamental result in this area [3] can be summarised as follows:

- For a test to be successful, all test data within the test set must produce the results as defined by the specification
- The test data selection criteria is reliable if it consistently produces test sets which are successful, or it consistently produces test sets which are not successful
- If a set of test data is chosen using a criterion that is reliable and valid, then the successful execution of that test data implies that the program will produce correct results over its entire input domain
- Leading to the result: the only criterion that can be guaranteed to be reliable and valid is one that selects the entire program domain (i.e. exhaustively tests the program)

Exhaustive testing (i.e. testing software with every possible combination of inputs) is not in general *feasible* – that is it would take too long, or require too much memory space. A quick example demonstrates this.

1.6.1 Exhaustive Testing Example

Consider a method bound() as defined below:

```
// return a value of x bounded by the upper and lower values
// return lower if x<=lower
// return upper if x>=upper
// return x if lower<x<upper
long bound(int lower, int x, int upper);
```

If *int* is defined as a 32-bit, signed integer, then each input parameter has 2^{32} possible input values. Exhaustive testing would require using every possible combination of input values, leading to 2^{96} tests. If, on average, each test took 1 nano-second to execute – possible on a modern, multi-core processor – then the test would take 2^{96} (or about 10^{29}) nano-seconds to complete. This is approximately 10^{12} years! The sun is only $4.6 * 10^9$ years old. So clearly this is not feasible, even for a simple function.

A simple class might easily hold 12 bytes of data (class attributes), so to test every possible state of such a class is also infeasible.

1.6.2 Implications

It would be desirable to test all possible inputs, combinations of inputs, and sequences of inputs, to the system, both valid and invalid. But, as shown previously, this is normally infeasible in practice as it would take too long.

Thus, ways of reducing this number need to be found. For both time and cost efficiencies, good tests should (a) have a high probability of finding faults, (b) not duplicate one another, (c) be independent in what they measure to prevent faults concealing each other, and (d) test as much of the code as possible. Satisfying all these criteria is difficult in practice!

[3] Goodenough and Gerhart, *Toward a Theory of Test Data Selection*, 1975

The fundamental basis of all testing techniques is selecting a subset of the possible input parameters: normally, a very small subset! Therefore the quality of a test technique is determined by how effective this subset is in finding faults. Unfortunately there is no well established theory of faults, so test techniques tend to be *heuristic* techniques, based on fundamental principles and practice.

In general, there are two types of approach to selecting a subset. One is to take the specification, and generate input values that 'exercise' the specification. This is referred to as *Black-Box* testing. The other is to take the implementation (most techniques use the source code), and generate input values that 'exercise' the implementation. This is referred to as *White-Box* testing.

Exercising the specification means to have generated enough tests that every possible different specified type of processing has been executed. This not only means that every different category of output has been generated, but also that every different category of output has been generated for every different input cause.

Exercising the implementation means to have generated enough tests so that every component that makes up the implementation has been demonstrated to produce a valid result when executed. At its simplest, each line of source code is regarded as a separate component. But, as you will see in following chapters, there are more sophisticated ways of identifying components that provide for fuller coverage of a program's possible behaviours.

In all tests, just exercising the code or the specification is not enough. A test must not only do this, it must also compare the actual output to the expected output. A test passes if they are the same, and fails if they are not.

A third form of testing is based on *Fault Insertion*. This has its foundations in hardware testing, where typically a connection will be artificially held at a high or low voltage representing a "stuck at 1" or "stuck at 0" condition. The hardware is then run, typically through simulation, to ensure that the fault is detected and handled correctly by the hardware. The fault represents a data fault - all data that is transmitted down that connection will be subject to the fault. In software fault testing, faults can be inserted either into the code, or into known-good data. When inserted into *data*, the purpose is similar as for hardware testing – to ensure that the fault is detected and handled correctly by the software. When inserted into *code*, the purpose is fundamentally different: it is to measure the effectiveness of the testing.

1.7 When To Finish Testing

The decision of when testing is finished can be made from different viewpoints:

- From a budgetary point of view, when the time or budget allocated for testing has expired.
- From an activity point of view, when the software has passed all of the planned tests.
- From a risk management viewpoint, when the predicted failure rate meets the quality criteria.

Ultimately, the decision has to take all these viewpoints into account – it is rare for software to pass all of its tests prior to release. Two techniques to assist in making this decision are the *Risk of Release*, and *Usage-based Criteria*. During testing, Usage-based Criteria give priority to the most frequent sequences of program events. The Risk of Release predicts the cost of future failures, based on the chance of these failures occurring, and the estimated cost of each.

Note that there may also be ethical and legal issues involved in making this decision. Testers need to make sure that testing is adequate from a professional ethics viewpoint, especially where software is used in life-critical systems. Software test completion criteria may be an important factor if a software failure results in litigation.

1.8 Notes on Book Structure

This book is intended to be read in sequence, building up from general principles to specific examples of testing. A brief description of the structure is given below to help the reader understand the order in which material will be introduced:

Chapter 2 – Principles of Software Testing

This chapter describes the underlying principles of Black-Box Testing, White-Box Testing, and Fault Insertion.

Chapter 3 – Unit Testing

The application of these principles to Unit Testing is described.

Chapter 4 – Unit Testing Examples

This chapter shows how to apply these techniques in detail, using a number of worked examples to assist the reader.

Chapter 5 – Static Verification

This provides a brief introduction to static verification techniques.

Chapter 6 – Testing Object-Oriented Software

The underlying principles of object-oriented testing are described, and then their application to testing object-oriented software is described with examples.

Chapter 7 – Integration and System Testing

This chapter defines Integration and System Testing, and discusses the application of the underlying principles as described in Chapter 2, to these forms of testing.

Chapter 8 – Software Test Automation

Test automation is a key principle of modern testing practices. This chapter describes a number of representative tools, and provides some examples of how to use them.

Chapter 9 – Testing in the Software Process

Various software development process models are described, with an emphasis on describing where software testing fits into them.

Chapter 10 – Advanced Testing Issues

Software testing is a very open and active research field – there are many unanswered questions. This chapter gives the reader an idea of how some selected problems might be addressed, and gives a very brief overview of active research directions.

In normal use, Black-Box Testing comes first, and then White-Box Testing follows to selectively increase coverage measures (such as percentage of statements executed). This avoids duplication of tests. This book follows the same ordering, with Black-Box techniques being addressed before White-Box ones in each chapter.

The Java programming language has been used for all examples. Most of them can be converted to C or C++ in a straightforward manner. The principles shown, and the approach used for Java, serves as an example for testing in similar procedural or object-oriented languages.

Chapter 2

Principles of Software Testing

This chapter introduces the key principles that underlie the different software test techniques described later. The principles of Black-Box and White-Box Testing can be applied to testing at different stages of development: to *Unit Testing* as the code is developed, to *Integration Testing* as code is incrementally added to the system, and to *System Testing* for each release of the final system.

Code can be verified *statically* by reviewing or analysing the code, and *dynamically* by executing the code. These two approaches are described in the following sections.

2.1 Dynamic and Static Verification

The fundamental difference between these approaches is that Static Verification does not require the execution of the software code, while Dynamic Verification does.

2.1.1 Static Verification

Static Verification (or Static Analysis) can be as straightforward as having someone of training and experience reading through the code to search for faults. It could also take a mathematical approach consisting of symbolic execution of the program, or a formal approach consisting of symbolic verification of the translation between the specification and the source code. The main problem with these two latter approaches is that they can be very time-consuming, and thus may only be suitable in certain cases where the cost of achieving very high quality is justified. A number of tools are available to aid static program analysis.

2.1.2 Dynamic Verification

Dynamic Verification (or Software Testing) confirms the operation of a program by executing it. Test Cases are created that guide the selection of suitable Test Data (consisting of Input values and Expected Output values). The Input values are provided as inputs to the program during execution, the Actual Outputs are collected from the program, and then they are compared with the Expected Outputs.

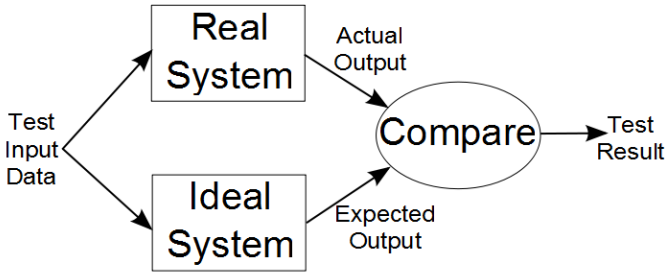


Figure 2.1: Software Tests Procedure

As shown in Figure 2.1, the same test inputs are applied to the Ideal System and the Real System, and the actual and expected outputs are compared, leading to a Test Result. The Ideal System is represented by the specification, while the Real System consists of the actual code. The Ideal System may be referred to as the Test Oracle, especially when it is automated. The test data is derived from the set of test cases, each of which has been chosen to validate particular features or aspects of the system. The result of the test is a pass or fail. However, for a test to be successful it does not have to produce a pass result. This is because even a failed test will give us some new knowledge about the system: it will tell us for which inputs the system is not working as specified.

2.2 Black-Box and White-Box Testing

Black-Box testing (also referred to as Specification-Based or Functional Testing) is based entirely on the program specification, and uses no knowledge of the inner workings of the program code. Black box testing aims to verify that the program meets the specified requirements, without any knowledge of how the program works internally. By exploring different behaviours, as defined in the specification, the tests can expose faults where the implementation does not match the specification.

Black-Box Testing techniques are characterised by how a subset of the possible combinations of input values has been selected. For example, *boundary value analysis* means to select all the possible boundary values for every input and output parameter.

Key Point: Black-Box Test Cases and Test Data are derived solely from the functional specifications.

White-Box testing (also referred to as Implementation-Based or Structural Testing) uses the *implementation* of the software to derive tests. The tests are designed to exercise some aspect of the program code, such as the statements or the decisions it contains. By examining how the program works, and selecting test data to cause specific components of the program to be executed, the tests can expose errors in the program structure or logic. Test cases and test data are derived both from the implementation and the functional specifications.

White-Box Testing techniques are characterised by their *Coverage Criteria*. This refers to the percentage of the components have been exercised by the test. For example, 100% *statement coverage* means that all the lines of source code in a program have been executed at least once during the test.

Key Point: White-Box Test Cases are derived from the code; the Test Data is derived from both the code and the functional specifications.

Table 2.1 compares some of the key characteristics of Black-Box and White-Box Testing.

Table 2.1: Comparison of Black-Box and White-Box Testing

Black-Box Testing	**White-Box Testing**
Tests are only dependent on the specification.	Tests are dependent on both the implementation and the specification.
Tests can be re-used if the code is updated to fix a fault, or additional functionality is added.	Tests are generally invalidated by any changes to the code.
Tests only require the specification to be developed.	Tests require that the code is written prior to developing the tests.
Tests do not ensure that all the code components have been exercised however, and thus are liable to missing "errors of commission" where extra functionality has been added to the code that is not in the specification.	Tests ensure that all the (selected) components have been exercised, however they do not ensure that the specification has been fully covered. They are thus liable to missing "errors of omission" where required functionality has not been implemented.
In general, it is difficult to automate measuring the effectiveness of Black-Box tests (i.e. how much of the specification has been covered).	In general, it is relatively easy to automate measurement of the effectiveness of White-Box tests (i.e. how much of the implementation has been covered).

2.2.1 Errors of "Omission" and "Commission"

Both Black-Box and White-Box Testing have weaknesses. White-Box Testing does not typically find faults relating to missing functionality (referred to as *Errors of Omission*). Black-Box Testing does not typically find faults relating to extra functionality (referred to as *Errors of Commission*).

Consider the example `boolean isZero(int x)`, that returns true if x is zero, otherwise returns false, with the following source code:

```
1  public static boolean isZero(int x)
2  {
3     boolean rv=true;
4     if (x>=0)
5        rv=false;
6     else if (x<-1)
7        rv=false;
8     return rv;
9  }
```

This contains two errors: one of omission, and one of commission.

1. The method does **not** return true when $x == 0$. This is missing functionality, and therefore an error of omission. In this case it is caused by a fault on Line 4, which should read `if (x>0)`.

2. The method returns true **incorrectly** when $x == -1$. This is added functionality, not in the specification, and therefore an error of commission. The fault is in Line 6 which, for this algorithm, should read `else if (x<0)`.

This is a simple example, with what are essentially two *typos* (minor typing errors) in the code. Error handling is often a source of omission faults in complex code, where incorrect input data is not identified correctly. Misunderstanding the specification is often a source of faults of commission, where extra input conditions are identified by the programmer which were not in the specification.

Another way of looking at this is from the coverage viewpoint:

Black-Box testing provides for coverage of the *specification*, but not full coverage of the *implementation*. That is, there may be code in the implementation that produces results not stated in the specification.

White-Box testing provides for coverage of the *implementation*, but not of the *specification*. That is there may be behaviour stated in the specification for which there is no code in the implementation.

It is for these reasons that Black-Box testing is done first, to make sure the code supports the specification. This is then augmented by White-Box testing, to make sure the code does not provide anything extra (not in the specification).

2.3 Test Approaches

The purpose of software testing is to measure the quality of the software. Two complementary ways of achieving this have been developed: by generating tests from the specification (Black-Box Testing), and by generating tests from the implementation (White-Box Testing). A third approach, the purpose of which is both to detect faults and to measure the quality of testing, is fault insertion (Mutation Testing).

These three approaches to deriving tests are described in more detail in the following sections.

2.3.1 Black-Box Testing

The basic principle of Black-Box Testing can be expressed in a number of different ways:

- Test against the specification.
- Use test coverage criteria based on the specification.
- Develop test cases derived from the specification.
- "Exercise" the specification.

They all reflect the idea of testing the function of the software using the specification as a source of test cases.

Regarding software as a mapping from input values to output values, the purpose of Black-Box Testing is to ensure that every value in the input domain maps to the correct value in the output domain as shown in Figure 2.2.

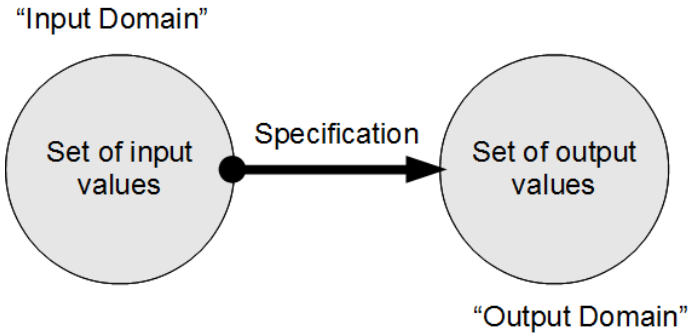


Figure 2.2: Specification Model of Testing

This is ideally achieved by exercising all the mappings in the specification. As discussed in Section 1.6.1, exhaustive testing is seldom feasible, and so a subset of possible inputs must be selected to cover key mappings between the input and output domains.

Measurement

It is difficult to automatically measure the degree of coverage of the specification that Black-Box Testing has achieved. Correct implementation of Black-Box Tests therefore relies heavily on the quality of the tester's work.

2.3.2 White-Box Testing

The basic principle of White-Box Testing can be expressed in a number of different ways:

- Test against the implementation.
- Use test coverage criteria based on the implementation.
- Develop test cases derived from the implementation.
- "Exercise" the implementation.

They all reflect the idea of testing the function of the software using the structure of the implementation as a source of test cases.

Regarding software as a set of components that create output values from input values, the purpose of White-Box Testing is to ensure that executing the components always results in the *correct* output values as shown in Figure 2.3.

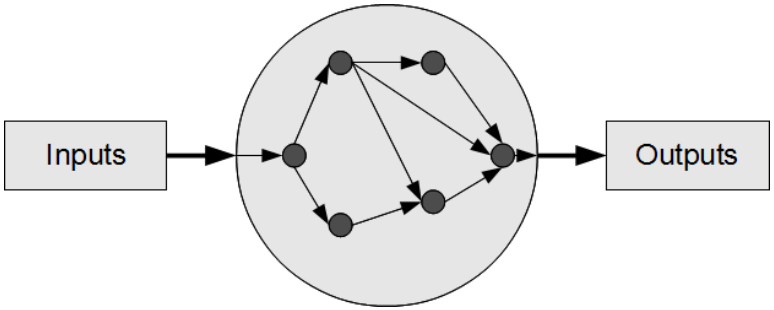


Figure 2.3: Implementation Model of Testing

This is ideally achieved by exercising all the components (including combinations and sequences of components) in the implementation. As discussed already, exhaustive testing is seldom feasible, and so a subset of possible inputs must be selected to exercise key sets of components.

Measurement

Many test environments provide tools that *instrument* the code: by recording which instructions have been executed, these tools can calculate simple coverage figures, such as percentage of lines executed, or percentage of branches taken during the tests (see Chapter 8). Whereas 100% coverage of these simple criteria does not indicate complete testing – there may be complex combinations of branches that expose a fault – less that 100% coverage indicates that there is untested code or untested branches.

These measurement tools can be used to measure the coverage provided by Black-Box Testing, and White-Box Testing can then be used to augment this in order to achieve full coverage.

Some authors suggest that setting any goal less than 100% coverage does not assure quality. However, a lot of effort may be expended attaining a coverage approaching 100%. The same effort might actually find more faults using a different test approach. For example: using static testing instead of dynamic testing. Using one of the coverage-based white box techniques, such as Statement testing, Branch testing, or Decision/condition coverage, a typical goal is to attain 80%-90% coverage or more before releasing [1].

However, these recommendations are not absolute and safety-critical software may have a higher goal. Furthermore, a higher coverage goal may also be set for unit testing than for integration testing since a failure in lower-level code may affect multiple high-level callers. One strategy that usually increases coverage quickly is to first attain some coverage throughout the entire test program before striving for high coverage in any particular area. By briefly visiting each of the test program features, obvious or gross failures are more likely to be found early on.

2.3.3 Fault Insertion

The most common technique is referred to as *Mutation Testing* where faults (or "mutations") are inserted into the source code, and the code checked to see if the mutant produces a different output. In *Weak Mutation Testing* this check is carried out manually. In *Strong Mutation Testing* this check is carried out by executing the code.

[1] A figure of 85% Branch coverage is given in Grady, *Practical Software Metrics for Project Management and Process improvement* as an appropriate figure for good characterization. A value of over 85% for Branch Coverage was given in Vinter & Poulsen, *Experience-Driven Process Improvement Boosts Software Quality*

2.4 Test Activities

In order to test software, test data must be selected to provide inputs. This requires selecting a test technique, analysing the specification and/or the source code, identifying test cases, and specifying the tests (which includes creating the test data).

For automated testing, the tests are then implemented in software. The tests are run, the program outputs collected, and these are compared against the expected outputs to produce test results.

There are four key outputs generated in the process of generating tests, produced in the order shown:

1. Analysis Outputs
2. Test Cases
3. Test Data
4. Test Code (or Test Procedure for manual testing)

Reviews can be used to ensure the quality of testing. For a test review to be performed, the reviewer must have access to these four outputs.

2.4.1 Analysis Outputs

All test designs require some form of analysis to be undertaken. This may either be analysis of the source code, or analysis of the specification. The outputs of the analysis are used to determine Test Cases. In practice, the analysis output might not be detailed – but there is value in doing so, as it allows the Test Cases to be reviewed more easily for completeness.

2.4.2 Test Cases

Test Cases are particular items to be 'covered' by a test. Test Design involves finding these test cases, and then specifying tests that cover them.

Some examples of Test Cases are:

- A particular value for a parameter to take.
- A particular relationship between two parameters to be achieved.
- A particular line of code to be executed.
- A particular branch in the code to be taken.

A test may cover multiple Test Cases – in fact the goal of most test techniques is to reduce the number of tests by covering as many Test Cases as possible in a single test. This is for the simple reason of reducing the time it takes to execute a test. For a small number of tests this is not important. But large software systems may have tens of thousands of tests, and the execution time may run into multiple days.

Any values used in stating a Test Case should be as generic as possible, referencing constants by name, and stating relationships rather than actual values where possible. This allows maximum flexibility in designing the Tests themselves, allowing a single test to cover as many cases as possible.

Test Cases are divided into two categories: 'normal' and 'error' cases. It is important to separate these as multiple normal cases may be incorporated into the same test. Multiple error cases may not. This is because of **error hiding**. Most software does not clearly differentiate between all the possible error causes: therefore in order to test each error, only one error may be tested at a time. In this book, all error cases are marked with an asterix *****.

Each Test Case requires a unique identifier, so that it can be referenced. The identifier should be unique for the Software Under Test (SUT) – to achieve this, it is often useful to use a prefix based on the test technique being used (e.g. EP-001 for Test Case 1, using Equivalence Partitioning). It can be useful to also include the prefix "TC" as well, to differentiate the Test Case identifiers (e.g. TC-EP-001).

In simple testing, all the inputs are passed as arguments to the code, and the output is returned as a return value. In practice, it is not unusual for there to be both *explicit* arguments (passed in the call to the code), and *implicit* inputs (for example, in Java, class attributes read by the code; or, in C, global variables). Also, in practice, there may both be an *explicit* return value, and other *implicit* outputs (such as, in Java, class attributes written by the code). These should all be included in the Test Cases.

In the worked examples in this book, for completeness each test is treated independently. Because of this, and to improve clarity, the Test Case identifiers and Test numbers are only unique within each test. In practice, a prefix would be added to each of these, as indicated above, to make them unique to all the tests for each SUT.

2.4.3 Test Data

The Test Data for each Test is specified based on the Test Cases.

For normal Test Cases, each Test should cover as many Test Cases as possible. For error Test Cases, each Test must only cover exactly one Test Case. Selecting data to minimize the number of tests is a skill – initially the important factor is to make sure to cover all the Test Cases. Reducing the number of tests is a secondary optimisation. With practice, it becomes easier to see these optimisations as one is selecting the test data.

A specification for a test is derived from one or more Test Cases. The specification must provide the following information:

- An identification of the SUT: name and version number.
- A unique identifier for the Test (e.g. T001 for Test 1).
- A list of the Test Cases covered by the Test.
- The Test Data. This consists of:
  - Input Values – these should be actual specific values.
  - Expected Output – this is **always** derived from the specification, **never** from the source code

It is crucial that every Test for a particular SUT has a unique identifier. A particular Test may cover Test Cases derived using different techniques, so it is not necessarily useful to include the technique name as a prefix to the test identifiers. One possible approach, used in this book, is to number the Tests for a SUT in sequence as they are designed.

The Test Data may include additional, related, information. For example, in Object-Oriented Testing, setting input values may require calling a number of different methods, and so the names of the methods and the order in which they are to be called must also be specified.

The list of Test Cases covered should be used as the guide for determining when enough tests have been created. Each Test Case must be covered if possible (sometimes impossible Test Cases may be created, which must be clearly identified in either the Test Cases or the Test Data).

It is good practice to document not only the Test Cases that are covered by each Test, but also the Test that covers each Test Case. This allows the tester, or a reviewer, to quickly ensure that every Test Case is covered.

2.4.4 Test Code (or Test Procedures)

Recent trends are to automate as many tests as possible. However, there is still a place for manual testing. This requires that a *procedure* for executing the test be documented. The documentation must describe how to setup the correct environment, how to execute the code, the inputs to be provided, and the expected outputs.

Unit Tests are always automated; Integration Tests usually are; System Tests are where possible. Implementing an automated test involves writing code to invoke the SUT with the specified input parameters, and comparing the *Actual Output* with the *Expected Output*. It is good practice to use the *same name* in Test Code as used in the Test

Specification. Each specified test should be implemented separately – this allows individual failures to be clearly identified. In practice, tests are grouped into *Test Suites* which allow a tester to select a large number of tests to be run as a block.

2.5 Analysis of Software Specifications

2.5.1 Parameters

Methods (and functions) have *explicit* and *implicit* parameters. Explicit parameters are passed in the method call. Implicit parameters are not: for example, in a C program they may be global variables; in a Java program, they may be attributes. Both types of parameter must be considered in testing. A complete specification should include all inputs and outputs.

2.5.2 Parameter Ranges

All parameters have both *natural* values, and *specification-based* ranges of values. The natural range is based on the type of the parameter. The specification-based ranges, or partitions, are based on the specified processing. It often helps in analysing ranges to use a diagram – see Figure 2.4. This figure shows a Java *int*, which is a 32-bit value, having a minimum value of 2^{-31} and a maximum value of $2^{31} - 1$ (or Integer.MIN_VALUE and Integer.MAX_VALUE), giving the natural range:

- **int** [Integer.MIN_VALUE..Integer.MAX_VALUE]

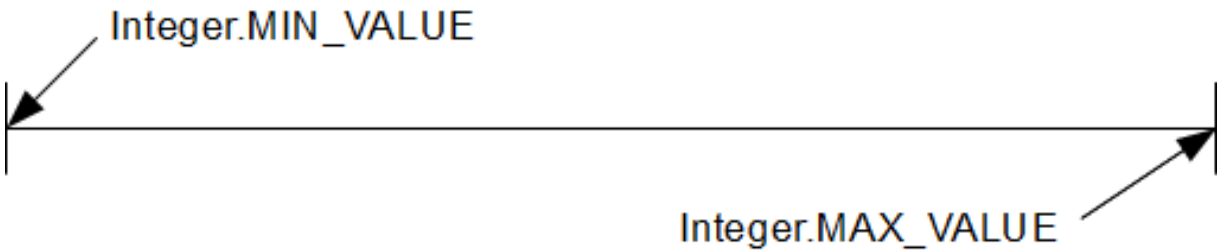


Figure 2.4: Natural Ranges

Natural ranges for a number of other common types are shown below:

- **byte** [Byte.MIN_VALUE..Byte.MAX_VALUE]
- **short** [Short.MIN_VALUE..Short.MAX_VALUE]
- **long** [Long.MIN_VALUE..Long.MAX_VALUE]
- **char** [Character.MIN_VALUE..Character.MAX_VALUE]

Natural ranges for types with no natural ordering are treated slightly differently – each value is a separate range containing one value:

- **boolean** [true][false]
- **enum Colour {Red, Blue, Green}** [Red][Blue][Green]

Compound types, such as arrays and classes are more complicated to analyse, though the principles are the same. These are considered separately in Chapter 10.

2.5.3 Equivalence Partitions

An Equivalence Partition (EP) is a range of values for a parameter for which the specification states equivalent processing.

Consider a method, `boolean isNegative(int x)`, which accepts a single Java int as its input parameter. The method returns true if x is negative, otherwise false. From this specification, two equivalence partitions for the parameter x can be identified:

1. Integer.MIN_VALUE..-1
2. 0..Integer.MAX_VALUE

These specification-based ranges are called *Equivalence Partitions* – according to the specification, any value in the partition is processed equivalently to any other value. See Figure 2.5.

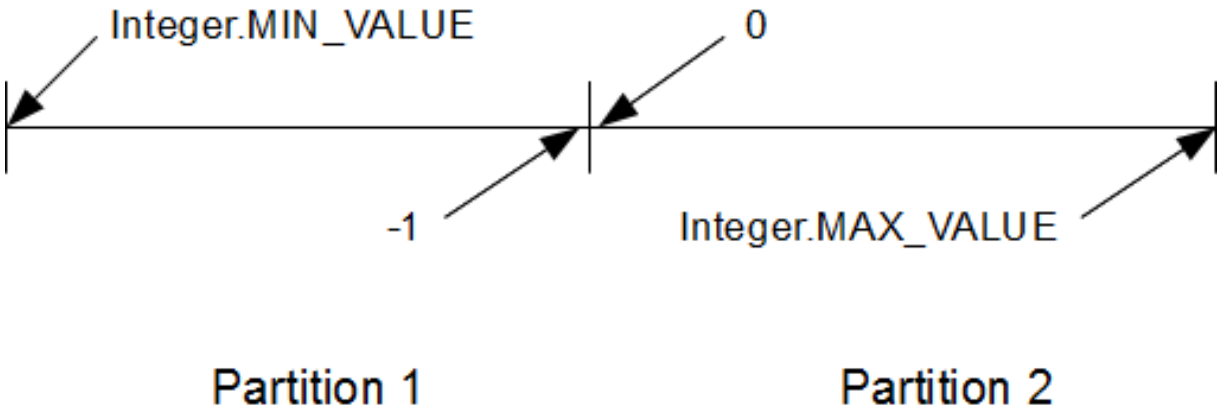


Figure 2.5: Equivalence Partitions

Both input and output parameters have natural ranges and equivalence partitions.

A Java *boolean* is an enumerated type with the values *true* and *false*. Each enumerated value is a separate range. In this example, the two different values are produced by different processing, and so the return value has two Equivalence Partitions:

1. true
2. false

Selecting Equivalence Partitions

Some guidelines for identifying Equivalence Partitions are as follows:

- Every value for every parameter must be in one equivalence partition.
- There are no values between partitions.
- The natural range of the parameter provides the upper and lower limits for partitions where not specified otherwise.

Equivalence Partitions are used in testing as, according to the specification, any one value can be selected to represent any other value. So, instead of having a separate test for *every* value in the partition, a single test can be executed using a *single* value from the partition. It is equivalent to any other value, and can be picked from anywhere in the partition. Traditionally a value in the middle of the partition is picked. Equivalence Partitions are useful for testing the fundamental operation of the software: if the software fails using EP values, then it is not worth testing with more sophisticated techniques until the faults have been fixed.

2.5.4 Boundary Values

Each Equivalence Partition has an upper and lower boundary value. Experience has shown that many software failures are due to the incorrect handling of limits, and thus Boundary Values (BV) provide a useful increase in sophistication over Equivalence Partitions. At the cost of doubling the number of tests!

For the example `isNegative(int x)`, the boundary values for x are as follows:

1. Integer.MIN_VALUE
2. -1
3. 0
4. Integer.MAX_VALUE

And the boundary value for the return value are the same as the equivalence partitions (as it is an enumerated type) – each EP is a range with only one data value:

1. true
2. false

Some rules for picking boundary values:

- Every parameter has a boundary value at the top and bottom of every equivalence partition.
- For a contiguous data type, the successor to the value at the top of one partition must be the value at the bottom of the next.
- The natural range of the parameter provides the ultimate maximum and minimum values.

It is important to note that boundary values **do not** overlap, and that there is **no gap** between partitions.
A convenient shorthand for specifying partitions and their boundary values is as follows:

x: [Integer.MIN_VALUE..-1][0..Integer.MAX_VALUE]

return value: [true][false]

2.5.5 Combinations of Values

Some programs do simple data processing just based on the input *values*. Other programs exhibit different behaviour based on the *combinations* of input values. Handling complex combinations correctly is likely to be a source of faults, and they need to be analysed to derive associated test cases.

There are a number of different techniques for identifying relevant combinations, such as Cause-Effect Graphs and Decision Tables. In this book, the recommended technique is Truth Tables – they are simplest to create, and tests can be derived directly from them.

The analysis of combinations involves identifying all the different combinations of input *causes* to the software and their associated output *effects*. The causes and effects are described as logical statements (or predicates), based on the specification of the software. These expressions specify the conditions required for a particular variable to cause a particular effect.

To identify a minimum subset of possible combinations that will test all the different behaviours of the program, a Truth Table is created. The inputs ("Causes") and outputs ("Effects") are specified as Boolean expressions (using predicate logic). Combinations of the causes are the inputs that will generate a particular response from the program. These Causes and Effects are combined in a Boolean graph or Truth Table that describes their relationship. Test Cases are then constructed that will cover all possible combinations of Cause and Effect. For N independent causes, there will therefore be a total of 2^N different combinations. The Truth Table specifies how the software should behave for each combination.

Causes and Effects

First the causes and effects for the software must be expressed as Boolean expressions. This is best demonstrated using examples, as shown below.

Example A Consider the example `boolean isNegative(int x)` as described previously.

There is a single *cause*:

1. x<0 (which can be true or false)

And a single *effect*:

1. the return value (which can be true or false)

Notes:

- For *mutually exclusive* expressions, such as "x<0" and "x>=0", only one form of the expression is needed, as it can take on the value true or false.
- For Boolean variables, the two expressions "variable==true" and "variable==false" are **not** needed. The single expression "variable" (here, return value) can take on the value true or false.

Example B. Consider the method `boolean isZero(int x)` which returns true if x is zero, and otherwise returns false.

There is a single *cause*:

1. x==0

And a single *effect*:

1. the return value

Example C. Consider the method `int largest(int x, int y)`, which returns the largest of the two input values.

The *causes* are:

1. x>y
2. x<y

And the *effects* are:

1. return value==x
2. return value==y

Notes:

- Where there are three possible situations (here, x<y, x==y, and x>y) you need at least two expressions to cover them:
  1. If x>y, then x<y is false.
  2. If x<y, then x>y is false.
  3. If x==y, then x>y and x<y are both false.
- Where the effect is for an output (here the return value) to take on one of the input values, it is important to list all the possibilities, as they are not mutually exclusive. In this case, when x is equal to y, the output is equal to both x and y.

Truth Tables

A *Truth Table* is used to map the causes and effects through *rules*. Each rule states that under a particular combination of input *causes*, a particular set of output *effects* should result.

To generate the Truth Table, each Cause is listed in a separate row, and then a different column is used to identify each combination of Causes that creates a different Effect. Each column is referred to as a Rule in the Truth Table – each Rule is a different test case.

The Truth Tables for the three examples are shown below. Note that "T" is used as shorthand for true, and "F" for false.

Table 2.2 shows the Truth Table for isNegative():

- Rule 1 states that if x<0, then the return value is true.
- Rule 2 states that if !(x<0), then the return value is false.

Table 2.2: Truth Table for isNegative()

	Rules	
	1	2
Causes		
x<0	T	F
Effects		
return value	T	F

Table 2.3 shows the Truth Table for isZero():

- Rule 1 states that if x==0, then the return value is true.
- Rule 2 states that if !(x==0), then the return value is false.

Table 2.3: Truth Table for isZero()

	Rules	
	1	2
Causes		
x==0	T	F
Effects		
return value	T	F

Table 2.4 shows the Truth Table for largest(). Note that there is no Rule 4, as the combination "(x>y) and (x<y)" is not logically possible:

- Rule 1 states that if (x>y) and !(x<y), then the return value is x.
- Rule 2 states that if !(x>y) and (x<y), then the return value is y.
- Rule 3 states that if !(x>y) and !(x<y), implying that (x==y), then the return value is equal to both x and y.

Table 2.4: Truth Table for largest()

	Rules		
	1	2	3
Causes			
x>y	T	F	F
x<y	F	T	F
Effects			
return value==x	T	F	T
return value==y	F	T	T

"Don't Care" Conditions

"Don't care" conditions exist where the value of a *cause* has no impact on the *effect*. These "Don't care" conditions are used to reduce the number of rules where the same output will be generated irrespective of whether the Cause is true or false. In the worst case, if there are no "Don't care" conditions, N Causes will create 2^N Rules. "Don't care" conditions are represented by a "*" for the *causes* in a Truth Table (see examples below).

Also, an "*" is used where an *effect* might be either true or false – the value is not determined by the specification.

An additional example serves to show how to use "Don't care" conditions. Method `boolean condIsNeg(int x, boolean flag)` returns true when x is negative and the value of flag is true, and always returns false if the value of flag is false.

The causes are:

1. x<0
2. flag

and the effect is:

- return value (which can be true or false)

From these the Truth Table shown in Table 2.5 is produced. Note the *don't-care* condition for the cause x<0 when the flag is false.

- Rule 1 states that when x<0 and flag, the return value is true.
- Rule 2 states that when !(x<0) and flag, then the return value is false.
- Rule 3 states that when !flag, the return value is false.

Table 2.5: Truth Table for condIsNeg(int,boolean)

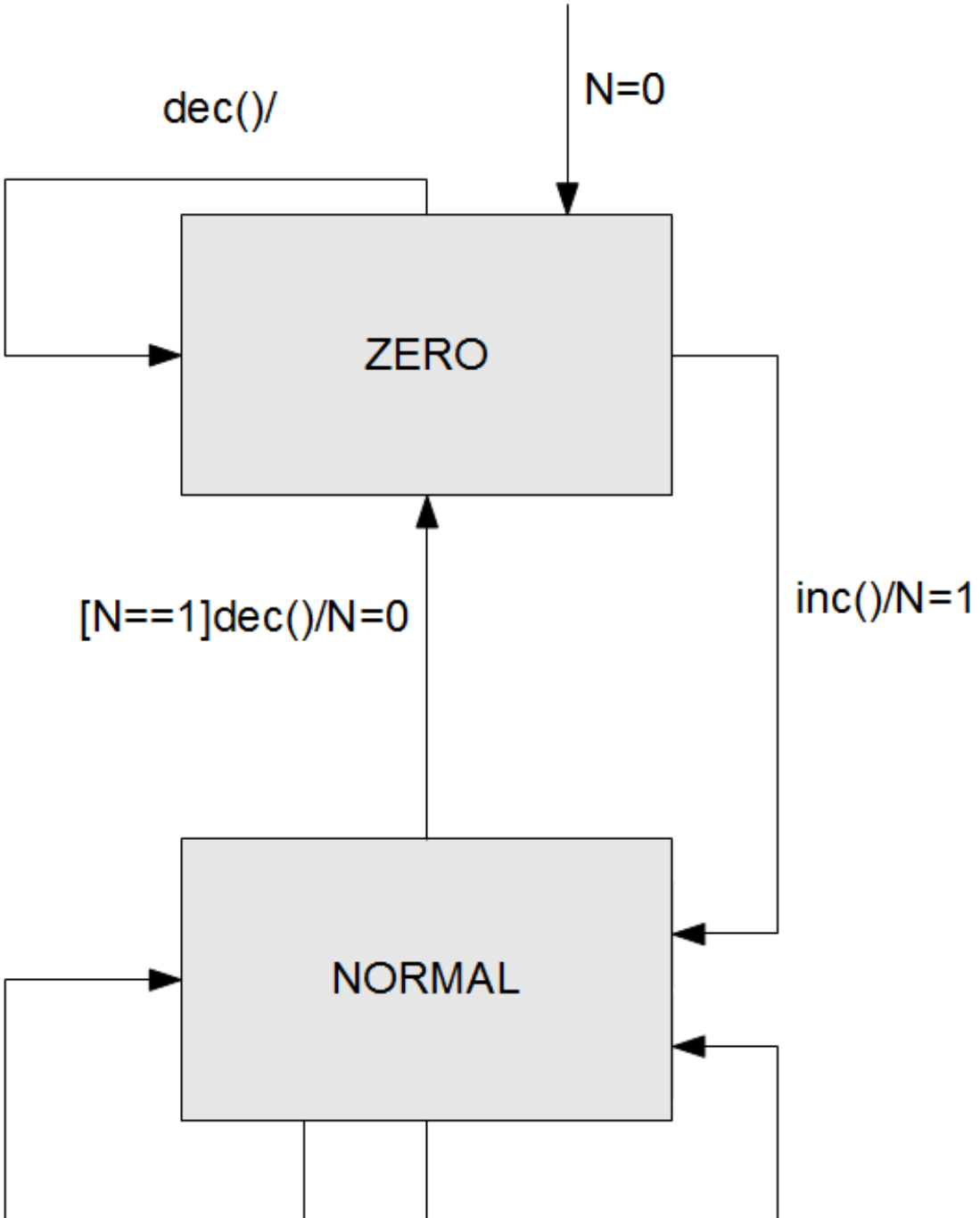
	Rules		
	1	2	3
Causes			
x<0	T	F	*
flag	T	T	F
Effects			
return value	T	F	F

2.5.6 Sequences of Values

Sequences of values are important where the software preserves state. The response to a particular input value can vary depending on the state, and the state depends on the previous sequence of values.

The normal approach for analysing sequences is by using a state table to identify the states the software can be in, and the response ("Action") to each input ("Event") in each state.

An example *State Diagram* is shown in Figure 2.6. The arrows represent *transitions* from one state to another. The annotation against each transition consists of an *event/action* pair. The *event* is the input event that causes the transition in that state, and the *action* is the output action or response that should take place during the transition. The expressions in square brackets – "[" and "]" – are called *guards*. A guard must be true for a transition to take place. The number N is referred to as a *state variable* which is used to prevent a *state explosion* if every possible different state has to be included. It is used when states differ only in the value of certain parameters, as shown in the example.



The state diagram shown in Figure 2.6 specifies the response of a counter with three defined methods: *inc*rement the counter, *dec*rement the counter, and *getValue*. The state diagram specifies that:

- The counter starts in the ZERO state with value 0
- In the ZERO state:
  - *dec* has no effect
  - *inc* moves the counter back into the NORMAL state with the value N set to 1
- In the NORMAL state, *inc* will increment the counter
  - *dec*, if $N > 1$ will decrement the counter
  - *dec*, if $N == 1$ will enter ZERO state with N set to 0

Note that a State Table can be used to show exactly the same information in a different form. A State Table has a row for each state, and a column for each event. The table cells show the next state and the action.

Analysis of a State Diagram includes identifying:

- All the transitions. Sometimes *implicit* transitions occur when the response to an event is not defined explicitly in the diagram – in this case the *implicit* transition is for the software to stay in the same state and for no action to take place.
- All the paths from entry to exit. In this diagram there is no exit state, so all the paths from entry to every state should be identified.
- All the circuits in the diagram that start and end in the same state.

State is important for much non-object oriented software, particularly for real-time/control systems and communications. This topic is addressed in more detail with Object-Oriented Testing in Chapter 6.

2.6 Analysis of Software Components

Software can be regarded as being constructed from a number of different types of component. The main types used in testing are described below.

2.6.1 Control Flow Graphs

A Control Flow Graph (CFG) is a directed graph showing the flow of control through a segment of code. They are mainly used at the source code level. A node in the graph represents one or more indivisible statements in the source code. Each edge represents a jump or branch in the flow of control. A node with two exits represents a block of code terminating in a decision, with true and false exits.

The CFG is language independent, and provides a simplified model of the source code. This allows both sequential blocks of code, and branches, to be easily identified.

CFG's for the basic programming constructs of sequences, selection (if), selection (if-then-else), iteration (while), and iteration (do-while) are shown in the following figures.

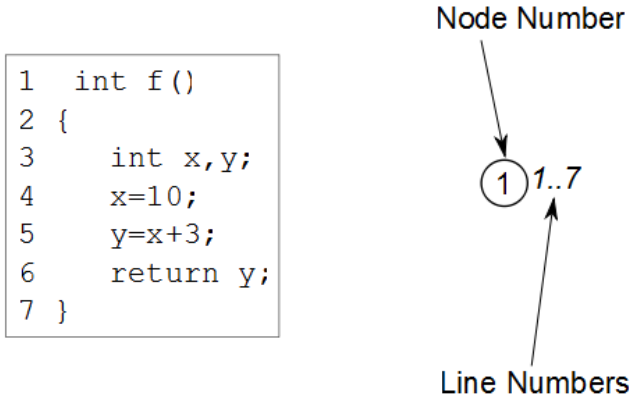


Figure 2.7: CFG for Sequence

In Figure 2.7, the code in lines 1–7 is always executed as a sequence, so are represented as a single node.

Note: there is no standard convention for including braces "{" and "}" and other non-executable source code notation, except to be consistent. The benefit of always including the braces is that it is much easier to review CFG's and ensure that no lines have been left out.

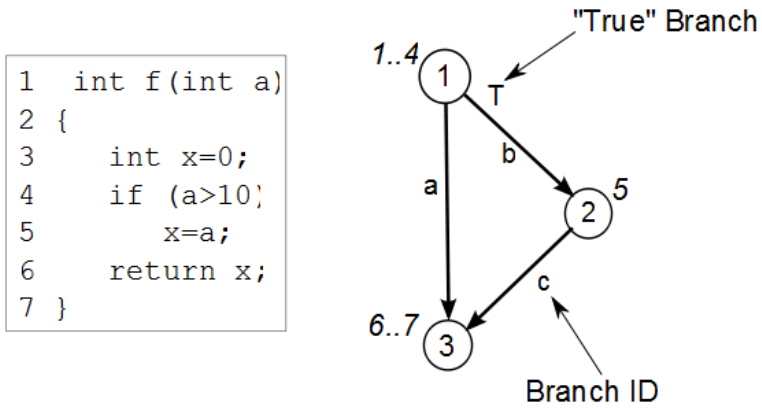


Figure 2.8: CFG for Selection (if-then)

In Figure 2.8, the code in lines 1–4 is always executed as a sequence. If the decision on line 4 evaluates to true, then line 5 is executed (node 2), and then control moves to line 6 (node 3). If the decision on line 4 evaluates to false, then control jumps straight to line 6 (node 4).

```
1  int f(int a)
2 {
3     int x=0;
4     if (a>10)
5        x=a;
6     else
7        x=10;
8     return x;
9 }
```

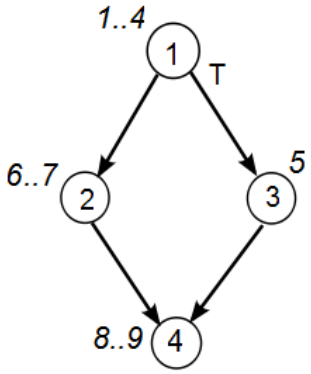


Figure 2.9: CFG for Selection (if-then-else)

In Figure 2.9, the code in lines 1–4 is always executed as a sequence. If the decision on line 4 evaluates to true, then line 5 is executed (node 2), and then control moves to line 8 (node 4). If the decision on line 4 evaluates to false, lines 6–7 are executed (node 3), and then control moves to lines 8–9 (node 4).

```
...
1  switch(a) {
2     case 0:
3        b=33;
4        break
5     case 1:
6        b=44;
7        break
8     default:
9        ok=false;
10       break;
11 }
...
```

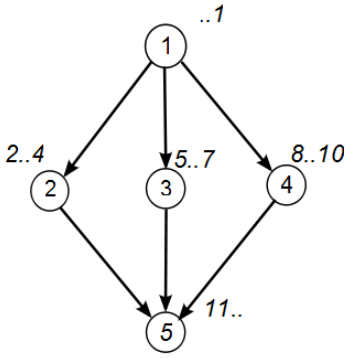


Figure 2.10: CFG for Selection (switch)

In Figure 2.10, the code ending on line 1 is executed as a sequence. Depending on the value of a, control then jumps to either line 2 (node 2), line 5 (node 3), or line 8 (node 4). Lines 2..4 execute as a sequence, and control moves to line 11 (node 5). Lines 5..7 execute as a sequence, and control moves to line 11 (node 5). And lines 8..10 execute as a sequence, and control moves to line 11 (node 5).

```
1 int f(int a)
2 {
3     int x=0;
4     while (a>0){
5         x++;
6         a--;
7     }
8     return x;
9 }
```

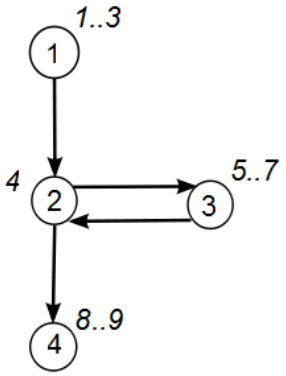


Figure 2.11: CFG for Iteration (while)

In Figure 2.11, the code in lines 1–3 is always executed as a sequence – note that the while statement must be in its own node as the decision is evaluated every time through the loop. If the decision on line 4 (node 2) evaluates to true, then lines 5–7 are executed (node 3), and then control returns to line 4 (node 2). If the decision on line 4 (node 2) evaluates to false, then control jumps to line 8 (node 4).

```
1 int f(int a)
2 {
3     int x=0;
4     do {
5         x++;
6         a--;
7     } while (a>0);
8     return x;
9 }
```

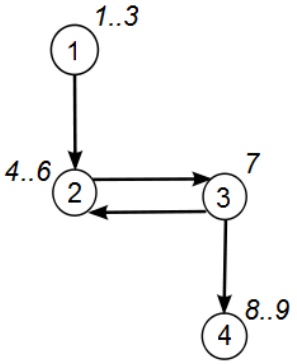


Figure 2.12: CFG for Iteration (do-while)

In Figure 2.12, the code in lines 1–3 is always executed as a sequence. The body of the do-while loop is then executed, lines 4–6 (node 2). Note the difference in structure from the previous example. If the decision on line 7 (node 3) evaluates to true, then control jump back to line 4 (node 2). If the decision on line 7 (node 3) evaluates to false, then control jumps to line 8 (node 4).

```
1 int f(int a)
2 {
3    int x=1;
4    for (int i=0; i<a; i++)
5       x=x*2;
6    return x;
7 }
```

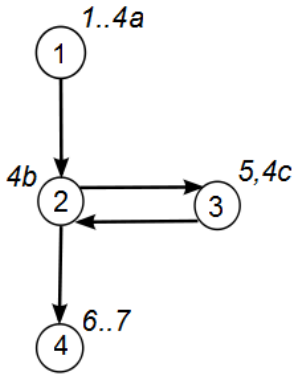


Figure 2.13: CFG for Iteration (for)

In Figure 2.13, the code in lines 1–3 followed by the initial for-loop assignment (line 4a) is always executed as a sequence (node 1). The decision in the for-loop, `i<a` on line 4b, is then evaluated (node 2). If true, then the body of the for-loop, on line 5, is then executed, followed by the for-loop increment on line 4c (node 3). When the decision evaluates to false, then lines 6..7 are executed (node 4)..

Larger CFGs are shown in the worked examples in Chapter 4.

2.6.2 Decisions and Conditions

CFGs represent the flow of control through a program. But many faults are not directly associated with the control flow itself, but rather with the decisions made in selecting the flow of control. There are two key components: *decisions* and *conditions*.

A *decision* is a compound boolean expression used (in Java) in the `if`, `for`, and `while` statements. A *condition* is a simple Boolean expression that can be used to make up a Decision[2]. A decision consists of one or more conditions. These are shown in example `boolean condIsNeg(int,boolean)` with the following source code:

```
1   boolean condIsNeg(int x, boolean flag) {
2      boolean rv=false;
3      if (flag && (x<0))
4         rv = true;
5      return rv;
6   }
```

Notes:

- Line 3 contains a decision: "`(flag && (x<0))`".
- This decision contains two conditions: "`flag`" and "`(x<0)`".

[2]A simple Boolean expression contains no Boolean operators

2.6.3 Paths

Instead of considering a program as being constructed from the individual statements (shown connected by branches in a CFG), a program can also be considered as being constructed from a number of paths from start to finish.

The only complexity is in handling loops: a path that makes i iterations through a loop is considered to be distinct from a path that makes $i + 1$ iterations through the loop, even if the same nodes in the CFG are visited in both iterations. Most programs, therefore, have a very large number of paths.

It is easiest to base the paths on the CFG – each path being represented by a sequence of nodes visited. The paths for `boolean condIsNeg(int,boolean)` are shown in Figure 2.14.

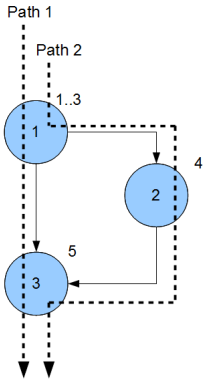


Figure 2.14: Paths for condIsNeg()

The paths for `boolean condIsNeg(int,boolean)` can be documented by listing the nodes executed for each path as follows:

1. Nodes: 1,3
2. Nodes: 1,2,3

To reduce the large number of different paths where loops are involved, equivalence classes of paths are selected. For the purposes of testing, two paths are considered equivalent if they differ only in the number of loop iterations. This gives two classes of loops:

- ones with 0 iterations
- ones with n iterations ($n >0$)

The number of start-to-finish paths, and the nodes on each path, can be calculated using the technique of *Basis Paths* as described in the following section.

Basis Paths

The flow of control through a CFG is represented as a regular expression, which is then evaluated to give the number of end-to-end paths. Loops can be executed an indefinite number of times: this is resolved by treating them as code segments that can be executed exactly zero or one times.

There are three operators defined for the regular expression:

. *concatenation* – this represents a sequence of nodes in the graph

\+ *selection* – this represents a decision in the graph (e.g. the `if` statement).

$()^*$ *iteration* – this represents a repitition in the graph (e.g. the `while` statement).

Consider the following example:

```
1   i=0;
2   while (i<list.length) {
3      if (list[i]==target)
4         match++;
5      else
6         mismatch++;
7      i++;
8   }
```

This is represented by the CFG in Figure 2.15.

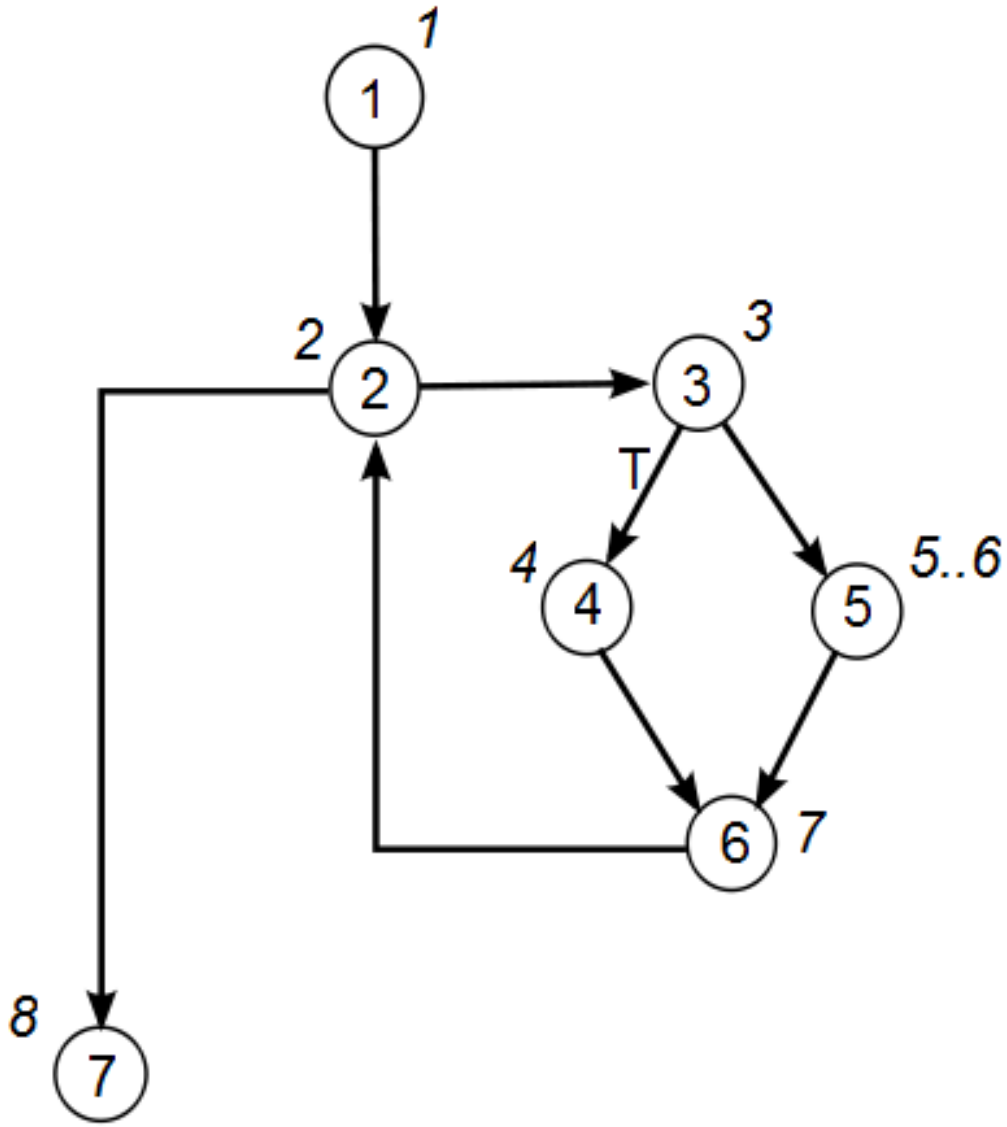


Figure 2.15: CFG for Basis Paths Example

This CFG can be represented by the following regular expression, using the operators described:

$$1 \cdot 2 \cdot (3 \cdot (4 + 5) \cdot 6 \cdot 2)^* \cdot 7$$

The loops are simplified by replacing $(expression)^*$ with $(expression + 0)$ representing the two equivalence paths, one with a single iteration through the loop *expression*, and one with no iterations, where 0 represents the *null* statement. This gives:

$$1 \cdot 2 \cdot ((3 \cdot (4 + 5) \cdot 6 \cdot 2) + 0) \cdot 7$$

This can be described as node 1 followed by 2 followed by either 3 followed by either 4 or 5, or null, followed by 7. Expanding this, where the + symbol represents alternatives, gives the following paths:

1. 1–2–7
2. 1–2–3–4–6–2–7
3. 1–2–3–5–6–2–7

By replacing each of the node numbers (and *nulls*) by the value 1, and then evaluating the expression as a mathematical formula (using + for addition, and · for multiplication), the number of paths can be calculated as follows:

$$paths = 1 \cdot 1 \cdot ((1 \cdot (1 + 1) \cdot 1 \cdot 1) + 1) \cdot 1 = 3$$

In practice the paths can often be identified by hand without using this technique.

Note: for "null-else" statements, where there is an *if* with no *else*, the expression $(n + 0)$ is used. The node n represents the true decision, and the node 0 represents the *null* else decision.

2.6.4 Data-Flow Testing

The previous analysis techniques are concerned with the flow of control through the program. An alternative is to analyse the flow of data through a program. A program accepts inputs, performs computations, assigns new values to variables, and returns results. Data values "flow" from one statement to another: a data value produced in one statement will be used by a later statement.

There are two possible uses for variables (parameters, local variables, and attributes) in a program: they can be assigned a new value (*definition*), and they can be referred to in an expression (*use*). Each possible path from the definition of a variable to its subsequent use is referred to as a d-u pair. Not all paths from every definition to every use are possible, so it is useful to refer to the pairs as *candidate* d-u pairs until this has been confirmed.

For each variable, every definition and every use of the variable in the method should be identified. Then every candidate d-u pair must be examined to see if there is a possible execution path from the definition to the use. If so, this d-u pair is added to the list. If not, this d-u pair is discarded.

Analysis of d-u pairs can identify problems with data referred to as *Data Flow Anomalies*. There are three types of anomaly:

1. Defined and then re-defined before use.
2. Used before defined.
3. Defined but not used subsequently.

2.6.5 Ranking

Not all testing techniques can be directly compared to each other – but a number of the White-Box techniques can, and this ranking is shown in Figure 2.16. The techniques nearer the top of the figure are said to *subsume* those lower down: this means that they guarantee to provide at least the same level of coverage as the techniques lower down.

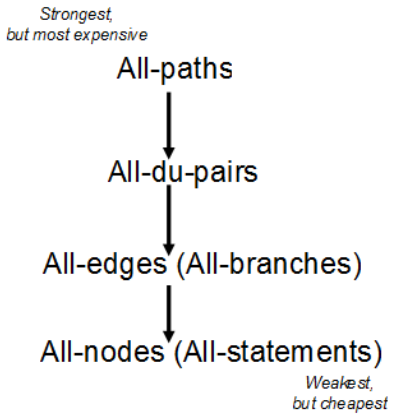


Figure 2.16: Coverage Criteria Ranking

2.7 Analysis of Targets for Fault Insertion

Inserting faults into a program requires analysing the source code to identify suitable places for this insertion – referred to as *fault targets*. Simple candidate targets are primarily based on potential *typos* – small typing mistakes. These are often referred to as *statement-level mutations*. At a higher level of sophistication, algorithms and software interfaces are suitable targets. And at a higher level again, for Object-Oriented programs, class relationships and class interactions are candidate targets, often referred to as *class-level mutations*. This book will only consider simple targets.

The most commonly used mutations are derived from original research into faults in Fortran programs. There are many variations – this book uses *Offutt's 5 Sufficient Mutations* to demonstrate the technique. In practice all Mutation Testing is automated, due to the very large number of possible mutations. Most tools are based on these sufficient mutation operators, along with additional options.

2.7.1 Offutt's 5 Sufficient Mutations

There are a very large number of possible minor faults that can be inserted into source code. Of these, the following five form a reasonable subset for practical testing purposes[3] which is addressed in this book:

ABS Absolute value: change a value x to be -ABS(x), ABS(x), and 0

AOR Arithmetic operator: change an operator from {+ - * / %} to be each of the other operators

LCR Logical operator: where *op* is the operator, change a logical expression (x *op* y) to be (x||y), (x&&y), (x), (y), (x *op* true),(x *op* false), (true *op* y), (false *op* y)

ROR Relational operator: where *cmp* is the operator, change a relational expression (x *cmp* y) to be x, (x<y), (x≤y), (x>y), (x≥y), (x! =y), (x==y), y, true, false

UOI Unary operator: insert each of {! - ++ −−} as a prefix operator before an expression, and both of {++ −−} after an expression as a postfix operator

Note that automated mutation testing tools typically use a much larger subset of possible mutations, at both the statement and the class level.

2.8 Test Artefacts

The test activity produces a number of interim outputs as shown in Figure 2.17. These reflect the significant effort required to design and implement tests.

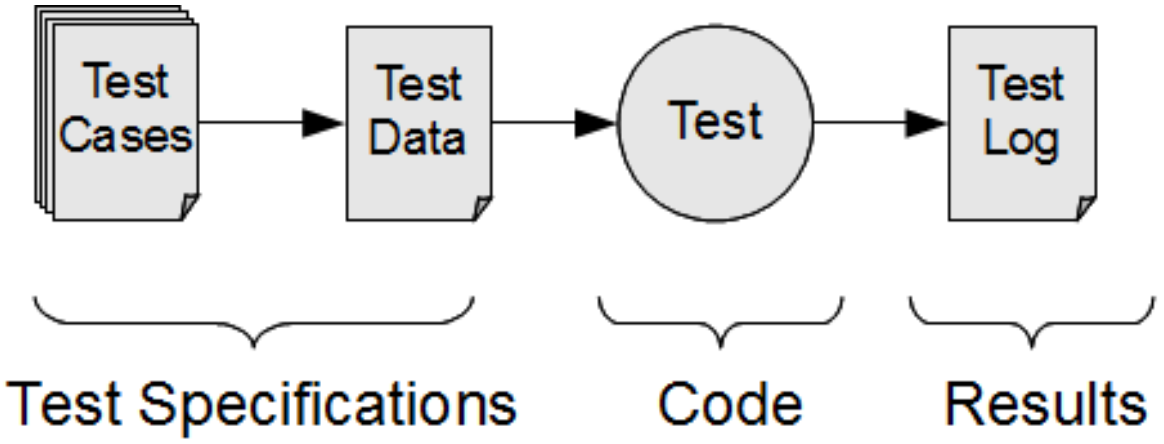


Figure 2.17: Test Components

Test specifications need to contain the following information:

- The identify of the System or Software Under Test (SUT) – this includes the name and version number

[3]Offutt, Mason, Rothermel, Untch and Zapf, *An Experimental Determination of Sufficient Mutant Operators*

- The test technique(s) being applied
- Results of the analysis (of the specification or of the software components)
- The Test Cases (these are the specific items that the test data must exercise)
- The Test Data (this includes the test identifiers, input and expected output values, and the Test Cases covered by each test)

In this book, Test Data is specified using the following template (see Table 2.6). There is no standard for this format – but it is important to be consistent to allow test specifications to be reviewed by other developers or testers. The "ID" column provides a unique Test ID for each test.

Table 2.6: Test Data Specification Template

ID	**Test Cases Covered**	**Inputs**	**Expected Output**
		(parameter names here)	(parameter names here)
(ID) etc	(List here)	(Values here)	(Values here)

Note that duplicate Test Data (in a previous test for the same SUT) should be identified at this stage. This provides a key mechanism to minimise the amount of work required to develop the tests, and also the length of time it takes to execute them. Tests are often organised in *test suites* for convenience; the same test can then be referenced from multiple suites.

Chapter 3

Unit Testing

In this chapter, specific testing techniques are described, and examples given of their application. Unit Testing is used as it provides the best environment for demonstrating the application of software testing principles.

3.1 Introduction

Within this chapter the following key test techniques are described:

Black-Box

1. Equivalence partitioning
2. Boundary value analysis
3. Combinational testing
4. Sequential testing
5. Random input data
6. Error guessing

White-Box

7. Statement coverage
8. Branch coverage
9. Condition coverage
10. Decision condition coverage
11. Multiple condition coverage

12 Path coverage

13 d-u pair coverage

Fault Insertion

14 Strong mutation testing

The next sections will present examples to demonstrate the application of all these techniques to a program. It will show how the test cases and test data are created and will then comment on the strengths and weaknesses of each approach. The black box techniques will be examined first and then followed by the white box techniques.

3.2 Usage

In general, the normal usage of Black-Box and White-Box testing techniques is as follows. Black-Box testing is used initially to verify that the software satisfies the specification:

- Use Equivalence Partitioning to verify the basic operation of the software
- If the specification contains boundary values, use Boundary Value Analysis to verify correct operation at the boundaries
- If specification states different processing for different combinations of inputs, use Combinational Testing to verify correct behaviour for each combination
- If the specification contains state-based behaviour, or different behaviour for different sequences of inputs, then use Sequential Testing to verify this behaviour
- If there are reasons to suspect that there are faults in the code, perhaps based on past experience, then use Error Guessing to try and expose them
- If the typical usage of the software is known, then use Random Test Data to verify the correct operation under these usage patterns

For each of these tests, measure the statement and branch coverage. Normally the goal is to achieve 100% statement coverage and 100% branch coverage – because these are easy to measure automatically. If this has not been achieved, then White-Box techniques can be used as follows:

- Use Statement Coverage to ensure 100% of statements have been executed
- Use Branch Coverage to ensure that 100% of the branches have been taken

Subsequently, if the code contains complex decisions, or if 100% branch coverage has not been achieved:

- Use Condition Coverage to ensure that every condition has been exercised
- Use Decision/Condition Coverage to ensure that every decision and every condition has been exercised
- Use Multiple Condition Coverage to ensure that every combination of conditions has been exercised

These White-Box test techniques can be further augmented as follows:

- If the code contains complex end-to-end paths, then use Path Testing to ensure coverage of these
- If the code contains complex data usage patterns, then use d-u Pair Testing to ensure coverage of these

In all cases, the decision to proceed with further tests is based on a *cost-benefit* tradeoff: balancing the extra time and work required to do the extra tests justified against the extra confidence they will provide in the software quality. Often it is a judgement call as to what level of testing to execute.

Fault Insertion can subsequently be used to measure the effectiveness of these tests in finding particular faults.

Notes:

- Black-Box tests can be written before, or in parallel with, the code (as they are based on the specifications).
- It normally serves no purpose to execute White-Box tests before Black-Box tests.
- White-Box testing can **never** be used as a substitute for Black-Box testing (see Section 2.2.1 for details).
- White-Box tests must be reviewed, and probably changed, every time the code is modified.

3.3 Black-Box Techniques

Black-Box Tests are characterised by how the input data is selected from the specification. For all Black-Box tests, testing is based on the specification, so there is no need to see the source code. This allows tests to be developed before the code, or in parallel with the code, giving more flexibility in planning a project than is the case for White-Box tests.

In general, the goal of each different technique is to provide coverage of a particular aspect of the specification.

3.3.1 Equivalence Partitioning (EP)

Description

Equivalence Partitioning is based on selecting representative values of each parameter from the equivalence partitions. Each equivalence partition for each of the parameters is a test case. Both the inputs and the output should be considered. The technique invariably involves generating as few tests as possible: each new test should select data from as many uncovered partitions as possible. Error cases should be treated separately to avoid *error hiding*.

The goal is to achieve 100% coverage of the equivalence partitions.

Test Cases

Each partition for each input and output is a Test Case. It is good practice to give each Test Case for each SUT a unique identifier. It is often useful to use the prefix "EP-" for Equivalence Partition test cases. Note that the values selected from the partitions are *not* the Test Cases, they are the Test Data.

Test Data

Input Test Data is selected, based on Test Cases which are not yet covered. Ideally, each normal Test will include as many additional normal Test Cases as possible. Each error Test must only include one error Test Case.

Expected output values are derived from the specification. However, the tester must ensure that all the Test Cases related to the output parameters are covered. It may be necessary to read the specification "backwards" to determine input values that will result in an output value being in the required equivalence partition.

Hint: it is usually easiest to identify Test Data by going through the Test Cases in order, selecting the next uncovered Test Case for each parameter, and then selecting an Equivalence Partition value. There is no reason to use different values from the same partition – in fact it is easier to review the Test Data for correctness if the one particular value is chosen from each partition, and then used throughout.

Comment

Equivalence Partitions provide a minimum level of black-box testing. At least one value has been tested from every input and output partition, using a minimum number of tests. These tests are likely to ensure that the basic data processing aspects of the code are correct. But they do not exercise the different decisions made in the code.

This is important, as decisions are a frequent source of mistakes in the code. These decisions generally reflect the boundaries of input partitions, or the identification of combinations of inputs requiring particular processing. These issues will be addressed in the following techniques.

Strengths

Provides a good basic level of testing.

Well suited to data processing applications where input variables may be easily identified and take on distinct values allowing easy partitioning.

Provides a structured means for identifying basic Test Cases.

Weaknesses

Correct processing at the edges of partitions is not tested.

Combinations of inputs are not tested.

The technique does not provide an algorithm for finding the partitions or selecting the test data.

3.3.2 Boundary Value Analysis (BVA)

Description

Programming faults are often related to the incorrect processing of boundary conditions, so an obvious extension to Equivalence Partitioning is to select two values from each partition: the bottom and the top values. This doubles the number of tests, but is more likely to find boundary-related programming faults. Each boundary value for each parameter is a test case. As for Equivalence Partitioning, the number of tests is minimised by selecting data that includes as many uncovered test cases as possible in each new test. Error tests are as always considered separately – only one error boundary value is included per error test.

The goal is to achieve 100% coverage of the boundary values.

Test Cases

Each boundary value for each partition for each input and output is a Test Case. It is good practice to give each Test Case for each SUT a unique identifier. It is often useful to use the prefix "BVA-" for Boundary Value Analysis test cases. Note that, unlike Equivalence Partitions, the values selected from the boundaries *are* the Test Cases.

Test Data

Input Test Data is selected, based on Test Cases which are not yet covered. Ideally, each normal Test will include as many additional normal Test Cases as possible. Each error Test must only include one error Test Case.

Expected output values are derived from the specification. However, the tester must ensure that all the Test Cases related to the output parameters are covered. It may be necessary to read the specification "backwards" to determine input values that will result in an output value having the required boundary value.

Hint: it is usually easiest to identify Test Data by going through the Test Cases in order, and selecting the next uncovered Boundary Value for each parameter.

Comment

There is little published evidence that using Boundary Values improves the effectiveness of testing, but experience indicates that this is likely to cover significantly more possible errors than Equivalence Partitions.

Note that Boundary Value Analysis provides exactly the same Test Cases as Equivalence Partitioning for boolean and enumerated parameters.

Strengths

Test Data values are provided by the technique.

Tests focus on areas where faults are more likely to be found.

Weaknesses

Combinations of inputs are not tested.

3.3.3 Testing Combinations of Inputs (CI)

Description

The previous two techniques reduce the number of tests by not considering combinations. Combinational testing, using *Truth Tables*, provides additional coverage by identifying as few tests as possible to cover all the possible combinations that have an impact on the output.

In many text books this technique is referred to as "Cause-Effect Graphing" and "Truth-Table Testing". We use the more general term of "Combinational Testing" as it best reflects the intent of testing combinations of parameter values, rather than a specific way to identify these combinations.

The goal is to achieve 100% coverage of the rules in the truth table.

Test Cases

Each rule from the truth-table is a Test Case. If an SUT includes multiple truth-tables (for example, in a class) then it is useful to give each Test Case a unique identifier.

Often a truth-table will only include normal cases, due to the number of rules required to describe all the possible error cases. If error cases are included, then it is often in a separate table for clarity.

Test Data

Input Test Data is selected, based on Test Cases (rules) which are not yet covered. Each Test will cover exactly one Test Case (or rule).

Expected output values are derived from the specification (the truth-table).

Hint: it is usually easiest to identify Test Data by going through the Test Cases (rules) in order, and selecting a value for each parameter that matches the required causes. As for equivalence partitions, it can be easier to review a test for correctness if as few different values as possible are used for each parameter. For expected output, the value from the specification must, if the technique has been followed properly, match the required effect.

Comment

The number of tests is reduced using 'don't-care" conditions where the value of a particular cause has no effect on the output. This means that Combinational Testing does not test *all* the combinations of causes.

Strengths

Exercises combinations of test data

Expected outputs are created as part of the process

Weaknesses

The truth tables can sometime be very large. The solution is to identify sub-problems, and develop separate tables for each.

Very dependent on the quality of the specification - more detail means more causes and effects, which takes more time to test; less detail means less causes and effects, but less effective testing

Testing all the possible combinations of Causes and Effects would cause a very large number of tests for a typical program. Picking a minimum number of rules, based on using "don't care" conditions reduces the number of tests significantly – at the cost of reducing the test coverage.

It should be noted that Equivalence Partitions/Boundary Values are complementary to Combinational Testing. There is again little published evidence as to the effectiveness of Truth Table testing, but experience in programming indicates that this is likely to cover different errors from Equivalence Partitions and Boundary Values.

3.3.4 Testing Sequences of Inputs (SI) or State

Description

Many software systems are state based – their behaviour is based not only on the current inputs, but also on their current state (essentially defined by the history of previous inputs). A particular state is reached based on the sequence of inputs to date.

The normal way to describe state behaviour is via a State Table or a State Diagram, showing all the states, the transitions between the states that different input events cause, and the actions to be taken on each state transition.

The goal in state-based testing is to exercise the software by providing inputs that should (according to the specification) cause it to transition through different states, and to then check that the corrects state transitions have taken place, and that the correct actions (or side-effects) have also taken place.

The key test design strategies for state-machines are:

- Piecewise (test some transitions)
- All transitions (test all transitions at least once)
- All *n-event* transition sequences (test all sequences of transitions for a particular value of n)
- All round-trip paths (starting and ending at the same state)
- M-length signature (testing all sequences of M transitions)
- Exhaustive testing (testing all possible sequences of transitions)

The goal is to achieve 100% coverage, based on the strategy above. For example, 100% of the transitions.

Test Cases

The Test Cases depend on the strategy used. For example, using *All Transitions*, each transition is a Test Case. For completeness, *implicit* transitions should be included – these are events that do not result in a change of state. Each Test Case should be a unique identifier – if the transitions are labeled on the state diagram, then this identifier should be based on the label.

There are no normal or error cases in a state diagram – every transition is "valid", though some may require error values in order to be executed. Note that if two different events cause the same state transition, then they should be treated as two different Test Cases.

Test Data

Input Test Data is selected, based on Test Cases (transitions) which are not yet covered.

Expected output values are derived from the specification.

Hint: it is usually easiest to identify Test Data by going through the Test Cases (rules) in order, and selecting a value for each parameter that matches the required causes.

State-based testing applies to all software that maintains state between calls. In this book, examples of state-based testing are addressed with Object-Oriented Testing (Chapter 6).

3.3.5 Random Input Data (RID)

Description

In the previous techniques, data has been selected to try and provide a certain amount of coverage of the specification. An alternative approach is to generate input data randomly.

Test data is generated using random number generators. The distribution may be uniform, or chosen to mimic, in a statistical sense, the type of inputs that the program will receive in real use. If the specification is clearly written and thorough, then it should be possible to find the set(s) of possible input values.

The goal is to achieve a "reasonable" coverage of the possible values for each input parameter, based on its distribution. This can be determined heuristically (using, for example, 10 random values), or based on a statistical sample size determined from the required confidence in the coverage.

Test Cases

Each Test Case is represented by a set of (random) input values, one for each parameter.

If the test is fully automated, then each Test Case is represented by a distribution of values for a particular parameter. This will normally include the upper and lower limits, and the distribution to be used between these limits to select a random value.

Each Test Case should be given a unique identifier.

Test Data

Input Test Data is selected, normally automatically, based on the Test Cases.

Expected output values are derived from the specification. This may be manual or automated. Automating the interpretation of a specification to produce the expected output is difficult. Two approaches are to write a *Test Oracle* in a higher-level language, or to use *post-conditions* to determine the validity of an output after it is produced (rather than producing the expected output value before the test is run).

Comment

Random Test Data generation is straightforward to implement and leads to a fast generation of Test Cases. However, calculating the Expected Output from the specification is as time consuming as for EP and BV.

If the distribution/histogram of the real-world input data is known, then this provides a mathematical basis for selecting a set of input test case values. The measured test failure rate then provides an indication of the expected failure rate in use.

However, if the distribution is not known, then the basis for choosing the random data may not reflect its use, and the failure rate cannot be predicted from the results.

Furthermore, the random input data obtained may not have a sufficient set of illegal or extreme values, or even combinations of valid values, that will test the program thoroughly.

Strengths

Fast generation of test cases

Can offer a mathematical basis for selecting an appropriate set of input values

Weaknesses

Possibly an insufficient set of extreme or illegal values may be tested

If the distribution of the input is unknown the input values chosen may not reflect typical usage

Note: randomisation can also be used to select a small set of tests from a large suite in order to execute the suite more quickly. This can be particularly useful for regression testing. The effectiveness of this depends on how the random tests are selected. More sophisticated techniques may be "directed", using feedback from each test to select data for the following test.

Random data selection is sometimes used for *stability testing*, to ensure that no input data value causes the software to crash or raise unexpected exceptions. This technique is easy to implement in an automated manner, but is unlikely to find faults except in low-quality code.

3.3.6 Error Guessing (EG)

This is an *ad-hoc* approach, based on intuition and experience. Test data is selected that is likely to expose faults in the code. Some typical examples of inputs likely to cause problems are:

- Empty or null strings, arrays, lists, and class references. These may find code that does not check for empty or non-null values before using them.
- Zero as a value, or as a count of instances or occurrences. These may find divide-by-zero faults.
- Spaces or null characters in strings. This may find code that does not process strings correctly, or does not trim *whitespace* before trying to extract data from the string.
- Negative numbers. These may find faults in code that only expects to receive positive numbers.

The goal is to cover as many values as possible which in the experience of the tester are likely to expose faults in the code.

Test Cases

The tester selects values which are likely to produce errors. Each value is a Test Case. Each Test Case should have a unique identifier.

This technique can produce both normal and error Test Cases. The values selected are those that are likely to expose faults in the code, they are not necessarily illegal values.

Test Data

Input Test Data is selected, based on Test Cases which are not yet covered. As for other test techniques, error cases should be execued individually.

Expected output values are derived from the specification. It may be required to read the specification "backwards" to determine input values for output parameter Test Cases.

Comment

With experienced testers, this can be a very effective complement to other testing techniques. It depends on how well the testers know the types of mistakes that the developers are likely to make, or mistakes that have a high impact on the final product.

Strengths

Intuition can frequently provide an accurate basis for finding faults.

The technique is very efficient, as it focuses on likely faults.

Weaknesses

The technique relies on experienced testers – but they are not always available.

The *ad-hoc* nature of the approach means it is hard to ensure completeness of the testing.

3.4 White-Box Techniques

White-Box Tests are characterised by how the input data is selected from analysis of components that make up the code. These components may be simple, such as the statements in the program, or they may be more complex, such as the executable paths from start to finish through a program.

It is important to note that White-Box testing is usually used after Black-Box testing in order to improve coverage of the internal components that form the program. This means that many Test Cases will already have been executed during Black-Box testing. These should *not* be duplicated – instead, the tester should rather indicate at least one of the Black-Box tests that covers the Test Case.

In general, the goal of each different technique is to provide coverage of a particular aspect of the implementation.

3.4.1 Statement Coverage (SC)

Description

The goal of *Statement Coverage* is to make sure that every line of code has been executed during testing. The ideal test completion criteria is 100% statement coverage.

First of all, draw a CFG for the program. Each node on the CFG, representing a sequence of indivisible source code statements, is a test case. Then working from the CFG, and the source code, work out input values required to ensure that every source code statement is executed. Finally, using the specification – NOT the source code – work out the correct output for each set of input values.

Test Cases

Each statement in the source code is a Test Case. Normally, a single line of source is regarded as being a statement. Using a CFG the number of Test Cases is minimised: each node in the CFG is a Test Case. It is usual to give each Test Case a unique identifier based on the node numbers in the CFG (e.g. SC-1 or TC-SC-1 for node 1).

Test Data

Test data is selected to ensure that every statement, or node in the CFG, is executed. This selection requires the tester to review the code, and select input parameter values that cause the required statements to be executed. This requires careful analysis of the decisions in the CFG.

The expected outputs *must* come from the specification. It is very easy to make the mistake of reading the expected output from the code – and the tester must be careful to avoid doing this.

Hint: it is usually easiest to start with a simple path through the CFG for the first test. Then, by reading the code carefully, work out what required changes to the input values are required to cause a different path, with as many different nodes as possible, to be executed.

Comment

The level of statement coverage achieved during testing is usually shown as a percentage – for example, 100% statement coverage means that every statement has been executed, 50% means that only half the statements have been covered. Normally, each line of code is regarded as a statement.

Statement Coverage is the weakest form of White-Box testing. 100% statement coverage will probably not cover all the logic of a program, or cause all the possible output values to be generated.

Note that the analysis phase of Statement Coverage testing allows *unreachable* code to be identified.

Strengths

Provides a minimum level of coverage by executing all statements at least once. There is a significant risk in releasing software before every statement has been executed at least once during testing

Statement Coverage can generally be achieved using only a small number of tests

Weaknesses

Can be difficult to determine the required input parameter values

May be hard to test code that can only be executed in dangerous circumstances

Does not provide coverage for "NULL else" conditions. For example, in the code:

```
if (number < 3) number++;
```

Statement Coverage does not force a test case for number$\geq$3

Not demanding of compound logical conditions. For example, in the code:

```
if ( (a>1) || (b=0) ) then x = x/a;
```

The test case "(a=2, b=0)" executes both the `if` statement and the following assignment statement, but it does not cover the case where the assignment statement is not executed.

Note that Statement Coverage is often used as a supplementary measure of Black-Box Testing – mainly because it is easy to measure automatically. If Black-Box testing does not result in the required coverage – normally 100% – then this White-Box technique can be used to increase the coverage to the required level.

3.4.2 Branch Coverage (BC)

Description

The goal of *Branch Coverage* is to make sure that every branch in the source code has been taken during testing. The ideal test completion criteria is 100% branch coverage.

First of all, draw a CFG for the program. Then working from the CFG, and the source code, work out input values required to ensure that every source code branch is taken. Finally, using the specification – NOT the source code – work out the correct output for each set of input values.

Test Cases

Each branch in the source code is a Test Case. Using a CFG the number of Test Cases is minimised: each edge in the CFG is a Test Case. It is usual to give each Test Case a unique identifier based on the edge labels in the CFG (e.g. BC-a or TC-BC-a for edge a).

Test Data

Test data is selected to ensure that every branch, or edge in the CFG, is executed. This selection requires the tester to review the decisions in the code, and select input parameter values that cause the necessary outcomes from each decision.

The expected outputs *must* come from the specification. It is very easy to make the mistake of reading the expected output from the code – and the tester must be careful to avoid doing this.

Hint: it is usually easiest to start with a simple path through the CFG for the first test. Then, by reading the code and CFG carefully, work out what required changes to the input values are required to cause a different path, with as many different edges as possible, to be executed.

Comment

Branch coverage ensures that each output path from each "decision" is tested at least once. Note that 100% branch coverage guarantees 100% statement coverage – but the test data is harder to generate. Branch Coverage is a stronger form of testing than Statement Coverage, but it still does not exercise either all the reasons for taking each branch, or combinations of different branches taken.

Strengths

Resolves the problem with "NULL else" conditions.

Weaknesses

Can be difficult to determine the required input parameter values

Undemanding of compound decisions – it only requires that each edge be followed, not that every reason for following each edge be tested.

In advanced testing, exception handling is included in Branch Coverage. Each exception raised and caught is regarded as a branch.

Branch Coverage ensures that *every* edge in the CFG has been executed. *Decision Coverage* is an equivalent technique, that has different Test Cases, but results in the same outcome. To achieve Decision Coverage, every decision must take on the values true and false. The Test Cases are therefore that each edge be taken out of each node with multiple outgoing edges. Decision Coverage ensures that every decision is covered for both its true and false outcomes.

Note that Branch Coverage, like Statement Coverage, is often used as a supplementary measure of Black-Box Testing – mainly because it is easy to measure automatically. If Black-Box testing does not result in the required coverage – normally 100% – then this White-Box technique can be used to increase the coverage to the required level.

3.4.3 Condition Coverage (CC)

Description

A complex decision is formed from multiple (Boolean) conditions. Condition Coverage extends Branch Coverage by ensuring that, for complex decisions, each condition within the decision is tested for its true and false values. Note that: it is not necessary that the decision itself take on true and false values!

The goal of Condition Coverage is for every condition in every decision to take on the value True and False.

Test Cases

Each condition in each decision in the source code has two Test Cases – for true and false outcomes. There is no need to consider simple decisions, unless they have not already been covered. Each Test Case should be given a unique identifier (for example: CC-1-1-T and CC-1-1-F for condition 1 of decision 1 in the source code).

Test Data

Test data is selected to ensure that every condition in every decision takes on the value true and false. This selection requires the tester to review the complex decisions and the conditions in the code, and select input parameter values that cause the necessary outcomes from each condition.

The expected outputs *must* come from the specification. It is very easy to make the mistake of reading the expected output from the code – and the tester must be careful to avoid doing this.

Hint: it is usually easiest to start with the first decision in the source code. Then, by reading the code carefully, work out what input values are required to cause all the conditions to take on the values true and false. Then examine the source code and see what other conditions (Test Cases) have been caused by these values. Then determine what changes to the input values are required to cause any uncovered conditions to take on the values true and false. It may take time to work out the particular inputs required to minimise the number of tests to do this.

Comment

Note that Condition Coverage does not guarantee Branch Coverage! In order to achieve Branch Coverage, every Decision must be taken, as well as every Condition caused.

Strengths

Does focus on condition outcomes

Weaknesses

Can be difficult to determine the required input parameter values

Does not always achieve branch coverage, as shown in Figure 3.1), for the following test cases:

- a=true, b=false
- a=false, b=true

Note that each condition (a and b) has taken on the values true and false, but the decision (on line 1) always evaluates to false.

```
01    if (a&&b)
02        f1();
03    else
04        f2();
```

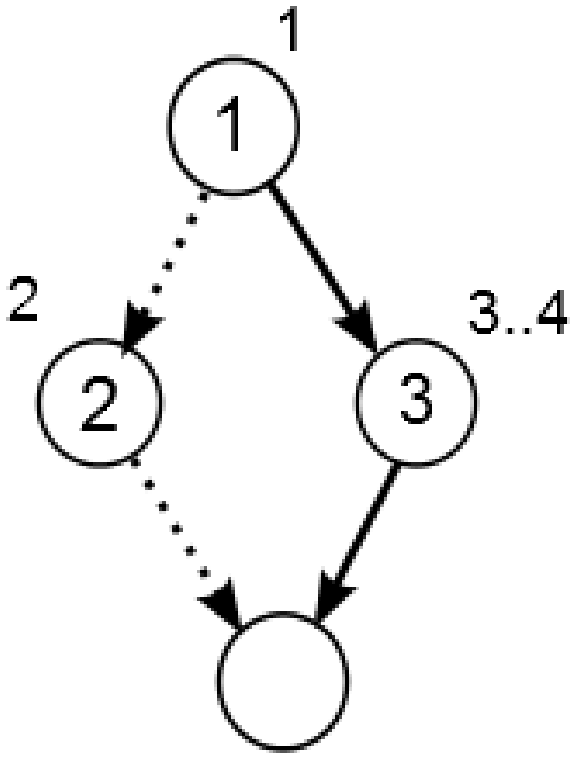


Figure 3.1: CC without BC

3.4.4 Decision Condition Coverage (DCC)

Description

Tests are generated to cause every decision to be taken at least once (branch coverage), and also every condition to be true and false at least once (Condition Coverage).
The goal is to achieve 100% coverage of every decision and 100% coverage of every condition.

Test Cases

Each decision has two test cases (for true and false outcomes), and every condition in each decision has two Test Cases (for true and false outcomes). There is no need to consider simple decisions, unless they have not already been covered. Each Test Case should be given a unique identifier (for example: DCC-1-T and DCC-1-F for decision 1

in the source code, and DCC-1-1-T and DCC-1-1-F for condition 1 of decision 1).

Test Data

Test data is selected to ensure that every decision takes on the value true and false, and that every condition in every decision takes on the value true and false. This selection requires the tester to review the complex decisions and the conditions in the code, and select input parameter values that cause the necessary outcomes from each decision and condition.

The expected outputs *must* come from the specification. It is very easy to make the mistake of reading the expected output from the code – and the tester must be careful to avoid doing this.

Hint: it is usually easiest to start with the first decision in the source code. Then, by reading the code carefully, work out what input values are required to cause all the conditions to take on the values true and false. And if the decision has not taken on both values, what extra inputs are required for it to do so. Then examine the source code and see what other decision and conditions (Test Cases) have been caused by these values. Then determine what changes to the input values are required to cause any uncovered decision and conditions to take on the values true and false. It may take time to work out the particular inputs required to minimise the number of tests to do this.

Comment

The true and false outcomes of every decision and every condition are covered.

Strengths

Stronger coverage that just Condition Coverage or Decision Coverage.

Weaknesses

Even though every decision is tested, and every condition is tested, not every possible *combination* of conditions is tested.

Can be difficult to determine the required input parameter values.

3.4.5 Multiple Condition Coverage (MCC)

Description

Tests are generated to cause every possible combination of conditions for every decision to be tested.

The goal is to achieve 100% coverage of every decision and 100% coverage of every condition.

A *Truth-Table* is the best way to identify all the possible combinations of values.

Test Cases

Each decision with n conditions has 2^n test cases (assuming the conditions are independent) – one for each combination of conditions.

Each Test Case should be given a unique identifier (for example: MCC-1-1, MCC-1-2, etc for combinations 1 and 2 of the conditions in decision 1 in the source code). It is often useful to draw up a table showing all the possible combinations and showing the identifier for each combination.

Test Data

Test data is selected to ensure that every combination of conditions in every decision is covered. This selection requires the tester to review the complex decisions and the conditions in the code, and select input parameter values that cause the necessary combinations.

The expected outputs *must* come from the specification. It is very easy to make the mistake of reading the expected output from the code – and the tester must be careful to avoid doing this.

Hint: it is usually easiest to start with the first decision in the source code. Then, by reading the code carefully, work out what input values are required to cause all the combinations of conditions to occur. Then examine the source code and see what other combinations of conditions in other decisions (Test Cases) have been caused by these values. Then determine what changes to the input values are required to cause any uncovered combinations of conditions to occur. It may take time to work out the particular inputs required to minimise the number of tests to do this.

Comment

Note that not all combinations of conditions are always possible. Even though multiple condition testing covers every possible combination of conditions in a decision, it does not cause every possible combination of decisions to be taken.

Strengths

Tests all possible combinations of conditions in every decision

Weaknesses

Can be expensive: n conditions in a decision give 2^n test cases

Can be difficult to determine the required input parameter values

3.4.6 Path Coverage (PC)

Description

Path testing causes every possible path from entry to exit of the program to be taken during test execution. By inspection of the code, input data is selected to cause execution of every path. Expected outputs are derived from the specification.

The goal is to achieve 100% coverage of every start-to-finish path in the code.

Test Cases

Each unique path from start to finish is a Test Case. Remember that the standard technique is for each loop to be in two different paths: one with zero times through the loop, and one with at least one time through the loop.

Each Test Case should be given a unique identifier (for example: PC-1 for path 1). It is useful, in order to ensure that no paths are missed, to list the edges in the CFG covered by each path.

Test Data

Test data is selected to ensure that every path is followed. This selection requires the tester to review the code carefully, and select input parameter values that cause these paths to be executed.

The expected outputs *must* come from the specification. It is very easy to make the mistake of reading the expected output from the code – and the tester must be careful to avoid doing this.

Hint: it is usually easiest to begin with the simplest path from entry to exit, and by reading the source code work out the required input values required to cause it to be executed. Then examine the source code and see what changes to the inputs are required to cause similar paths to be executed. There should be one test per path.

Comment

Path testing matches the flow of control through a program, but can be difficult to realize for complex programs – some paths may not be possible.

Strengths

Does create combinations of paths not exercised using other methods

Weaknesses

Can be computationally intensive if the program is complex and many paths are found

Does not explicitly evaluate the conditions in each decision

An alternative approach, which may generate fewer tests, is to generate tests that match the flow of data through a program, as shown in the following section.

3.4.7 d-u pair Coverage (DUP)

Description

The principle in d-u pair testing is to execute each path between the definition of the value in a variable and its subsequent use. The motivation is to verify the correctness of a program with all the possible flows of data through the program being covered.

The goal is to achieve 100% coverage of every possible du-pair in the code.

Test Cases

Every possible d-u pair is a Test Case. Some d-u pairs will be impossible to execute – these are best handled as "candidate" d-u pairs, and *not* assigned a Test Case.

Each Test Case should be given a unique identifier (for example: DU-1-1 for pair 1 for variable 1, or DU-x-1 where x is the name of the variable). It is essential to list the candidate d-u pairs in the code, and then give Test Case identifiers to the possible ones. As the Test Data is worked out, this table may need updating.

Test Data

Test data is selected to ensure that every d-u pair is executed. This selection requires the tester to review the code carefully, and select input parameter values that cause the path from every definition to every use to be followed.

The expected outputs *must* come from the specification. It is very easy to make the mistake of reading the expected output from the code – and the tester must be careful to avoid doing this.

Hint: it is usually easiest to start with any combination of inputs, follow the path taken, and list the d-u pairs covered. Then examine the source code and see what changes to the inputs are required to cause uncovered d-u pairs to be executed. Cover as many d-u pairs as possible in each test.

Comment

The d-u pair testing technique provides comprehensive testing of all the Definition-Use paths in a program, but generating the test data is a very time consuming exercise.

Strengths

This is a strong form of testing

Generates test data in the pattern that data is manipulated in the program rather than following abstract branches

Weaknesses

Number of test cases is very large, can be up to $\sum_{i=1}^{N} d_i \cdot u_i$, where d_i is the number of definitions for variable i, u_i is the number of uses, and N is the number of variables (this includes arguments, local variables, and class attributes)

With pointer variables can be difficult to determine which variable is being referenced

With arrays it is difficult to determine which element in the array is being referenced (the solution often used is to treat the entire array as a single variable)

3.5 Fault Insertion

3.5.1 Strong Mutation Testing (SMT)

Description

Mutation testing is a form of "fault-insertion" testing. In Strong Mutation Testing faults are inserted into "mutations" of the source code, and the tests (developed using one of the previous techniques) are run against these.

The tests should fail for every mutation.

This form of testing can be used (a) to show that the particular faults inserted do NOT exist in the code, and/or (b) to verify the validity of the test cases created (by ensuring that they fail on all the inserted faults).

There are many types of program mutations that can be made. To simplify things a simple set of five categories is often used: ABS, AOR, LCR, ROR, and UOI (see Section 2.7.1 for details).

The goal of mutation testing is to achieve 100% coverage of the possible mutations identified.

Comment

The effectiveness of mutation testing depends on how well the selected mutation operators reflect actual patterns of faults in the code. Ideally, the operators should be picked based on feedback from actual faults. However, it is unlikely that more abstract faults can be artificially generated, so the focus is on single-change faults.

Strengths

It shows the absence of particular faults.

It provides an assessment of the quality of the test data.

Weaknesses

Computationally expensive as a huge number of mutants can be generated.

Automation can remove much of the manual work in mutation testing.

Weak Mutation Testing involves making the same mutations, but manually reviewing the code to see if the mutations would produce a different result.

Chapter 4

Unit Testing Examples

In this chapter, two examples are used to illustrate the application of the different test techniques described in the previous chapter. Example 1 ("seatsAvailable") is based on calculating whether the required number of seats are available for an airline reservation system. Example 2 ("premium()") is based on calculating a car insurance premium.

Note that, for completeness, each test is treated independently. In practice, all the tests for a particular SUT would be treated together to avoid redundancy. If a new test were identified that repeats the values used for a previous test, then that new test would not be created, and the Test Case would be documented as being covered by that previous test. This would also require the use of extended identifiers for the Test Cases and Tests, as discussed in the previous chapter (for clarity, these have not been used).

4.1 Example One: seatsAvailable()

This example is based on an airline reservation system. It uses a function that calculates whether there are a certain number of seats free.

4.1.1 Description

A program for airline seat reservation takes two inputs. The first is the number of free seats and the second is the number of seats required. Both numbers are integers. If the number of free seats entered is less than zero, or the number of seats required is less than one, then the program return variable is false. If the number of seats required is less than or equal to the number of free seats, then the return variable is true. Otherwise, the return variable will be false.

4.1.2 Specification

Program Inputs

freeSeats: 0..totalSeats

seatsRequired: 1..totalSeats

Program Outputs

return value:

false if any inputs are invalid
true if seatsRequired≤freeSeats
false if seatsRequired>freeSeats

Notes: totalSeats is a constant (with value 100)

4.1.3 Source Code

The source code for `seatsAvailable()` is as follows:

```
1   public static boolean seatsAvailable(int freeSeats,
                                         int seatsRequired)
2   {
3      boolean rv=false;
4      if ( (freeSeats>=0) && (seatsRequired>=1)
                           && (seatsRequired<=freeSeats) )
5         rv=true
6      return rv;
7   }
```

4.1.4 Equivalence Partitioning

Reminder: an equivalence partition is a range of values for a parameter for which the specification states the same processing.

In this example there are three types of processing defined:

- success: enough seats are available to meet the request
- failure: enough seats are not available to meet the request
- failure: an error in one or more of the input parameters

Partitions

The partitions for the input parameters to seatsAvailable() are shown in Table 4.1.

Table 4.1: Input Partitions for seatsAvailable()

Parameter	Range
freeSeats	INT_MIN..-1 0..seatsRequired-1 seatsRequired..totalSeats(see note 1) totalSeats+1.. INT_MAX
seatsRequired	INT_MIN..0 1..freeSeats freeSeats+1..totalSeats (see note 2) totalSeats+1..INT_MAX

Notes:

1. If seatsRequired==totalSeats, then freeSeats has a single value partition at totalSeats.
2. If freeSeats==totalSeats-1, then seatsRequired has a single value partition at totalSeats. If freeSeats==totalSeats, then this partition does not exist.
3. INT_MIN is the minimum possible integer value (Integer.MIN_VALUE in Java).
4. INT_MAX is the maximum possible integer value (Integer.MAX_VALUE in Java).
5. totalSeats is not a parameter to this function: it is a constant. It is best to keep your test designs as generic as possible, avoiding actual data values until they are needed. The values, for example, might be language specific. Values for the edges of partitions are expressed in terms of the constant totalSeats.

The partitions for the output from seatsAvailable() are shown in Table 4.2.

Table 4.2: Output Partitions for seatsAvailable()

Parameter	Range
Return Value	true false

Test Cases

Each partition is a test case, as numbered below. In practice, the partitions and test cases can be identified in a single table.

Table 4.3: Test Cases for seatsAvailable()

Case	Parameter	Range	Test
1*	freeSeats	INT_MIN..-1	3
2		0..seatsRequired-1	1
3		seatsRequired..totalSeats	2
4*		totalSeats+1..INT_MAX	4
5*	seatsRequired	INT_MIN..0	5
6		1..freeSeats	2
7		freeSeats+1..totalSeats	1
8*		totalSeats+1..INT_MAX	6
9	Return Value	true	2
10		false	1

Notes:

1. * indicates an error case.
2. totalSeats is not a parameter to this function: it is a constant. In order to generate test data, a value must be specified. If the value is not known, this can be done by specifying test values as relationships to totalSeats. In this case, the value is specified, and so the values can be specified as actual values.
3. The "Test" column indicates one or more tests that cover this Test Case. This is filled in after the Test Data has been completed (see next section). This provides quick confirmation that every Test Case has been covered in a test.

Test Data

The data for each test should include a unique identifier (here they are numbered), the test cases covered, the input values, and the expected values.

Test input data is selected from arbitrary values within the equivalence partitions. For each additional test, data is selected to cover as many additional normal test cases as possible. Each error case must have its own unique test – there can only be one error case covered by any one test.

Table 4.4: EP Tests for seatsAvailable()

ID	Test Cases Covered	Inputs		Expected Output
		freeSeats	seatsRequired	return value
1	2,7,10	50	75	false
2	3,6,9	50	25	true
3	1*	-100	25	false
4	4*	200	25	false
5	5*	50	-100	false
6	8*	50	200	false

Notes:

1. Reviewing the Test Cases covered column allows us to confirm that all the test cases have been covered by these tests.
2. Each error case must be tested separately – otherwise multiple simultaneous errors may hide correct error processing for one of the errors.

It is important to include the Test Cases Covered column. This allows the tester to confirm that all the test cases have been covered, and also makes it easy for a test auditor to review the tests for completeness.

4.1.5 Boundary Value Analysis

Reminder: Boundary Values are the upper and lower values for each Equivalence Partition.

Boundary Values and Test Cases

Each boundary value is a test case. The input boundary values and test cases are shown in Table 4.5.

Table 4.5: Input Boundary Values for seatsAvailable()

Case	Parameter	Boundary Value	Test
1*	freeSeats	INT_MIN	11
2*		-1	12
3		0	7
4		seatsRequired-1	8
5		seatsRequired	9
6		totalSeats	10
7*		totalSeats+1	13
8*		INT_MAX	14
9*	seatsRequired	INT_MIN	15
10*		0	16
11		1	7
12		freeSeats	9
13		freeSeats+1	8
14		totalSeats	10
15*		totalSeats+1	17
16*		INT_MAX	18

Notes:

1. * indicates an “error” case – each error case must be tested separately

And the output boundary values are shown in Table 4.6.

Table 4.6: Output Boundary Values for seatsAvailable()

Case	Parameter	Boundary Value	Test
17	Return Value	true	9
18		false	7

Test Data

Test input data is selected from the boundary values. For each additional test, data is selected to cover as many additional normal test cases as possible. Each error case must have its own unique test – there can only be one error case covered by any one test.

Table 4.7: BV Tests for seatsAvailable()

ID	Test Cases Covered	Inputs		Expected Output
		freeSeats	seatsRequired	return value
7	3,11,18	0	1	false
8	4,13,18	49	50	false
9	5,12,17	50	50	true
10	6,12,13,14,17	100	100	true
11	1*	INT_MIN	50	false
12	2*	-1	50	false
13	7*	101	50	false
14	8*	INT_MAX	50	false
15	9*	50	INT_MIN	false
16	10*	50	0	false
17	15*	50	101	false
18	16*	50	INT_MAX	false

Notes:

1. by definition, Boundary Value Analysis covers all the Equivalence Partition test cases.

At the expense of approximately twice the number of tests, the minimum and maximum value of each Equivalence Partition has been tested at least once, using a minimum number of tests. However, combinations of different boundary values have not been exhaustively tested: in this example there is a limit of 128 candidate combinations (not all are possible).

4.1.6 Combinational Testing

Summary

Reminder: a Truth Table is used to identify all the possible combinations of inputs that produce different output (rules).

Causes

The number of causes should be minimized to reduce the size of the Truth Table – in particular, where a parameter has a range of values that provide a particular response, this can be expressed as a single cause.

The causes for this program, taken from the specification, can be expressed as follows:

1. $0 \leq$ freeSeats $\leq$ totalSeats
2. $1 \leq$ seatsRequired $\leq$ totalSeats
3. seatsRequired $\leq$ freeSeats

Effects

- return value == true

Truth Table

Table 4.8: Truth Table for seatsAvailable()

	Rules			
	1	2	3	4
Causes				
$0 \leq$ freeSeats $\leq$ totalSeats	F	*	T	T
$1 \leq$ seatsRequired $\leq$ totalSeats	*	F	T	T
seatsRequired $\leq$ freeSeats	*	*	F	T
Effects				
return value	F	F	F	T
	19	20	21	22
	Tests			

Test Cases

Each Rule is a Test Case. The "Tests" entries are filled after the Test Data has been determined.

If the Truth Table covers error cases, then each error rule is a test case also.

4.1.7 Tests

Each Test Case must be covered in a separate test – it is not possible to have multiple combinations in the same test. Test input data is selected by picking values that satisfy the Causes and Effects for a Rule.

Each error test case has its own unique test.

Table 4.9: TT Tests for seatsAvailable()

ID	Test Cases Covered	Inputs		Expected Output
		freeSeats	seatsRequired	return value
19	1	-100	50	false
20	2	50	150	false
21	3	50	75	false
22	4	50	25	true

4.1.8 Using Random Test Data

Reminder: input data is selected at random; expected outputs are calculated from the specification.

Test Data

There is no information in the specification for this problem as to the statistical distribution of the input, so a Discrete Uniform probability distribution will be used (every random value is equally likely – as provided by the standard random number generator in Java).

Treating freeSeats and seatsRequired as Discrete Uniform random variables whose values can range from –totalSeats up to totalSeats, a number of random values for each input parameter can be generated. In this example, 6 random values are selected.

Table 4.10: Random Test Cases for seatsAvailable()

Case	**Parameter**	**Random Value**	**Test**
1	freeSeats seatsRequired	-23 13	23
2	freeSeats seatsRequired	71 -67	24
3	freeSeats seatsRequired	-83 -7	25
4	freeSeats seatsRequired	73 55	26
5	freeSeats seatsRequired	-43 27	27
6	freeSeats seatsRequired	-41 24	28

Test Data

The expected output values are derived from the specification for the randomly selected input data. The input values are generated automatically using random number generators with the correct distribution of values, as shown in the Test Cases.

Table 4.11: Random Data Tests for seatsAvailable()

ID	Test Cases Covered	Inputs		Expected Output
		freeSeats	seatsRequired	return value
23	1	-23	13	false
24	2	71	-67	false
25	3	-83	-7	false
26	4	73	55	true
27	5	-43	27	false
28	6	-41	24	false

4.1.9 Statement Coverage

Reminder: run tests that ensure that every statement of source code is executed.

CFG

Develop the CFG from the source code, clearly labeling all the nodes and edges, and identifying which lines of source code are associated with each node.

Figure 4.1 shows the Control Flow Graph for the program seatsAvailable().

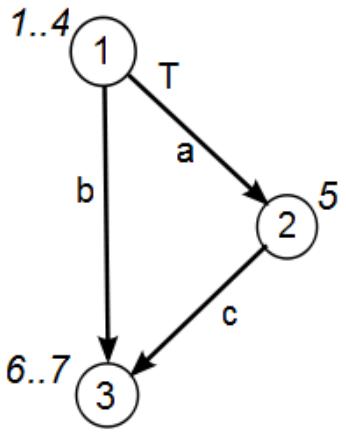


Figure 4.1: CFG for seatsAvailable()

- Node 1: Lines 1–4
- Node 2: Line 5
- Node 3: Lines 6, 7

Test Cases

Each node, representing an indivisible sequence of lines of source code, is a test case: 1, 2, 3.

Table 4.12: SC Test Cases for seatsAvailable()

Case	Node	Test
1	1	29
2	2	29
3	3	29

Test Data

The tests should generate data that will cover all the nodes 1..3. Experience allows a minimum set of tests to be developed – but the emphasis is on ensuring full coverage.

If the technique is being used to augment Black-Box tests, then only those test cases (i.e. branches) not covered by the Black-Box Tests need be considered. In this example, however, we will consider this to be a standalone test, and cover all of the branches.

Table 4.13: SC Tests for seatsAvailable()

ID	Test Cases Covered	Inputs		Expected Output
		freeSeats	seatsRequired	return value
29	1,2,3	50	25	true

4.1.10 Branch Coverage

Reminder: run tests that ensure that every branch in the source code has been taken.

CFG

Develop the CFG from the source code, clearly labeling all the nodes and edges.

Test Cases

Each edge is a test case: a, b, c.

Table 4.14: BC Test Cases for seatsAvailable()

Case	Node	Test
1	a	30
2	b	31
3	c	30

Test Data

The tests should generate data that will cause all the branches to be taken: 'a'..'c'. Experience allows a minimum set of tests to be developed – but the emphasis is on ensuring full branch coverage.

If the technique is being used to augment Black-Box tests, then only those test cases (i.e. branches) not covered by the Black-Box Tests need be considered. Also, if the technique is being used to augment Statement Coverage, then only those test cases (i.e. branches) not covered by the Statement Coverage tests need be considered. In this example, however, we will consider this to be a standalone test, and cover all of the branches.

Table 4.15: BC Tests for seatsAvailable()

ID	Test Cases Covered	Inputs		Expected Output
		freeSeats	seatsRequired	return value
30	a,c	50	25	true
31	b	-50	25	false

4.1.11 Condition Coverage

Reminder: run tests that ensure that every condition in every decision has been evaluated as both true and false during test execution.

Decisions and Conditions

In the program there is one decision with three conditions on line 4/Node 1.

```
if ((freeSeats>=0) && (seatsRequired>=1)
                   && (seatsRequired<=freeSeats))
```

Test Cases

Each Boolean value for each condition is a test case.

Table 4.16: CC Test Cases for seatsAvailable()

Case	Condition	Test
1	(freeSeats≥0)	32
2	!(freeSeats≥0)	33
3	(seatsRequired≥1)	33
4	!(seatsRequired≥1)	32
5	(seatsRequired≤freeSeats)	32
6	!(seatsRequired≤freeSeats)	33

Test Data

In complex programs it can be difficult to generate test data to force each condition to be true and false, but in this example it is straightforward as only the input parameters are used in the decisions. Note that the expected outputs are derived from the specification – and not from the source code!

Table 4.17: CC Tests for seatsAvailable()

ID	Test Cases Covered	Inputs		Expected Output
		freeSeats	seatsRequired	return value
32	1,4,5	50	-25	false
33	2,3,6	-50	25	false

4.1.12 Decision/Condition Coverage

Reminder: run tests that ensure that every decision, and every condition in every decision, has been evaluated as both true and false during test execution.

Decisions and Conditions

In the program there is one decision with three conditions on line 4.

```
if ((freeSeats>=0) && (seatsRequired>=1) && (seatsRequired<=freeSeats))
```

Test Cases

Each Boolean value for each decision is a test case. And, in addition, each Boolean value for each condition is a test case.

Table 4.18: DCC Test Cases for seatsAvailable()

Case	Condition	Test
1	((freeSeats>=0) && (seatsRequired>=1) && (seatsRequired≤freeSeats))	34
2	!((freeSeats>=0) && (seatsRequired>=1) && (seatsRequired≤freeSeats))	35
3	(freeSeats>=0)	34
4	!(freeSeats>=0)	35
5	(seatsRequired>=1)	34
6	!(seatsRequired>=1)	35
7	(seatsRequired≤freeSeats)	34
8	!(seatsRequired≤freeSeats)	35

Test Data

Table 4.19: DCC Tests for seatsAvailable()

ID	Test Cases Covered	Inputs		Expected Output
		freeSeats	seatsRequired	return value
34	1,3,5,7	50	25	true
35	2,4,6,8	-50	-25	false

4.1.13 Multiple Condition Coverage

Reminder: run tests that ensure that every possible combination of conditions in every decision has been evaluated during test execution.

Decisions and Conditions

In the program there is one decision with three conditions on line 4.

```
if ((freeSeats>=0) && (seatsRequired>=1) && (seatsRequired<=freeSeats))
```

Test Cases

Each combination of values for the conditions is a test case.

Table 4.20: MCC Test Cases for seatsAvailable()

Case	freeSeats≥0	seatsRequired≥1	seatsRequired≤freeSeats	Test
1	T	T	T	36
2	T	T	F	37
3	T	F	T	38
4	T	F	F	
5	F	T	T	
6	F	T	F	39
7	F	F	T	40
8	F	F	F	41

Note: the shaded rows indicate impossible test cases, and therefore are not covered by any of the tests.

Table 4.21: MCC Tests for seatsAvailable()

ID	Test Cases Covered	Inputs		Expected Output
		freeSeats	seatsRequired	return value
36	1	50	25	true
37	2	50	75	false
38	3	50	-25	false
39	6	-50	25	false
40	7	-50	-75	false
41	8	-50	-25	false

4.1.14 Path Coverage

Reminder: run tests that ensure that every path from entry to exit has been taken during test execution.

Paths

Examining the control flow graph, two paths can be seen through the program

- Node 1–3
- Node 1–2–3

Test Cases

Each path is a separate Test Case, as shown below.

Table 4.22: Path Test Cases for seatsAvailable()

Case	Nodes	Test
1	1,3	30
2	1,2,3	31

Test Data

Each path must be tested in a separate test. It is straightforward to create tests to cover both these paths.

In this example, these test cases are already covered by tests derived using Branch Coverage. There is no need to duplicate the tests – it provides no extra coverage – instead the Test column for the Test Cases refers back to the existing tests.

However, it must be noted though that this will not always be the case – Branch Coverage does not guarantee Path Coverage.

Special Note

This provides an example of why unique identifiers must be used for different tests on the same SUT. Note that in all other cases duplicate tests have **not** been identified in order to show the full implementation of each test technique. This is purely for completeness in showing the application of the technqiues; in practice a tester would **never** duplicate tests.

4.1.15 D-U pair Coverage

Reminder: run tests that ensure that every possible path from the definition to the use of every variable (parameter, local variable, or attribute) has been followed during test execution.

Variables

- freeSeats
- seatsRequired
- rv

D-U Pairs

Note: a definition is the assignment of a value to a variable, including assignment at function entry. A use is the reading of the value from a variable. Increment and decrement operations cause a use followed by a definition.

The d-u pairs for freeSeats are shown in Table 4.23.

Table 4.23: DU Pairs for freeSeats

d-u pair	D	U
1	1	4

The d-u pairs for seatsRequired are shown in Table 4.24.

Table 4.24: DU Pairs for seatsRequired

d-u pair	D	U
2	1	4

And the d-u pairs for rv are shown in Table 4.25.

Table 4.25: DU Pairs for rv

d-u pair	D	U
3	3	6
4	5	6

Test Cases

Each d-u pair is a test case.

Table 4.26: DUP Test Cases for seatsAvailable()

Case	DUP	Test
1	1	42
2	2	42
3	3	43
4	4	42

Test Data

Multiple d-u pairs may be tested in each test – sometimes it is possible to test all the d-u pairs in a single test.

Table 4.27: DUP Tests for seatsAvailable()

ID	Test Cases Covered	Inputs		Expected Output
		freeSeats	seatsRequired	return value
42	1,2,4	50	50	true
43	1,2,3	50	75	false

4.1.16 Strong Mutation Testing

Reminder: changes, or *mutations*, are made to the original source code, and then existing tests are run to see if they cause a different output (resulting in test failure) during execution.

Mutations

Only one example is given here of all the possible mutations.

A Logical Connector Replacement (LCR) can be made on line 4 by replacing an AND in the condition with an OR. If the original line is:

```
(4) if ((freeSeats>=0) && (seatsRequired>=1)
                      && (seatsRequired<=freeSeats))
```

then one possible LCR mutation (Test Case LCR-1) is:

```
(4) if ((freeSeats>=0) || (seatsRequired>=1)
                      && (seatsRequired<=freeSeats))
```

If previously developed tests (e.g. Equivalence Partitions) are run against this version of the code, and the tests pass, then we know there is either a problem in the code or in the test, and they must both be carefully examined to find the cause! If the test fails, then we know that this fault was not in the original source code.

For example, the test data we derived for Branch Testing was

Table 4.28: Initial SMT Test Results for seatsAvailable()

ID	**Inputs**		**Expected Output**	**Actual Output**
	freeSeats	seatsRequired	return value	return value
30	50	25	true	true
31	-50	25	false	false

If we apply the same test to this mutated program, the test will still pass!

Here, the test data is insufficient to identify the mutation. This implies that if this error existed in our program code we could not identify it with the current test data. The test data now has to be extended to identify the mutation.

Table 4.29: Improved SMT Test Results for seatsAvailable()

ID	**Test Cases Covered**	**Inputs**		**Expected Output**	**Actual Output**
		freeSeats	seatsRequired	return value	return value
30		50	25	true	true
31		-50	25	false	false
44	LCR-1	0	0	false	true

Now, with the new test data, Test 3 will fail on the mutation. The mutation has been identified. Furthermore, there is now a test to ensure that this Logical connector error does not exist in the program.

4.2 Example Two: premium()

The second example is based on a car insurance system. It is a function that calculates the premium for car insurance.

Note that, as for the previous example, duplicate tests **are** included in order to show the full application of each technique. In practice, as shown in Section 4.1.14, tests would **not** be duplicated, and the existing test would be referred to instead.

This is a new SUT, so test identity numbers start at 1 again: there is no confusion between the tests "seats-test-1" and "premium-test-1".

4.2.1 Description

The basic cost of an insurance premium for drivers is €500. However, this premium can increase or decrease depending on three factors: their age, their gender and their marital status.

Age is an integer. Gender is given by the character 'M' for male and 'F' for female. Married is a boolean value.

Drivers that are below the age of 25, male and single face an additional premium increase of €1500. If a driver outside of this bracket is married or female their premium reduces by €200, and if they are aged between 45 and 65 inclusive, their premium goes down by €100.

Drivers below the age of 16 and greater than the age of 65 cannot be insured and will return a value of zero for the premium. Program error checking to prevent an illegal entry for gender will also return a value of zero for the premium.

4.2.2 Specification

Program Inputs

age: INT_MIN..15, 16..24, 25..44, 45..65, 66..INT_MAX

gender: 'M', 'F', invalid input

married: true, false

Program Outputs

return value:

0 - uninsurable age, or illegal inputs

200 - married or female, and $45 \leq \text{age} \leq 65$

300 - married or female, and $16 \leq \text{age} < 45$

400 - single male, and $45 \leq \text{age} \leq 65$

500 - single male, and $25 \leq \text{age} < 45$

2000 - single male, less than 25

4.2.3 Source Code

The source code for this method is shown below:

```
 1 public int premium(int age, char gender, boolean married) {
 2    int premium;
 3    if ((age<16) || (age>65) || (gender!='M' && gender!='F')) {
 4       premium=0;
 5    } else {
 6       premium=500;
 7       if ((age<25) && (gender=='M') && (!married)) {
 8          premium += 1500;
 9       } else {
10          if (married || gender=='F')
11             premium -= 200;
12          if ((age>=45) && (age<=65))
13             premium -= 100;
14       }
15    }
16    return premium;
17 }
```

4.2.4 Equivalence Partitioning

Summary

An equivalence partition is defined as a range of values for a parameter (input or output) for which the same processing is specified (though of course different output values may be generated). The input partitions must be identified before the test cases can be written.

Equivalence Partitions and Test Cases

The input partitions and Test Cases are shown in Table 4.30.

Table 4.30: EP Input Test Cases for premium()

Case	Parameter	Range	Test
1*	Age	INT_MIN..15	6
2		16..24	5
3		25..44	2
4		45..65	3
5*		66..INT_MAX	7
6	gender	'M'	3
7		'F'	1
8*		Invalid input	8
9	married	true	1
10		false	2

Notes:

1. * indicates an "error" case.
2. INT_MIN is the minimum possible integer value and INT_MAX is the maximum possible integer value.

The output partitions and Test Cases are shown in Table 4.31.

Table 4.31: EP Output Test Cases for premium()

Case	Parameter	Range	Test
11	premium	0	6
12		200	1
13		300	2
14		400	3
15		500	4
16		2000	5

Test Cases

Each partition is a test case, as numbered above.

Test Data

Table 4.32: EP Tests for premium()

ID	Test Cases Covered	Inputs			Expected Output
		age	gender	married	return value
1	4, 7, 9, 12	50	'F'	true	200
2	3, 7, 10, 13	30	'F'	false	300
3	4, 6, 10, 14	50	'M'	false	400
4	3, 6, 10, 15	30	'M'	false	500
5	2, 6, 10, 16	20	'M'	false	2000
6*	1	5	'M'	false	0
7*	5	70	'M'	false	0
8*	8	50	'G'	false	0

Note: each error case is tested separately – otherwise multiple simultaneous errors may hide correct error processing for one of the errors.

4.2.5 Boundary Value Analysis

Summary

Having identified the Equivalence Partitions, it is straightforward to identify the Boundary Values at the lower and upper end of each Partition.

Boundary Values and Test Cases

The input boundary values and Test Cases are shown in Table 4.33.

Table 4.33: BV Input Test Cases for premium()

Case	Parameter	Boundary Value	Test
1*	age	INT_MIN	15
2*		15	16
3		16	14
4		24	13
5		25	10
6		44	12
7		45	9
8		65	11
9*		66	17
10*		INT_MAX	18
11	gender	'M'	11
12		'F'	9
13*		invalid value	19
14	married	true	9
15		false	11

Notes:

1. * indicates an "error" case – each error case must be tested separately

The output boundary values and Test Cases are shown in Table 4.34.

Table 4.34: BV Output Test Cases for premium()

Case	Parameter	Boundary Value	Test
16	return value	0	7
17		200	9
18		300	10
19		400	11
20		500	12
21		2000	13

Test Cases

Each boundary value is a test case.

Test Data

Table 4.35: BV Tests for premium()

ID	Test Cases Covered	Inputs			Expected Output
		age	gender	married	return value
9	7, 12, 14, 17	45	'F'	true	200
10	5, 12, 14, 18	25	'F'	true	300
11	8, 11, 15, 19	65	'M'	false	400
12	6, 11, 15, 20	44	'M'	false	500
13	4, 11, 15, 22	24	'M'	false	2000
14	3, 11, 15, 21	16	'M'	false	2000
15*	1	INT_MIN	'M'	false	0
16*	2	15	'M'	false	0
17*	9	66	'M'	false	0
18*	10	INT_MAX	'M'	false	0
19*	13	16	'G'	false	0

Note: by definition, Boundary Value Analysis covers all the Equivalence Partition test cases.

4.2.6 Combinational Testing

Summary

To identify a minimum subset of possible combinations that will test all the different behaviours of the program, a Truth Table is created. The inputs ("Causes") and outputs ("Effects") are specified as Boolean expressions (using predicate logic); these expressions specify the conditions required for a particular variable. Test Cases are then constructed, one for each rule in the Truth Table.

Causes

The number of causes should be minimized to reduce the size of the Truth Table – in particular, where a parameter has a range of values that provide a particular response, this can be expressed as a single cause. The causes for this program, taken from the specification, can be expressed as follows:

1. age<16
2. 16≤age≤24
3. 25≤age<45
4. 45≤age≤65
5. age>65
6. gender='M'
7. gender='F'
8. Married==true

Effects

1. Premium=0
2. Premium=200
3. Premium=300
4. Premium=400
5. Premium=500
6. Premium=2000

Note: it is not necessary to include the cause "age>65" as this must be true if all the other possible value ranges for age are false. However, for clarity, it *is* included here.

Truth Table

To generate the Truth Table, each Cause is listed in a separate row, and then a different column is used to identify each combination of Causes that creates a different Effect. Each column is referred to as a Rule in the Truth Table – each Rule is a test case.

Table 4.36: Truth Table for premium()

	Rules											
	1	2	3	4	5	6	7	8	9	10	11	12
Causes												
age<16	T	F	F	F	F	F	F	F	F	F	F	*
16≤age≤24	F	T	T	T	F	F	F	F	F	F	F	*
25≤age<45	F	F	F	F	T	T	T	F	F	F	F	*
45≤age≤65	F	F	F	F	F	F	F	T	T	T	F	*
age>65	F	F	F	F	F	F	F	F	F	F	T	*
gender='M'	*	T	T	F	T	F	*	T	F	*	*	F
gender='F'	*	F	F	T	F	T	*	F	T	*	*	F
Married=true	*	F	T	*	F	F	T	F	F	T	*	*
Effects												
Premium=0	T	F	F	F	F	F	F	F	F	F	T	T
Premium=200	F	F	F	F	F	F	F	F	T	T	F	F
Premium=300	F	F	T	T	F	T	T	F	F	F	F	F
Premium=400	F	F	F	F	F	F	F	T	F	F	F	F
Premium=500	F	F	F	F	T	F	F	F	F	F	F	F
Premium=2000	F	T	F	F	F	F	F	F	F	F	F	F
	20	21	22	23	24	25	26	27	28	29a	29b	29c
	Tests											

For rules 1, 7, 10, and 11 gender must be either 'M' or 'F'. The notation 29a, 29b, 29c is an example of how to insert tests without having to renumber subsequent test IDs.

Test Data

Each Rule is a Test Case, and needs to be tested in a separate test. The test data is derived by picking values that satisfy the Causes and Effects for a Rule.

Table 4.37: TT Tests for premium()

ID	Test Cases Covered	Inputs			Expected Output
		age	gender	married	return value
20	1	15	'M'	false	0
21	2	20	'M'	false	2000
22	3	20	'M'	true	300
23	4	20	'F'	true	300
24	5	36	'M'	false	500
25	6	36	'F'	false	300
26	7	36	'M'	true	300
27	8	50	'M'	false	400
28	9	50	'F'	false	200
29a	10	50	'M'	true	200
29b	11	70	'M'	false	0
29c	12	36	'G'	true	0

4.2.7 Statement Coverage

Summary

First of all, draw a CFG for the program. Then working from the CFG, and the source code, work out input values required to ensure that every source code statement is executed. Finally, using the specification – NOT the source code – work out the correct output for each set of input values.

Figure 4.2 shows the Control Flow Graph for the progam `premium()`, with the nodes numbered from 1 to 11.

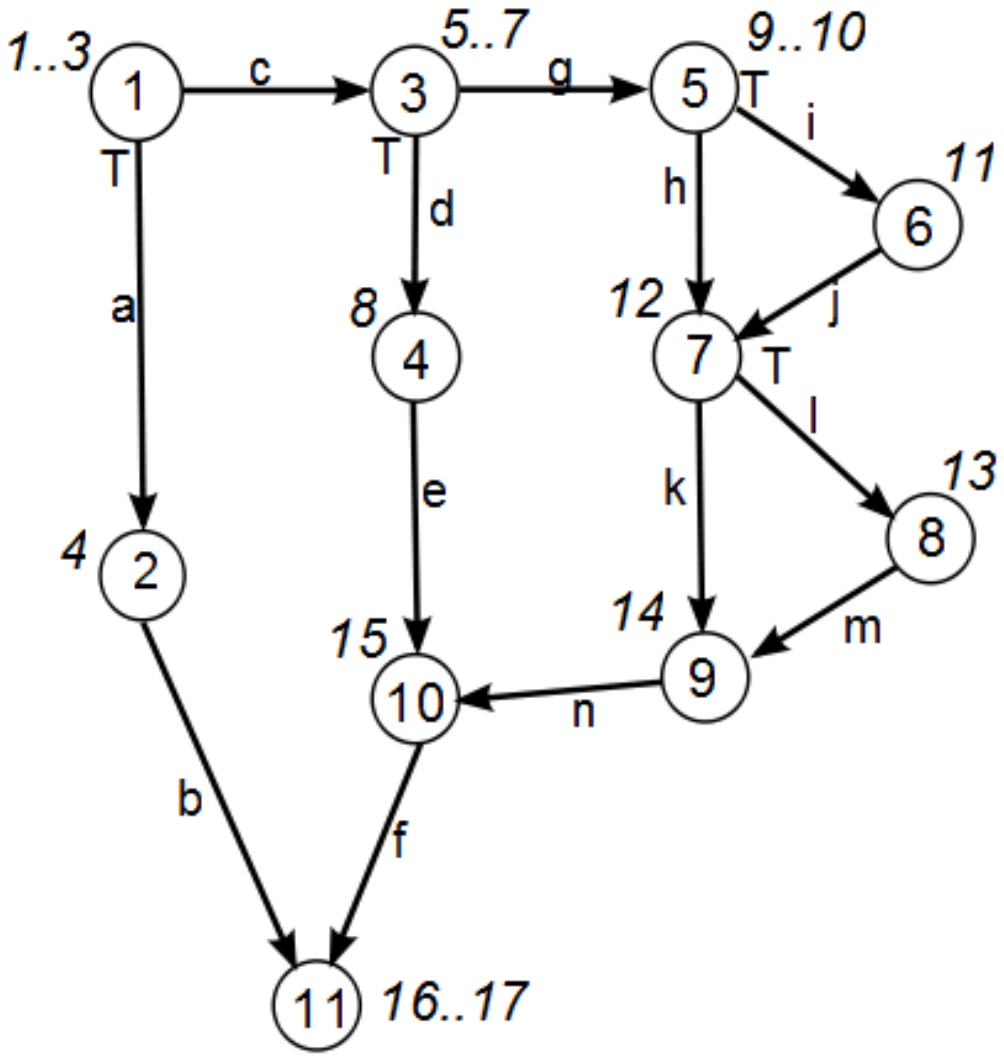


Figure 4.2: CFG for premium()

Test Cases

Each node, representing an indivisible sequence of lines of source code, is a test case (1–11).

Table 4.38: SC Test Cases for premium()

Case	Node	Test
1	1	30
2	2	30
3	3	31
4	4	31
5	5	32
6	6	32
7	7	32
8	8	32
9	9	32
10	10	31
11	11	30

Test Data

The tests should generate data that will cover all the nodes.

Table 4.39: SC Tests for premium()

ID	Test Cases Covered	Inputs			Expected Output
		age	gender	married	return value
30	1, 2, 11	15	'M'	false	0
31	1, 3, 4, 10, 11	24	'M'	false	2000
32	1, 3, 5, 6, 7, 8, 9, 10, 11	46	'F'	false	200

4.2.8 Branch Testing

Summary

First of all, draw a CFG for the program. Then working from the CFG, and the source code, work out input values required to ensure that every source code branch is taken. Finally, using the specification – NOT the source code – work out the correct output for each set of input values.

CFG

The CFG for `premium()` is given in Figure 4.2 with the edges labelled from 'a' to 'n'.

Test Cases

Each edge, representing a branch taken, is a test case: 'a'..'n'. Note that Test Cases don't need to be numbered – here the edge label is used to make the test cases easier to review against the CFG.

Table 4.40: BC Test Cases for premium()

Case	Node	Test
a	a	33
b	b	33
c	c	34
d	d	34
e	e	34
f	f	34
g	g	35
h	h	35
i	i	36
j	j	36
k	k	35
l	l	36
m	m	36
n	n	35

Test Data

Table 4.41: BC Tests for premium()

ID	Test Cases Covered	Inputs			Expected Output
		age	gender	married	return value
33	a, b	15	'M'	false	0
34	c, d, e, f	24	'M'	false	2000
35	c, g, h, k, n, f	25	'M'	false	500
36	c, g, i, j, l, m, n, f	45	'F'	true	200

4.2.9 Condition Coverage

Summary

A complex decision is formed from multiple (boolean) conditions. Condition Coverage augments Branch Testing by ensuring that, for complex decisions, each condition is tested for true and false values.

Conditions

In the program there is one decision with four conditions on line 3:

```
if ((age<16) || (age>65) || (gender!='M' && gender!='F'))
```

One decision with three conditions on line 7:

```
if ((age<25) && (gender=='M') && (!married))
```

One decision with two conditions on line 10:

```
if (married || gender=='F')
```

And one decision with two conditions on line 12:

```
if ((age>=45) && (age<=65))
```

Test Cases

Each Boolean value for each condition is a test case.

Table 4.42: CC Test Cases for premium()

Case	**Decision**	**Line**	**Condition**	**Test**
1	1	3	(age<16)	37
2			!(age<16)	38
3			(age>65)	38
4			!(age>65)	37
5			(gender!='M')	38
6			!(gender!='M')	37
7			(gender!='F')	37
8			!(gender!='F')	38
9	2	7	(age<25)	39
10			!(age<25)	40
11			(gender='M')	39
12			!(gender='M')	40
13			(!married)	39
14			!(!married)	40
15	3	10	(married)	40
16			!(married)	41
17			(gender=='F')	40
18			!(gender=='F')	41
19	4	12	(age≥45)	40
20			!(age≥45)	41
21			(age≤65)	40
22			!(age≤65)	Not Possible

Test Data

Particular care must be taken when selecting Test Data to make sure that the decision being tested can actually be reached with each particular set of values. For example, it is not possible to reach decision 4 on line 12 with age > 65.

Table 4.43: CC Tests for premium()

ID	**Test Cases Covered**	**Inputs**			**Expected Output**
		age	gender	married	return value
37	1, 4, 6, 7	15	'M'	false	0
38	2, 3, 5, 8	66	'F'	false	0
39	2, 4, 6, 7, 9, 11, 13	24	'M'	false	2000
40	2, 4, 5, 8, 10, 12, 14, 15, 17, 19, 21	65	'F'	true	200
41	2, 4, 6, 7, 10, 11, 13, 16, 18, 20, 21	25	'M'	false	500

Decision/Condition Coverage

Summary

Tests are generated to cause every decision to be taken at least once (branch coverage), and also every condition to be true and false at least once (Condition Coverage).

Test Cases

Each Boolean value for each decision, and each Boolean value for each condition, is a test case:

Table 4.44: DCC Test Cases for premium()

Case	Line	Decision/Condition	Test
1	3	((age<16) || (age>65) || (gender!='M' && gender!='F'))	42
2		!((age<16) || (age>65) || (gender!='M' && gender!='F'))	44
3		(age<16)	42
4		!(age<16)	43
5		(age>65)	43
6		!(age>65)	44
7		(gender!='M')	43
8		!(gender!='M')	43
9		(gender!='F')	43
10		!(gender!='F')	44
11	7	((age<25) && (gender=='M') && (!married))	44
12		!((age<25) && (gender=='M') && (!married))	45
13		(age<25)	44
14		!(age<25)	45
15		(gender=='M')	44
16		!(gender=='M')	45
17		(!married)	44
18		!(!married)	45
19	10	(married || gender=='F')	45
20		!(married || gender=='F')	46
21		married	45
22		!married	46
23		(gender=='F')	46
24		!(gender=='F')	46
25	12	((age≥45) && (age≤65))	45
26		!((age≥45) && (age≤65))	46
27		(age≥45)	45
28		!(age≥45)	46
29		(age≤65)	45
30		!(age≤65)	Not possible

Test Data

As for Condition Coverage, care must be taken when selecting Test Data to make sure that the decision being tested can actually be reached with each particular set of values. For example, it is not possible to reach the decision on line 12 with age > 65 (Test Case 30).

Table 4.45: DCC Tests for premium()

ID	Test Cases Covered	Inputs			Expected Output
		age	gender	married	return value
42	1, 3, 6, 8, 9	15	'M'	false	0
43	1, 4, 5, 7, 10	66	'F'	false	0
44	2, 4, 6, 8, 9, 11, 13, 15, 17	24	'M'	false	2000
45	2, 4, 6, 7, 10, 12, 14, 16, 18, 19, 21, 23, 25, 27, 29	65	'F'	true	200
46	2, 4, 6, 8, 9, 12, 14, 15, 17, 20, 22, 24, 26, 28, 29	25	'M'	false	500

4.2.10 Multiple Condition Coverage

Summary

Tests are generated to cause every possible combination of conditions for every decision to be tested.

Multiple Conditions

Decision 1: `((age<16) || (age>65) || (gender!='M' && gender!='F'))`

Decision 2: `((age<25) && (gender='M') && (!married))`

Decision 3: `(!married) && (gender='F')`

Decision 4: `(age>=45) && (age<=65)`

Test Cases

Each combination of values for the conditions is a test case.

The Test Cases for Decision 1 are shown in Table 4.46.

Table 4.46: MCC Test Cases for premium()/Decision 1

Case	age<16	age>65	gender!='M'	gender!='F'	**Test**
1	F	F	F	F	
2	F	F	F	T	47
3	F	F	T	F	48
4	F	F	T	T	49
5	F	T	F	F	
6	F	T	F	T	50
7	F	T	T	F	51
8	F	T	T	T	52
9	T	F	F	F	
10	T	F	F	T	53
11	T	F	T	F	54
12	T	F	T	T	55
13	T	T	F	F	
14	T	T	F	T	
15	T	T	T	F	
16	T	T	T	T	

Note: the shaded rows indicate impossible Test Cases. For example, Test Case 31 cannot be exercised as the decision on Line 3 catches any age values over 65.
The Test Cases for Decision 2 are shown in Table 4.47.

Table 4.47: MCC Test Cases for premium()/Decision 2

Case	age<25	gender='M'	!married	**Test**
17	F	F	F	48
18	F	F	T	59
19	F	T	F	56
20	F	T	T	47
21	T	F	F	60
22	T	F	T	61
23	T	T	F	57
24	T	T	T	58

The Test Cases for Decision 3 are shown in Table 4.48.

Table 4.48: MCC Test Cases for premium()/Decision 3

Case	married	gender='F'	**Test**
25	F	F	47
26	F	T	59
27	T	F	56
28	T	T	48

And the Test Cases for Decision4 are shown in Table 4.49.

Table 4.49: MCC Test Cases for premium()/Decision 4

Case	Age≥45	Age≤65	**Test**
29	F	F	
30	F	T	47
31	T	F	
32	T	T	48

Note: the shaded rows indicate impossible test cases

Test Data

Table 4.50: MCC Tests for premium()

ID	**Test Cases Covered**	**Inputs**			**Expected Output**
		age	gender	married	return value
47	2, 20, 25, 30	25	'M'	false	500
48	3, 17, 28, 32	46	'F'	true	200
49	4	25	'G'	false	0
50	6	66	'M'	false	0
51	7	66	'F'	false	0
52	8	66	'G'	false	0
53	10	15	'M'	false	0
54	11	15	'F'	true	0
55	12	15	'G'	false	0
56	2, 19, 27, 30	25	'M'	true	300
57	2, 23, 27, 30	24	'M'	true	300
58	2, 24, 25, 30	24	'M'	false	2000
59	3, 18, 26, 30	25	'F'	false	300
60	3, 21, 28, 30	24	'F'	true	300
61	3, 22, 26, 30	24	'F'	false	300

4.2.11 Path Testing

Summary

Path testing causes every possible path from entry to exit of the program to be taken.

Paths

The program can be described using a regular expression as follows:

$$1.(2+3.(4+5.(6+0).7.(8+0).9).10).11$$

Replacing all the node numbers, including nulls, by 1 to compute the number of paths through the program gives:

paths = 1.(1+1.(1+1.(1+1).1.(1+1).1).1).1
= 1.(1+1.(1+1.(2).1.(2).1).1).1
= 1.(1+(1+4).1)
= 6

From the regular expression, or by examining the CFG, the 6 possible paths through the program can be identified:

1. 1.2.11
2. 1.3.4.10.11
3. 1.3.5.7.9.10.11
4. 1.3.5.6.7.9.10.11
5. 1.3.5.7.8.9.10.11
6. 1.3.5.6.7.8.9.10.11

Test Cases

Each path is a separate Test Case, as shown below.

Table 4.51: Path Test Cases for premium()

Case	Nodes	Test
1	1.2.11	62
2	1.3.4.10.11	63
3	1.3.5.7.9.10.11	64
4	1.3.5.6.7.9.10.11	65
5	1.3.5.7.8.9.10.11	66
6	1.3.5.6.7.8.9.10.11	67

Test Data

Table 4.52: Path Tests for premium()

ID	Test Cases Covered	Inputs			Expected Output
		age	gender	married	return value
62	1	15	'M'	false	0
63	2	24	'M'	false	2000
64	3	25	'M'	false	500
65	4	25	'F'	true	300
66	5	45	'M'	false	400
67	6	45	'F'	false	200

4.2.12 d-u pair Testing

Summary

The principle in d-u pair testing is to execute each path between the definition of the value in a variable and its subsequent use.

Variables

- age
- gender
- married
- premium

D-U Pairs and Test Cases

The d-u pairs for age are shown in Table 4.53.

Table 4.53: DU Pairs for age

Case	D	U	Test
1	1	3	68
2	1	7	69
3	1	12	70

The d-u pairs for gender are shown in Table 4.54.

Table 4.54: DU Pairs for gender

Case	D	U	Test
4	1	3	68
5	1	7	69
6	1	10	70

The d-u pairs for married are shown in Table 4.55.

Table 4.55: DU Pairs for married

Case	D	U	Test
7	1	7	69
8	1	10	70

And the d-u pairs for premium are shown in Table 4.56.

Table 4.56: DU Pairs for premium

Case	D	U	Test
9	6	8	69
10	8	8	70
11	6	11	71
12	11	11	71
13	6	13	72
14	11	13	71
15	13	13	71
16	4	16	78
17	6	16	70
18	8	16	69
19	11	16	73
20	13	16	72

Test Data

Multiple d-u pairs may be tested in each test – sometimes it is possible to test all the d-u pairs in a single test as shown in Table 4.57.

Table 4.57: DUP Tests for premium()

ID	Test Cases Covered	Inputs			Expected Output
		age	gender	married	return value
68	1,4,16	15	'M'	false	0
69	1,2,4,5,7,9,10,18	24	'M'	false	2000
70	1,2,3,4,5,6,7,8,17	25	'M'	false	500
71	1,2,3,4,5,6,7,8,11,12,14,15,20	45	'F'	true	200
72	1,2,3,4,5,6,7,8,13,15,20	45	'M'	false	400
73	1,2,3,4,5,6,7,8,11,12,19	25	'F'	false	300

4.2.13 Strong Mutation Testing

Summary

Mutation testing is a form of "fault-insertion" testing. In Strong Mutation Testing faults are inserted into "mutations" of the source code, and the tests (developed using one of the previous techniques) are run against these. The tests should fail! This can be used (a) to show that the particular faults inserted do NOT exist in the code, and/or (b) to verify the validity of the test cases created (by ensuring that they fail on all inserted faults).

Mutation (1)

A Relational Operator Replacement (ROR) can be made on line 9 by inserting a NOT operator ("!") in the condition.

For the original line:

```
(9) if (married || gender=='F') {
```

we can define Mutation(1), for example, as the following ROR mutation:

```
(9) if (!married || gender=='F') {
```

If previously developed tests are run against this version of the code, and the tests pass, then we know there is either a problem in the code or in the test. If the test fails, then we know that this fault was not in the original source code.

For example, the test data we derived for Statement Testing is shown in Table 4.58.

Table 4.58: SC Tests for premium()

ID	Test Cases Covered	Inputs			Expected Output
		age	gender	married	return value
30	1, 2, 11	15	'M'	false	0
31	1, 3, 4, 10, 11	24	'M'	false	2000
32	1, 3, 5, 6, 7, 8, 9, 10, 11	46	'F'	false	200

If we apply the same test to this mutated program, the test still passes!

Table 4.59: SMT SC Test Results for premium()/Mutation(1)

Test No.	Expected Output	Actual Output
30	0	0
31	2000	2000
32	200	200

Here, the test data is insufficient to identify the mutation. This implies that if this error existed in our program code we could not identify it with the current test data. Recalling the test data that was generated for Branch Testing:

Table 4.60: BC Tests for premium()

ID	Test Cases Covered	Inputs			Expected Output
		age	gender	married	return value
33	a, b	15	'M'	false	0
34	c, d, e, f	24	'M'	false	2000
35	c, g, h, k, n, f	25	'M'	false	500
36	c, g, i, j, l, m, n, f	45	'F'	true	200

Applying this test data to the mutant the results of test 3 are found to be different. This validates the use of Branch testing for finding this type of fault.

Table 4.61: SMT BC Test Results for premium()/Mutation(1)

Test No.	**Expected Output**	**Actual Output**
33	0	0
34	2000	2000
35	500	300
36	200	200

Chapter 5

Static Verification

This book is primarily concerned with the verification of executable code, closely aligned with the IEEE definition of software testing. This chapter provides a brief introduction to some non-executable verification techniques.

The two most common forms of static verification approaches are design reviews and static code analysis.

5.1 Design Reviews

A design review provides the opportunity to verify that the design for software is correct before it is implemented. The design may be at a high/architectural level, or at a detailed level. The purpose of the review is to ensure that the design allows the software to fulfill its goals.

Using UML, there are two similar approaches to this, both based on following the trail of execution for each use scenario.

5.1.1 Informal Walk-through

The first is a *walk-through* where team members take on the roles of different objects, and the trail of execution is followed from the input stimulus to the output response. The focus is on making sure that all the correct objects and relationships are in place to allow the scenario to be completed. This is often an informal activity, with only required changes being noted down for future action. The focus is on trying to discover faults rather than verify that simple scenarios work. The benefits of this approach are in terms of time and effort – it is quick to prepare for, and quick to execute a review, so they can take place frequently. The disadvantage is that there is typically no written record, and a non-written approach can lead to ambiguities, leading to reduced software quality. This form is more appropriate for software requiring quick time-to-market, low development cost, or high flexibility (perhaps where the user requirements are not well understood in advance).

5.1.2 Formal Design Review

The second is a formal review, where in advance each scenario is addressed. UML sequence diagrams are used to demonstrate the trail of execution for each scenario – these show the objects, method calls, and parameter values. This tends to be a more formal activity, where the design is assessed against each scenario, and it either passes or fails each. The benefits of a formal review are that documenting the execution trails usually leads to most problems being discovered and fixed before the review, and that there

is a written record of the review, both leading to a higher quality product. The disadvantages are in terms of the time and effort it takes to prepare for and run the review. This form is more appropriate for software that has a high quality requirement, such as mission-critical, life-critical systems.

5.2 Static Code Analysis

There are many techniques for verifying the correct behaviour of software, or searching for potential faults, without executing it (for example: symbolic execution, source-code analysis tools, and manual review). Technically these are not software testing techniques; the term "Static Analysis" has been coined to describe these activities.

In this section two manual review techniques are described: walk-throughs and code inspections. The specification and/or the code is reviewed for errors by someone other than the programmer. This form of testing is very valuable and has advantages over execution-based testing. Reported experience shows that a large number of faults can be found using inspection. For example, according to a report by the Ganssle group on Code inspections [1].

- HP found that 80% of the defects detected during inspections were unlikely to be caught by testing
- HP, Shell Research, Bell Northern, and AT&T all found inspections to be 20 to 30 times more efficient than execution-based testing in detecting errors
- IBM found that inspections gave a 38% reduction in defects detected after unit testing

This has benefits in terms of cost and productivity because faults are found (and corrected) early on and less time is required for execution-based tests. The most significant disadvantage of these tests is that because of the interpersonal nature of these activities they could be used for a performance appraisal of the developer or developers. If the developer or developers sense this, it may encourage them to be defensive and unhelpful under questioning and responding badly to criticism. Thus, these meetings should be conducted in a very professional and non-threatening manner.

5.2.1 Walk-throughs

The objective of a walk-through is to detect faults in a specification/program but it can also ensure that it adheres to certain standards, be they internal or external to the company.

The input to the walk-through process is the specification/code under test, a set of objectives to review in relation to them, perhaps in the form of a set of guidelines or a checklist, and a copy of any relevant set of standards.

Sample checks may include:

- Are the requirements precise, unambiguous and clear?
- Are there any data-referencing or declaration errors?
- Are there any control-flow errors? Or errors in the input/output screens?

The output from the walk-through is normally a report that contains a checklist of faults that need to be fixed, and maybe a set of test cases to be applied in dynamic testing at a later stage. The personnel present at a walk-through are a presenter, an inquisitor (from the QA department) and an administrator. The presenter may or may not be the code author. All the material for the meeting should be distributed as a document in advance so that all attendees have ample time to read it.

The actual walk-through can take two forms. In the first the meeting is driven forward by the personnel. Here, the inquisitor will question the presenter about unclear and incorrect items in the code, and the presenter must respond to clarify the intention. In its second form the presenter will guide the inquisitor through the document page-by-page. Along the way, the inquisitor will interrupt the presenter with comments or questions triggered by the presentation. Any faults found are recorded for later correction – it is not usual to fix them immediately.

[1] Available at: http://www.ganssle.com/Inspections.pdf

5.2.2 Code Inspections

An inspection is more comprehensive than a walk-through. It is carried out by a team of four people, with the most important participants being the Designer and the Tester. The Designer is the person that has produced the software while the Tester is the person that must ensure its quality. The Inspection has five formal stages:

Stage 1 - Overview An overview document that details the product specifications/ design/code/plan is prepared by the person responsible for producing the product, the Designer. The document is distributed to participants.

Stage 2 - Preparation The Tester must understand the document in detail. A checklist of fault types generally found in inspections ranked by frequency should be available to help concentrate their efforts.

Stage 3 - Inspection At the meeting there is a walk-through of the document to ensure that each item in the checklist is covered. Figure 5.1 gives an example of a checklist (based on the Ganssle Group, *Guide to Code Inspections*, 2001). Any faults found are simply documented for later correction.

Stage 4 - Rework After the meeting all faults and issues are resolved.

Stage 5 - Follow-up The leader of the code inspection group must finally ensure that every issue has been resolved, and should produce a final report.

This will provide detail on items such as:

- faults found categorized by their type
- fault statistics (for example, the number of faults found compared to the number of faults found at same stage of development in other products)

The report should also be able to use this information to recommend the redesign of a module or modules if too many faults were found. Additionally, other modules with a similar function could be subject to more rigorous testing when they are produced. Once the report is finalized the inspection can be declared to be complete.

Project:
Author:
Function Name:
Date:

Number of Errors		
Major	**Minor**	**Error Type**
		Code does not meet firmware standards
		Function size and complexity unreasonable
		Unclear expression of ideas in the code
		Poor encapsulation
		Function prototype not correctly used
		Data types do not match
		Unitialised variables at start of function
		Unitialised variables going into loops
		Poor logic – does not function as needed
		Poor commenting
		Error condition not caught
		Switch statement without default case
		Incorrect syntax
		Non-reentrant code in dangerous places
		Slow code in an area where speed is important
		Other

Notes:

- *Major Errors could cause problems visible to a customer.*
- *Minor Errors include spelling mistakes, non-compliance with standards, and poor workmanship not causing problems visible to a customer.*

Figure 5.1: Sample Code Inspection Checklist

Chapter 6

Testing Object-Oriented Software

Testing object-oriented code uses the same principles discussed already, but applied within an object-oriented framework. First, this chapter discusses the features of object-oriented software, and then how the testing principles are applied is examined in more detail.

The basic model for testing OO software (invariably referred to as Object-Oriented Testing or OOT) is shown in Figure 6.1. Methods accept input data, interact with other methods and the class state (attributes), and provide output data. The relationships with other classes includes both calling methods in those classes, and inheritance.

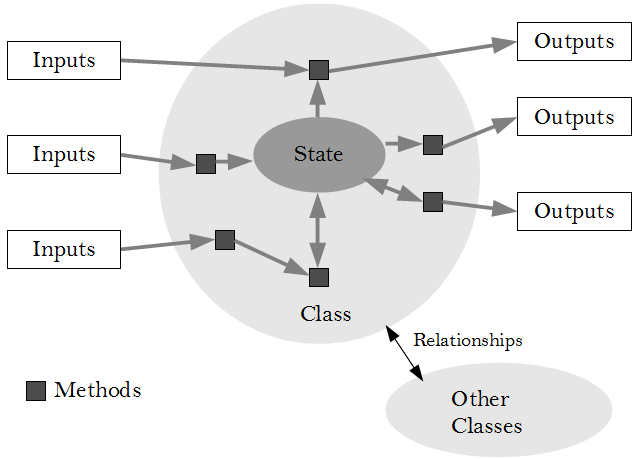


Figure 6.1: Object-Oriented Testing Model

6.1 Characteristics Of Object-Oriented Software

Object-orientation can be defined as programming with classes, inheritance, and messages. Classes provide a wrapper to package data (attributes) and associated code (methods). Inheritance provides a clean mechanism to re-use code, especially useful for implementing common features just once. And messages (or operators) provide access to the correct method within an inheritance tree. All of these OO features are intended to improve code quality, but they are complex and introduce their own "fault models" in turn:

- Classes: Objects Preserve State, but state control (the acceptable sequence of events) is typically distributed over an entire program. State control errors are likely.
- Inheritance: Dynamic Binding & Complex Inheritance Structures may cause many opportunities for faults due to unanticipated bindings or misinterpretation of correct usage.
- Messages: Interface Programming Errors are a leading cause of faults in procedural languages. OO programs typically have many small components and therefore more interfaces. Interface errors are more likely, other things being equal.

6.2 Effects Of OO On Testing

Encapsulation The packaging of classes means that a tester knows exactly what code will be manipulating what data structures, and what code is prevented from access via protection. However, encapsulation is an obstacle to testing – the test code cannot write or read the attributes directly. So a test may be unable to directly set the values of attributes which are inputs to a method, or check the values of attributes which are outputs.

Inheritance A subclass inherits the abilities of a class (the attributes and methods) and the responsibilities (specification), providing a well defined way of testing inheritance. Constructors are hard to test, yet a tester needs to make sure that subclasses correctly initialize inherited attributes. Also, testing an over-ridden method may be difficult: as it represents a change to the original algorithm or functionality, it may not be possible to reuse the superclass tests.

Polymorphism This can result in hard to understand code, making testing difficult. For example, minor changes in the superclass may cause a subclass to fail. Or, more difficult to find, minor changes in a subclass may cause a superclass to fail. Also, the tester needs to make sure that the correct method is called for all the possible polymorphic objects.

Message Sequence And State Object state is changed by messages, which provides many possibilities for faults (corrupted state, an object in an incompatible state for a message) which need to be tested. However, state faults are hard to observe, and message sequences are not easy to track.

6.3 Object Oriented Testing Models

Testing object-oriented software can be approached at five different levels, each with its own test "model".

6.3.1 Conventional Models

Methods are tested in "class-context". This means that rather than just passing arguments to a method, and checking the return value, it may also be necessary to call different methods to set and get attributes which are used as inputs and outputs to the method under test. White-box and Black-box unit testing techniques as already discussed can be applied to these tests.

Also, many of the White-Box principles can be extended for an object-oriented environment. For example, d-u pairs can be used to derive test cases within a method, between methods at the line level (every individual d- and u-usage of an attribute is considered), or between methods at the method level (every method is considered as a d-usage and/or u-usage).

6.3.2 Combinational Models

Where the responses of a class (i.e. of the methods in a class) are different depending on the combination of input values, and not the ordering of method calls, then Combinational Testing is indicated. This means that the effects of different combinations of input parameter values on the class are tested for. The overall response of the class can be described through Truth-Tables. The "Causes" will be based on the values of input parameters to "setter" and "action" methods; the effects will be based on the return values from "getter" methods. Test Cases are then based on the rules.

The purpose of using combinational models is to ensure that the class works properly under different combinations of inputs.

6.3.3 State Machine Models

State-machines are not limited to object-oriented software – much real-time, control, and communications software is heavily based on state tables. However, all classes with attributes are state dependent, so state-based testing is of direct application to them. In UML, Harel state-charts are used to specify this behavior.

In many cases the strategy used for classes is *all transitions*. Tests need to be designed to make sure that every transition from every state moves the object into the correct next state. A way is needed to find out what state the object is in: the simplest way is to provide symbolic names for the states, and provide a new test method to confirm whether an object is in the expected state.

The purpose of state-machine, or sequential, testing is to ensure that the software transitions through the state machine work correctly. It is not to ensure that every method works correctly in every state – that is the subject of different forms of testing.

6.3.4 Specification & Design Models

Object-oriented software is usually specified through a language, such as UML. Each item on each diagram has a meaning, and therefore represents a testable property of the system. For example, the multiplicity on a relationship between two classes in a class diagram shows the minimum and maximum number of possible object relationships. This can then be used with boundary value analysis to check that the software correctly matches the specification.

6.3.5 Built-In-Test

Because of encapsulation, which prevents external test software from accessing internal state or attributes, placing the test code inside the classes has particular benefits for object-oriented software. Most OO languages provide some support for this, and allow the checking to be turned on or off at compile-time or runtime. For example, Java provides assertions. These can be used to check that attributes have valid values (also referred to "class invariants") when a method is entered, that the inputs to a method are valid ("preconditions"), and that the outputs are correct ("postconditions").

Assertions do not replace the normal checking of input parameters. They do not remove the need for exception handling. And they do not remove the need to test! Their intended use is to prevent errors in the first place, to catch many simple faults, to help debugging as failures are found close to their faults, and also to provide a test "oracle" where the actual expected output values are not needed, just the relationship between them and the inputs.

6.4 Example

6.4.1 Class CarTax

Specification

- Class `CarTax` calculates the tax payable on a car based on its age and CO2 pollution rating
- `CarTax(int year)` - creates a new object, with the specified year of manufacture for the car, and the CO2 rating initialised to an invalid value. The year must be ≥1900 to be valid
- `void setCO2(int rating)` - sets the CO2 rating for the car (in milligrams per kilometre)[1]
- `int getCO2()` - returns the CO2 rating for the car
- `int getManufactureYear()` - returns the year of manufacture for the car
- `int calculateTax(boolean newSystem)` - calculates the annual tax payable as follows:
  - 3000 if not newSystem, and year of manufacture is ≤ 2010
  - co2rating*10 if newSystem, or year of manufacture > 2010
  - -1 if any of the inputs are invalid
- `int readAndZeroTax()` - returns the tax, and zeros the value

Source Code

```
class CarTax {

   private boolean valid;
   private int yearOfManufacture;
   private int co2Pollution;
   private int tax;

   CarTax(int year) {
      if (year>=1900) {
         valid = true;
         yearOfManufacture = year;
      } else
         valid = false;
      co2Pollution = -1000;
      tax = -1;
   }

   public void setCO2(int rating) {
      if (valid)
```

[1] Note that the internal implementation stores co2Pollution as 1000 times the rating, but this is not visible outside the class

```
            co2Pollution = rating * 1000;
        }

        public int getCO2() {
           return co2Pollution / 1000;
        }

        public int getManufactureYear() {
           if (valid)
              return yearOfManufacture;
           else
              return -1;
        }

        public int readAndZeroTax() {
           int temp = tax;
           tax = 0;
           return temp;
        }

        public int calculateTax(boolean newSystem) {
           if (valid)
              if ( (yearOfManufacture<=2010) ||
                   ( (yearOfManufacture>2010) && (newSystem) ) )
                 if (newSystem)
                    tax = 10 * (co2Pollution/1000);
                 else
                    tax = 3000;
           return tax;
        }

    }
```

6.4.2 Black-Box Testing in Class Context

Method `calculateTax()` is tested in class context as follows.

Partitions

The Equivalence Partitions for `calculateTax()` are as follows:

yearOfManufacture:

[Integer.MIN_VALUE..1899]
[1900..2010][2011..Integer.MAX_VALUE]

newSystem: [false][true]

co2Rating: [Integer.MIN_VALUE..-1][0..Integer.MAX_VALUE]

return value:

[Integer.MIN_VALUE..-2][-1]
[0..Integer.MAX_VALUE][3000]

Note: this is an example of *overlapping* partitions, discussed in more detail in Section 10.10. As this is a return value, it is treated as two independent partitions.

Test Cases

Based on these partitions, the following Test Cases are derived – note that impossible Equivalence Partitions are not included as possible Test Cases.

Table 6.1: EP Test Cases in Class Context for CarTax

Case	Parameter	Range	Test
1*	yearOfManufacture	Integer.MIN_VALUE..1899	3
2		1900..2010	1
3		2011..Integer.MAX_VALUE	2
4	newSystem	false	1
5		true	2
6*	co2Rating	Integer.MIN_VALUE..-1	4
7		0..Integer.MAX_VALUE	1
8*	Return Value	-1	3
9		0..Integer.MAX_VALUE	2
10		3000	1

Test Data

The Test Data is then specified as follows in Table 6.2. Note that for OOT, the format of the Test Data must be changed to include the different methods to be called, and the order in which the methods must be called. The return value from each non-void method should be checked against its expected value, but any other 'side-effects' (such as setting attribute values) are not explicitly checked – the full operation of each method is checked in separate tests.

Table 6.2: EP Tests in Class Context for CarTax

ID	Test Cases Covered	Inputs	Expected Output
T001	EP2,EP4,EP7,EP10	CarTax(1990) setCO2(150) calculateTax(false)	non-null 3000
T002	EP3,EP5,EP7,EP9	CarTax(2050) setCO2(150) calculateTax(true)	non-null 1500
T003	EP1,EP5,EP7,EP8	CarTax(1500) setCO2(150) calculateTax(true)	non-null -1
T004	EP3,EP5,EP6,EP8	CarTax(2050) setCO2(-10) calculateTax(true)	non-null -1

The tests should be implemented as follows:

- For each test the methods should be called in the order specified.
- For each method call, the parameter values should be as shown.
- For each method call, the return value (if non-void) should be compared with the expected output specified.
- The test fails if any return value is not equal to the expected output.
- The test passes if all return values are equal to the expected output.

Note: if the constructor raises an exception on invalid inputs, then the test code must catch that exception and fail the test.

6.4.3 White-Box Testing in Class Context

All of the previously described White-Box Testing techniques can be applied for each method in the class. The major difference is that often input and output parameters are set by calling methods other than the one under test.

In addition, some of the techniques can be extended from the method level to the class level. The most significant of these is class-wide d-u pair testing.

OO d-u pairs

As for conventional d-u pair testing, the first task is to identify all of the definitions and uses of each parameter and local variable within each method. An extension, for class-context, is to also identify the definitions and uses of each attribute within each method.

The uses and definitions can be identified at two different levels: at the line level (e.g. by source code line number), or at the method level (e.g. just by the name of the method). D-u pair testing can then be carried out at either of these levels:

Line-Level For each attribute, determine definitions and uses across all the methods in a class (by line number). Then determine data that will exercise all the possible du-paths within the class – that is execute the D-line-of-code followed by the U-line-of-code.

Method-Level Determine definitions and uses across all the methods in a class (by method) Then determine data that will exercise all the possible du-paths within a class – that is execute the D-method followed by the U-method.

Testing at the method level is much faster, but less comprehensive. Testing at the line level is much more comprehensive, but very time consuming.

D-U Pairs and Test Cases

Taking attribute co2Pollution as an example, and testing in class context at the method level, the d-u pairs are shown in Table 6.3. Note that Java will always set an `int` to the default value of 0.

Table 6.3: Method-level d-u pairs for co2Pollution

d-u pair	D	U	Test
DU1	CarTax()	getCO2()	5
DU2	CarTax()	calculateTax()	5
DU3	setCO2()	getCO2()	6
DU4	setCO2()	calculateTax()	6

Each possible d-u pair is a Test Case. At the method-level, all d-u pairs are possible (as methods can be called in any order). At the line-level, the code must be analysed to find the possible d-u paths.

Test Data

This is shown in Table 6.4. Note that calling getCO2() between `CarTax()` and `calculateTax()` does not interfere with the d-u pair DU2 as it is a *use* not a *definition*.

Table 6.4: DUP Tests for co2Pollution

ID	Cases	Inputs	Expected Output
T005	DU1,DU2	CarTax(1990) getCO2() calculateTax(false)	non-null -1 -1
T006	DU3,DU4	CarTax(2050) setCO2(150) getCO2() calculateTax(true)	non-null 150 1500

Comment

Whether it is worth the effort required for d-u pair testing is an open research question. Tool support, both to generate the test data and to measure the coverage, is also the subject of active research.

6.4.4 Combinational Testing

Applied at the class level, combinational testing is useful to test the combinations of interactions between the methods and their input parameters. Instead of testing a single method, this provides a technique to identify combinations across multiple methods. The following example shows a subset of causes and effects, related to the return value of the `calculateTax()` method, using *pairwise* testing of newSystem and year (discussed further in Section 10.12) to keep the Truth Table small.

Note that due to the potential size of class-wide Truth Tables, two techniques can be employed to keep them manageable. The first is to exclude *error* causes – as they need to be tested independently anyway due to *error hiding*. The second is to use multiple tables – shown here in the use of pairwise testing, instead of a full Truth Table.

Causes and Effects

- **Causes**
  - year<=2010
  - newSystem
- **Effects**
  - calculateTax()==3000
  - calculateTax()==10*co2
  - calculateTax()==-1

Truth Table

The Truth Table, showing the rules specifying the interaction between the methods in class CarTax, is shown in Table 6.5.

Table 6.5: Truth Table for CarTax

	Rules		
	1	2	3
Causes			
year<=2010	*	T	F
newSystem	T	F	F
Effects			
calculateTax()==3000	*	T	*
calculateTax()==10*co2	T	*	T
calculateTax()==-1	F	F	F
	7	8	9
	Tests		

Notes:

- Rule 2 has a "Don't Care" condition for the effect "return value==10*co2". The value *may* be 10*co2 (depending in the input value for co2Rating selected in the test), but it need not be. So this effect can not be specified or tested for true/false.
- The same reasoning holds for the "Don't Care"s in the effects for rules 1 and 3.
- There are no error cases, so the return value is never -1.

Test Cases

As for conventional testing, each rule is a Test Case.

Test Data

Table 6.6: TT Tests for CarTax

ID	Test Cases Covered	Inputs	Expected Output
T007	Rule-1	CarTax(1990) setCO2(150) calculateTax(true)	non-null 1500
T008	Rule-2	CarTax(1990) setCO2(150) calculateTax(false)	non-null 3000
T009	Rule-3	CarTax(2012) setCO2(150) calculateTax(false)	non-null 1500

Note that setCO2() must be called to set a valid co2 rating for the tests to work, even though combinations of this value are not being tested in this test.

For complete testing, separate tables for year and rating, and for newSystem and rating are also needed (not shown here). Or a larger Truth Table could be used, incorporating the combinations of all three parameters.

6.4.5 State-Machine Testing

This is based on testing the behaviour of a class to sequences of inputs, or its state. In UML, this is defined in a UML State Diagram. If this does not exist, it can often be generated based on the specifications for the methods.

States

A state-diagram for class CarTax is provided in Figure 6.2.

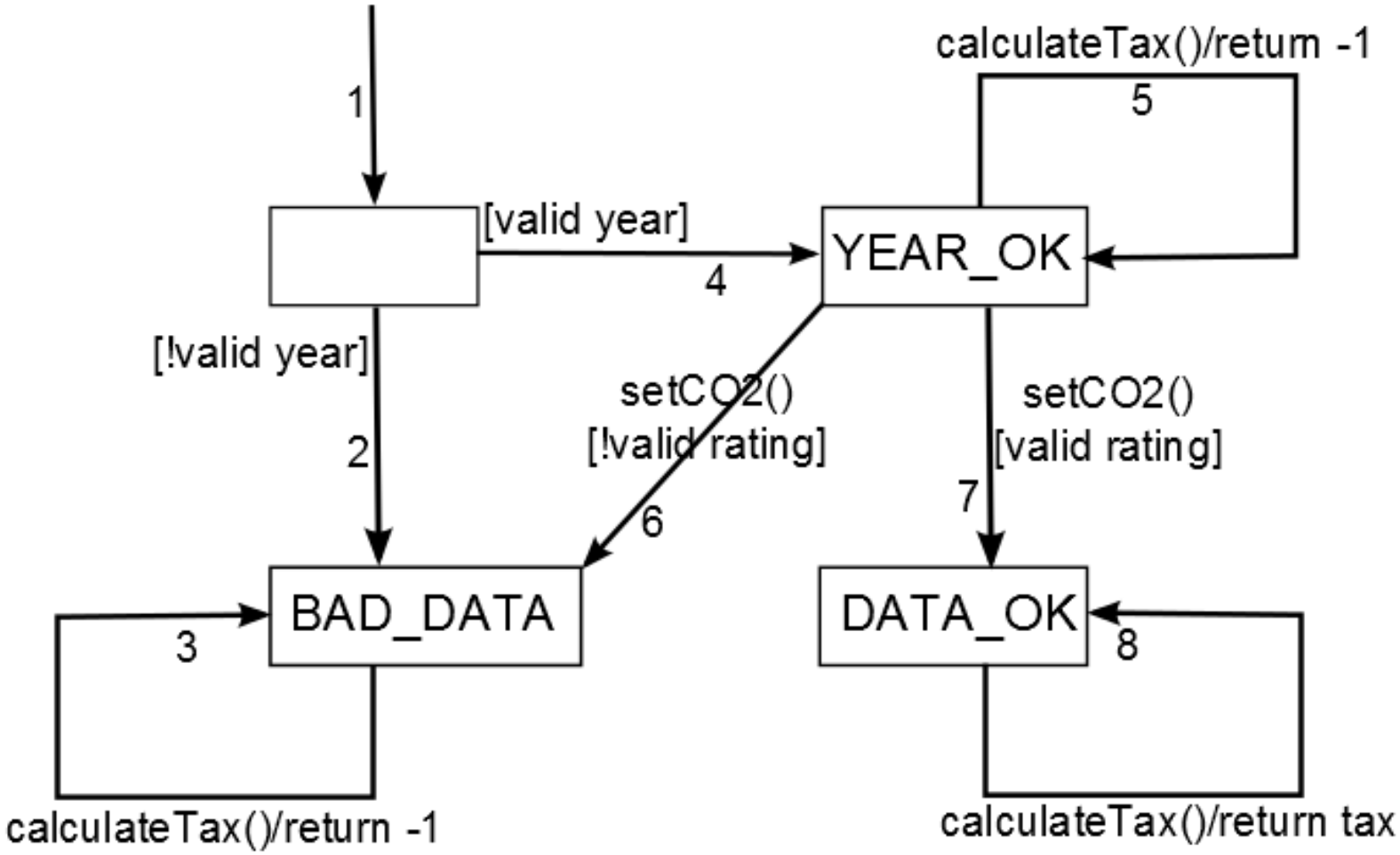


Figure 6.2: State Diagram for CarTax

Test Cases

Selecting the "every transition" test strategy, each numbered transition is a Test Case, as shown in Table 6.7.

Table 6.7: State Transitions for CarTax

Case	Transition	Test
S1	1	10
S2	2	10
S3	3	10
S4	4	11
S5	5	11
S6	6	11
S7	7	12
S8	8	12

Test Data

By reviewing the state diagram, data is selected to force CarTax() to transition through as many possible states per test. In this case, a minimum of 3 tests are required to do this.

Note that, in the test data shown in the following table, an additional call to calculateTax() is used after the final transition being tested to determine (as best as possible) that the system ends in the correct final state.

As there is no direct external visibility of the state, existing methods must be called to try and identify what state the system ends up in.

Table 6.8: State Tests for CarTax

ID	Test Cases Covered	Inputs	Expected Output
T010	S1,S2,S3	CarTax(1800) calculateTax(false) calculateTax(false)	non-null -1 -1
T011	S1,S4,S5,S6,S3	CarTax(2050) calculateTax(true) setCO2(-10) calculateTax(true) calculateTax(true)	non-null -1 -1 -1
T012	S1,S4,S7,S8	CarTax(2050) setCO2(150) calculateTax(true) calculateTax(true)	non-null 1500 1500

Note that each transition has been taken at least once.

6.4.6 Specification/Design Testing

Every item in the specification of an OO system is a potential source of test data. Using UML, this means that every symbol on every diagram can be used to derive test cases and test data! The Class Diagram is considered below.

Coverage Criteria

There are three coverage criteria associated with UML Class Diagrams:

- Generalisation Coverage
- Relationship/Multiplicity Coverage
- Attribute Coverage

Example

There are a very large number of such possible tests – and so just a representative sample is listed here for the UML Class Diagram:

- For each *Generalisation* relationship, test that the subclass satisfies the specification of the superclass
- For each *Association* relationship, test the cardinality for multiplicity, update, delete, missing, and wrong number of relationships
- For each *Aggregation* relationship, test independent creation/destruction of the container and component – the container should be able to exist independently from the components
- For each *Composition* relationship, test independent creation/destruction of the container and component – the container should not be able to exist independently from the components
- For each *Attribute*, ensure that it has taken on every possible value (for example, using Equivalence Partitioning or Boundary Values)

As an example, consider the Class Diagram shown in Figure 6.3.

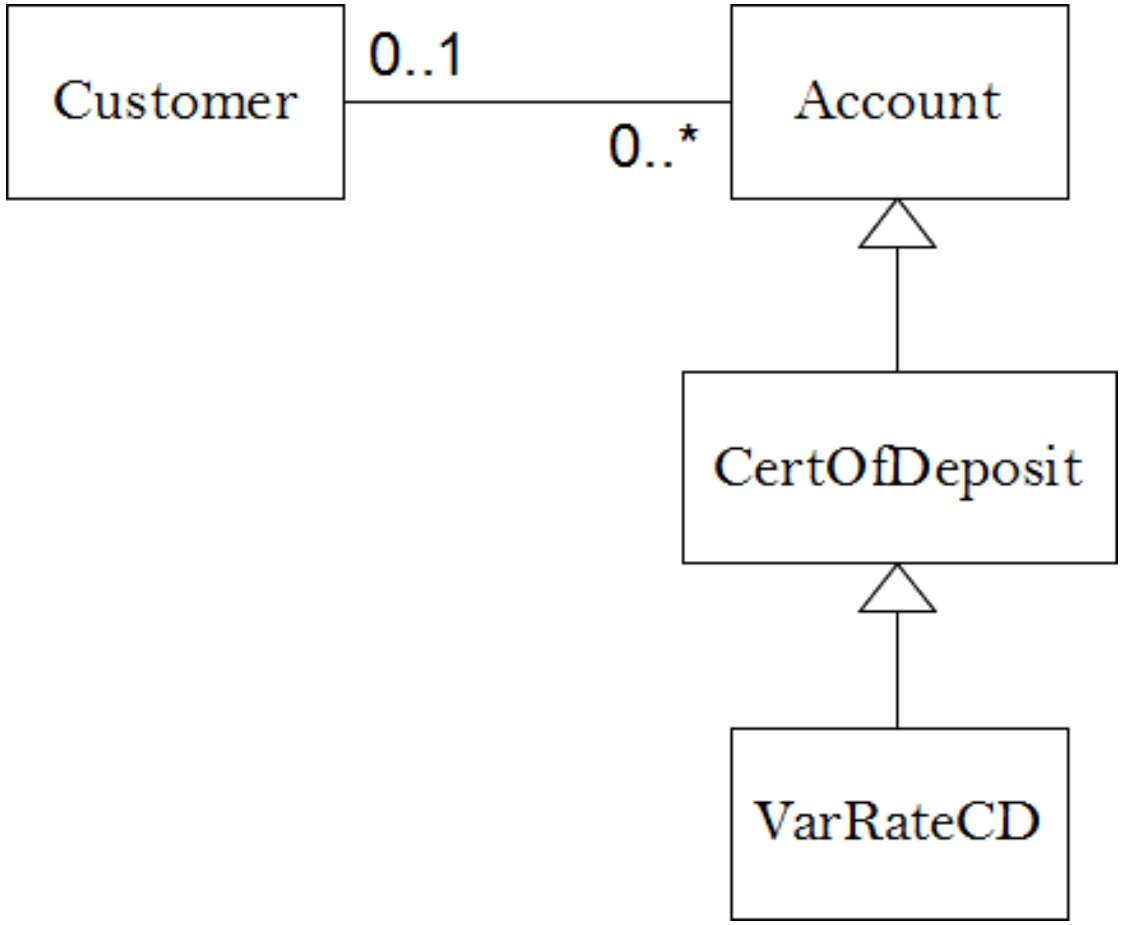


Figure 6.3: Class Diagram for Banking System

Candidate Test Cases (Generalisation Coverage) are:

- Test class CertOfDeposit with the Account tests
- Test class VarRateCD with the CertOfDeposit tests

Candidate Test Cases (Relationship/Multiplicity Coverage) are:

- Try to create a Customer with 0, 1, 1000 Accounts
- Try to create an Account with 0,1,2 Customers
- Make sure that deleting an object of class Customer does not delete associated Accounts
- Make sure that deleting an Account does not delete the associated Customers

This Class Diagram shows no attributes, and so there are no Test Cases for Attribute Coverage.

The Test Cases are referred to as *candidates* as they may not all be possible. For example, the class may provide no methods to allow an account to be added (making it impossible to test the behaviour if a second accound is added).

6.4.7 Built-In Testing

As the internal attributes of CarTax are not visible externally (note how some of the internal representations are not identical to the external ones), assertions can be added to the code to provide Built-In Testing. Note that Java *assertions* can be turned off at compile time.

These ensure that the attributes have valid values at method entry (pre-conditions & class invariants) and method exit (post-conditions & class invariants).

A Java assertion is a Boolean expression that defines the necessary conditions for correct execution – it will raise an exception if the expression evaluates to false at runtime.

General uses:

- checking implementation-specific assumptions
- checking pre-conditions (at method entry)
- checking post-conditions (at method exit)
- checking invariants (for an object) - must be true at all times

As as example, consider the method withdraw as shown below.

```
withdraw(Money amount)
{
   // check inputs
   ...
   // withdraw the amount if the balance allows
   ...
}
```

An invariant for the class is that the balance should never go less than zero:

(balance≥ 0)

So an assertion can be added to make sure that this condition holds at method entry and exit.

Note that in the following section **balance** refers to the value of balance at the location of the assertion, and **balance'** refers to the value of balance at method entry.

The postcondition is that, if the requested amount is greater than the balance at entry, then the balance at exit is equal to the balance at entry. Otherwise, the balance at exit is decremented by the requested amount:

(balance' ≥ amount)&&(balance==(balance' - amount))

||

(balance' < amount)&&(balance==balance')

To implement this, the entry value of balance (balance') must be manually saved at the start of the method as shown in the following code.

```
withdraw(Money amount)
{
   // save balance'
   Money old_balance=new Money(balance);
   // check invariant
   assert(balance>=0);
   // check inputs
   ...
   // withdraw the amount if the balance allows
   ...
   // check postcondition
   if (old_balance>=amount)
      assert(balance==(old_balance-amount));
   else
      assert(balance==old_balance);
   // check invariant
   assert(balance>=0);
}
```

Note the use of `new Money()` to ensure that a copy of the balance is taken (not just a reference).

The method can then be tested as usual, and the assertion provides additional verification that the method is working properly (as it can access internal state/attributes that the test code cannot).

Assertions do not:

- Replace the normal checking of input parameters.
- Replace normal exception handling.
- Replace the need to test.
- Replace the need to document code.

If added systematically, assertions can be used with random test data to provide fully automated Random Testing. However, manual saving of the attributes at entry, for use in postconditions, adds a significant overhead, and is a source of additional faults. See the discussion in Section 10.8.

Chapter 7

Integration and System Testing

Integration and system testing often both take place over the same interface (the system interface). But it is important to note that the purpose of testing is different – and thus different test data is required to meet the needs of each. Integration test data is selected to ensure that the components or sub-systems of a system are working correctly together. Test cases will therefore explore different interactions between the components, and make sure the correct results are produced. System test data is selected to ensure that the system as a whole is working. Test cases will therefore explore the different inputs and combinations of inputs to the system to ensure that the system satisfies its specification.

7.1 Integration Testing

Integration Testing takes a number of forms. When combining units together on the same layer of the software application and then testing them as each is added is termed Incremental Integration Testing. This may also apply if units are connected in a particular way across the application. Alternatively, as layers of the application are being completed testing across the layers can be either "Top-Down Integration Testing", "Bottom-Up Integration Testing", or "Sandwich Testing".

7.1.1 Drivers and Stubs

When doing integration testing, by definition the software is incomplete. This often requires the tester to write temporary code to take the place of not yet written software. It is a goal to minimise the volume of this temporary software – but it is often necessary for testing. The relationships of drivers and stubs to the software under test (SUT) in shown in Figure 7.1.

A test *driver* calls the software under test, passing the test data as inputs. In manual testing, where the system interface has not been completed, a test driver is used in its place to provide the interface between the test user and the software under test. The input data can be passed by the user to the driver, which in turn calls the software under test. The results are then collected by the driver and displayed to the user for verification. In automated testing (see Chapter 8), the test driver (also referred to as the test harness) contains the code for the automated tests. Drivers can have varying levels of sophistication. It could be hard-coded to run through a fixed series of input values, read data from a prepared file, contain a suitable random value generator, or use other techniques to derive input test data. In this case the input data is already in the test source code (or an input data file), and the outputs from the software under test are automatically compared with the expected outputs by the driver.

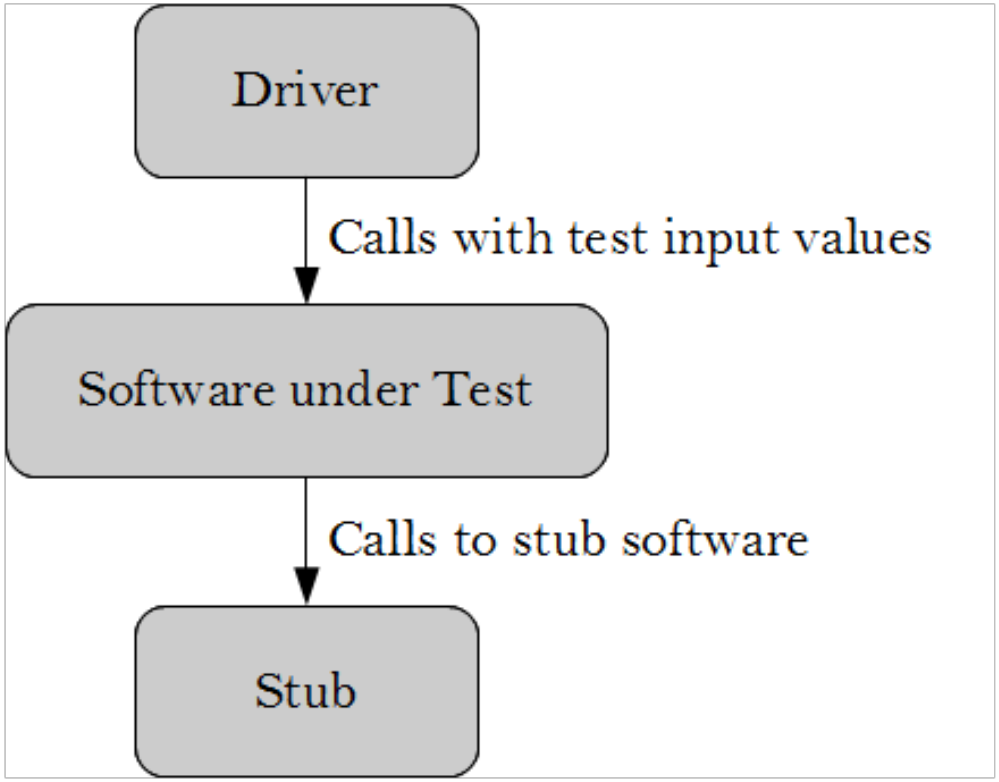


Figure 7.1: Test Drivers and Stubs

A *stub* is temporary or *dummy* software that is required by the software under test for it to operate properly. As this software has not yet been written, a throw-away version is developed to allow the testing to take place. The stub software has limited functionality - often it will provide fixed, or very limited, support for different values passed from the software under test. For example, it might only provide support for a limited set of input values, as required for the software under test to work with the test input data values.

An example of how a driver and stub might work in practice is shown in Figure 7.2. This is a simple GUI program to convert temperature from Celcius to Fahrenheit. The driver sends input data (temperature to be converted) via the windowing system to the user-interface, which is being tested. The core functionality to perform the conversion has not yet been coded, so a stub has been written to support the testing. The user-interface calls the stub to do the conversion, and then displays the result to the user. This result is picked up by the test driver, and checked for correctness. There are two test cases defined with the data values (0 and 100°C). The stub code only supports converting these two values.

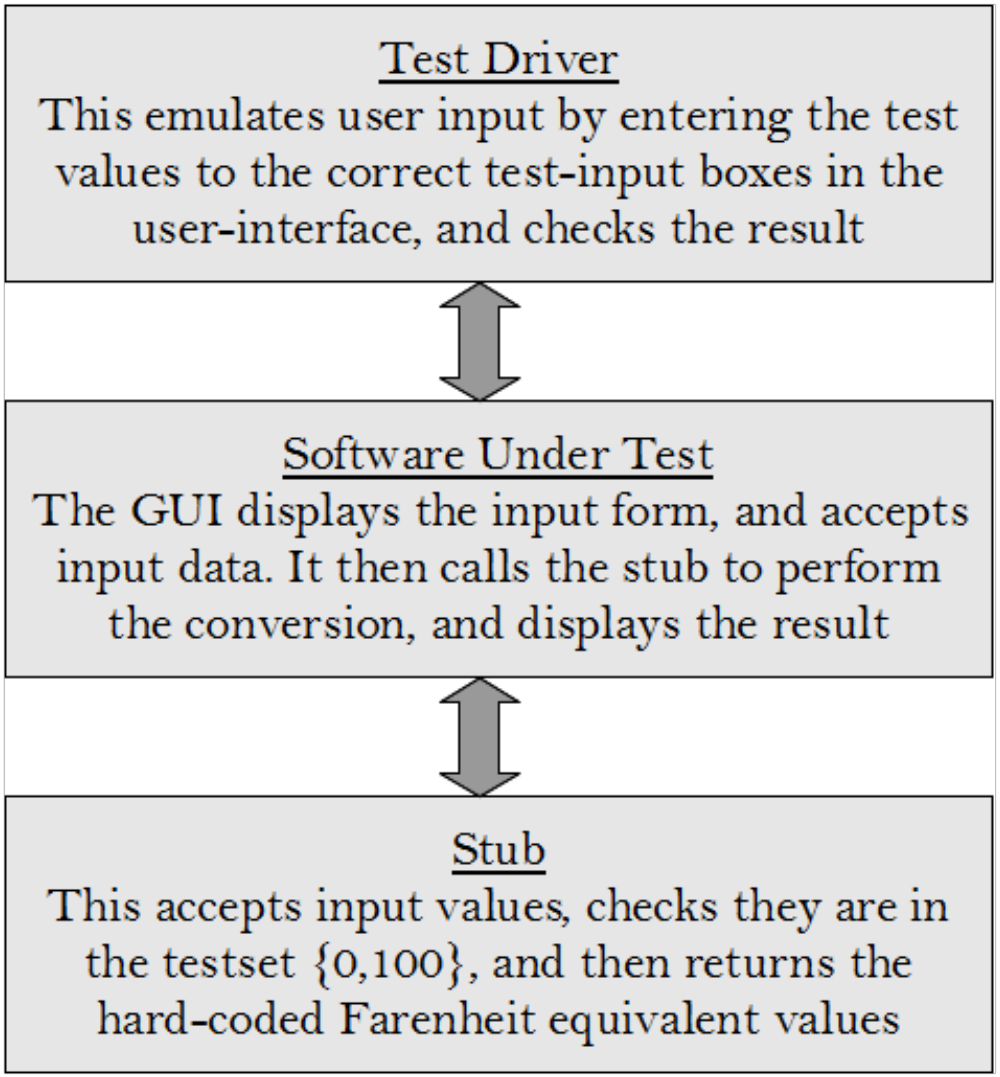


Figure 7.2: Drivers and Stubs Example

7.1.2 Top-Down Integration

If the top layer of the program has been produced first then Top-Down Integration testing is used. Figure 7.3 shows the direction of progress.

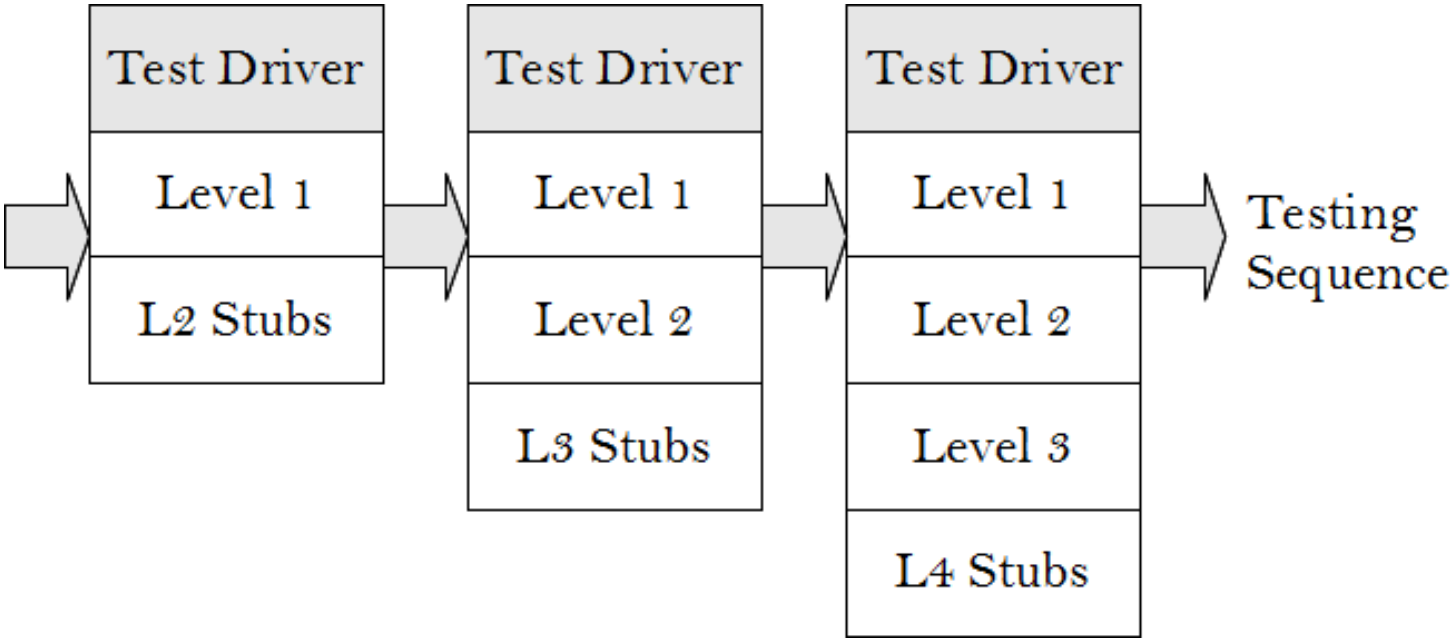


Figure 7.3: Top-down Integration Testing

Testing moves from the top layer of the program down to the bottom layers until it reaches the end. Stubs are used to simulate modules being called by the modules under test on a particular layer. A Stub can be as simple as a piece of code that will print a message to say that it was called to one that has to return specific values for specific situations. The strengths of using the Top-Down strategy is that the top layer provides an early outline of the overall program helping to find design errors early on and giving confidence to the team, and possibly the customer, that the design strategy is correct.

Weaknesses of the Top-Down Strategy include:

- The difficulty with designing stubs that provides a good emulation of the interactivity between different levels.
- If the lower levels are still being created while the upper level is regarded as complete, then sensible changes that could be made to the upper levels of the program that would improve its functioning may be ignored.
- When the lower layers are finally added the upper layers may need to be retested again.

7.1.3 Bottom-Up Integration

Bottom-Up Testing is the opposite to Top-Down Testing. It begins with the terminal modules in the program and drivers are used to emulate modules from the upper layers that call the lower modules being tested. Figure 7.4 shows an illustration of the procedure for Bottom-Up Testing.

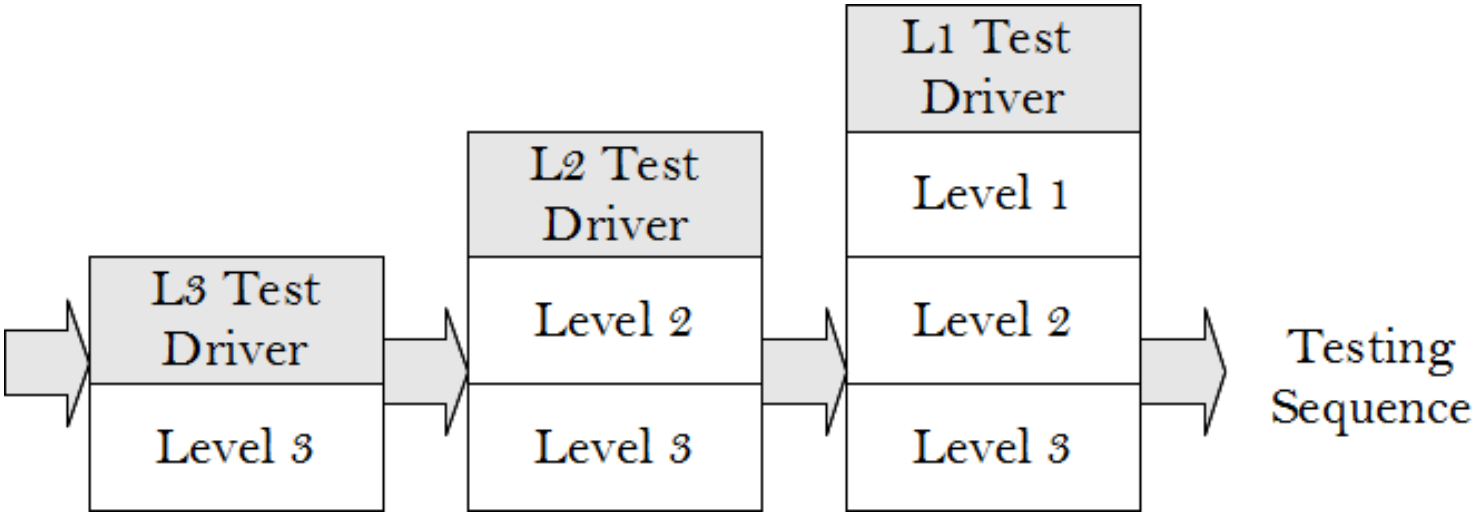


Figure 7.4: Bottom-Up Integration Testing

The strengths of Bottom-Up Testing overcome the disadvantages of Top-Down Testing. Additionally, drivers are easier to produce than stubs and because the tester is working upwards from the bottom layer, they have a more thorough understanding of the functioning of the lower layer modules and thus have a far better idea of how to create suitable tests for the upper layer modules.

The disadvantages of Bottom-Up Testing are:

- It is more difficult to imagine the working system until the upper layer modules are complete
- The important user interaction modules are only tested at the end
- Drivers must be produced

7.1.4 Sandwich Integration

Sandwich Testing is a hybrid of both of these. A target layer is defined in the middle of the program and testing is carried out from the top and bottom layers to converge at this target layer. It has the advantages that the top and bottom layers can be tested in parallel and can lower the need for stubs and drivers. However, it can be more complex to plan and selecting the "best" target layer can be difficult.

7.1.5 End-to-end User Functionality

As an alternative to these layered approaches, most modern development processes focus on developing increments of *end-to-end user functionality*. The software is integrated from the bottom up, but only small increments of user functionality are added across all the software layers (or *sideways*) – see Figure 7.5.

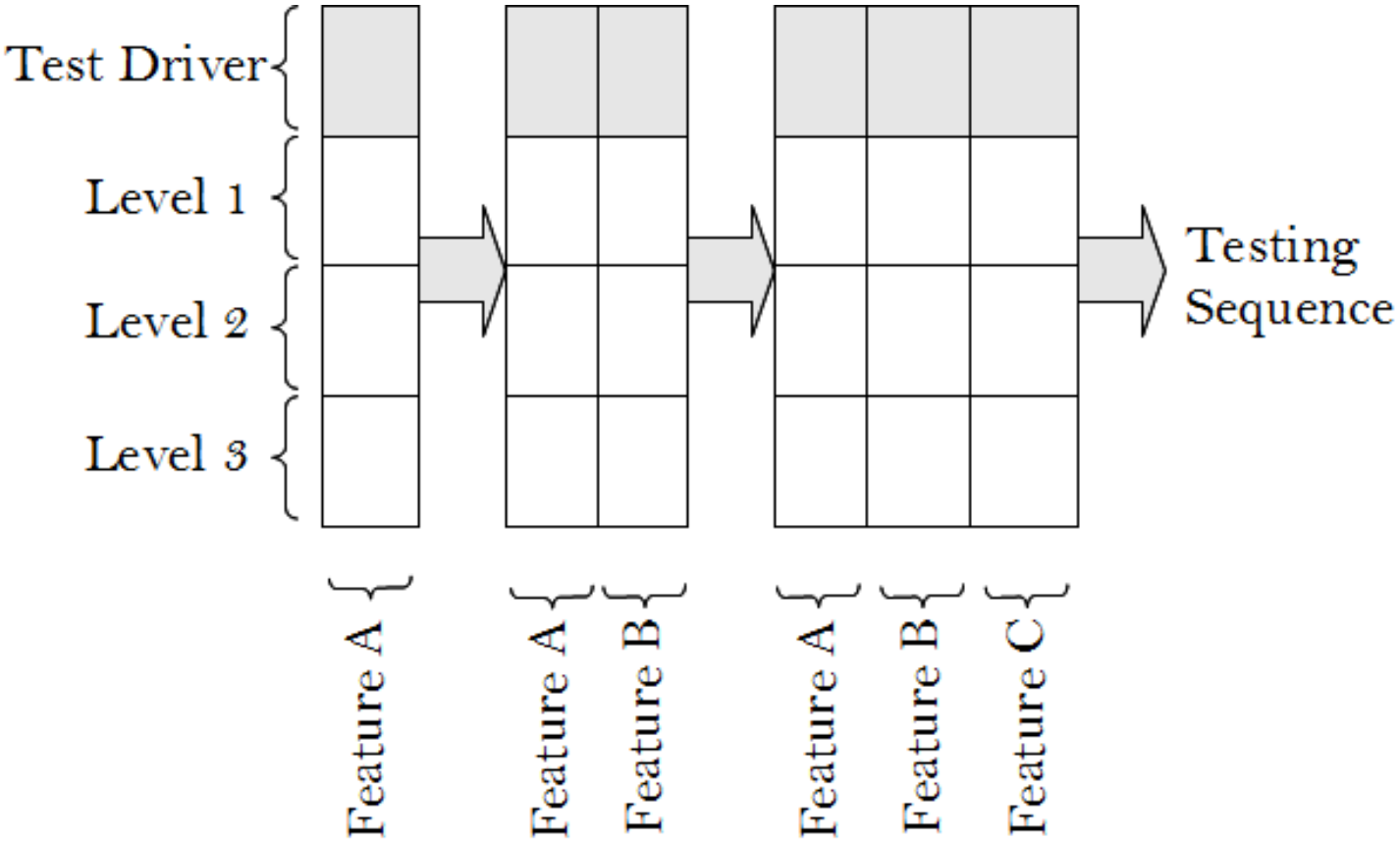


Figure 7.5: Sideways Integration Testing

This overcomes the problems associated with adding large amounts of functionality in Bottum-Up Testing. It also has the added advantage of allowing the end user to view and test the software as it is developed. This means that misinterpretations of the user's needs can be identified much earlier in the process, and that changes to functionality can be much more easily supported. The term *Agile* development is often used to describe this approach, and is often *Test-Driven*. This means that the tests for each addition of user functionality are developed first, so that as the work is integrated the tests (often developed in close contact with the customer) are ready to be executed immediately.

7.1.6 Test Cases

Test Cases for Integration Testing focus on exercising the interface between the components being integrated. This may include a significant subset of the Unit Tests and of the System Tests. But it will frequently also include additional tests that exercise internal specifications related to the interfaces.

For example, a simple application may consist of two classes: one for the GUI and one for the core functionality. The Unit Tests for the core functionality will ensure that it works correctly, and the Unit Tests for the GUI will have ensured that it responds to user actions by calling the correct GUI methods, and that results are correctly displayed to the user. The Integration Testing will verify that the GUI uses the core functionality correctly - that it calls the correct core methods with the correct parameters, and it collects and responds to returned values correctly. It will also ensure that the translation of data between the user format (for example, ASCII characters) used in the GUI, and the internal format (for example, binary numbers) used in the core is performed correctly.

7.1.7 Conclusion

Overall, when it comes to selecting an integration testing strategy for a software application a number of things should be considered. These include the number of stubs and drivers to be produced, the location of the critical modules in the system, which layers will be available for testing, and what order the layers are to be implemented in. Integration testing is difficult as it does require synchronization between testers and developers in the production, combination and testing of modules. Also, faults found during integration testing can be harder to resolve, and finally, if design features need to be changed they may result in conflicts with the customer.

7.2 System Testing

This means testing of the system as a whole and is almost invariably done using Black box tests. System testing can be broken down into a number of categories that may or may not be carried out on the product, as their necessity depends on the type of application it is. These are listed and explained below. It will be noticed that at times some of these testing activities overlap in terms of what they measure. This reflects that not all tests may be applicable to the one product.

7.2.1 System Test Categories

Conformance Testing Used to verify that the system conforms to a set of published standards. Many standards will have a published suite of conformance tests, or have a selected authority to run these tests. This is particularly important for communications software, as software system must correctly inter-operate with other implementations.

Documentation Testing Used to verify that the documentation (in printed form, online, help, or prompts) is sufficient for the software to be installed and operated efficiently. Typically a full installation is performed, and then the different system functions are executed, exactly as documented. Responses are checked against the documented responses.

Ergonomic Testing Used to verify the ease-of-use of the system. This can be either automated (verifying font sizes, information placement on the screen, use of colours, or the speed of progress by users of different experience levels), or manual (based on feedback forms completed by users).

Functional Testing As for Unit Testing, this is used to verify that the system behaves as specified. The interface used for testing is the *System Interface* which may consist of one or more of: the user interface, the network interface, dedicated hardware interfaces, etc.

Interoperability Testing Used to verify that the software can exchange and share information with other required software products. Typically, tests are run on all pairs of products to see that all the information that needs to be shared is transferred correctly. This is particularly important for communications software. Note that Conformance Testing is used to make sure that a system conforms to a standard; Interoperability Testing is used to make sure that it actually works with other conformant products. In theory, both are not needed: in practice, it has been found that conformance testing alone is not sufficient.

Performance/Load Testing Used to verify that the performance targets for the software have been met. Some of these tests can be static, but most are dynamic. These tests verify that metrics such as the configuration limits (static), response latency or maximum number of simultaneous users (dynamic) are measured and compared to the specified requirements.

Note: just measuring performance is not testing. To test performance, there must be a specification of what is required.

Portability Testing Used to verify that the software is portable to another operating environment. A selection of tests from the original platform are run in each new environment. The new platforms may be software platforms (such as 32-bit and 64-bit versions of the same operating system, or implementations by different vendors), or hardware platforms (such as mobile phones, tablets, laptops, workstations, and servers), or different environments (databases, networks, etc.).

Regression Testing Used to verify that any changes to the software (bug fixes, upgrades, new features, or other modifications to the original application) have not introduced new faults to previously working software. This is performed by re-executing some or all of the system tests again. See Section 9.6 for a discussion on using Repair Tests for Regression Testing. Minimising the number of tests to be run is an active research area (see Chapter 10).

Release Testing Used to verify, prior to distribution, that a product operates properly without interfering with applications or hardware. Typically System Tests are used to exercise the system being released, while the behaviour of other applications and the hardware is monitored for correct operation.

Stress Testing Used to verify failure behaviour and expose defects when the load exceeds the specified limits. Used to verify that the system degrades gracefully or not. This is one form of testing where the requirements may be somewhat vague, and verifying that the software has passed a test may require interpretation. Typically an excessive load is placed on the software, usually in terms of "input events", and the test is at least to ensure that the system does not crash. Ideally, the testing will verify that valid responses continue to be given.

Security Testing Used to verify that the system security features are not vulnerable to attack. The tests will deliberately attempt to break security measures. Examples would include impersonating another user, accessing protected files, accessing network ports, breaking encrypted storage or communications, or inserting unauthorised software.

Safety Testing Used to verify system operation under safety-critical conditions. The tests deliberately cause problems under controlled conditions and verify the system's response. Particularly necessary for life-critical, safety-critical, and mission-critical systems, such as medical device software or avionics.

7.2.2 System Level Functional Testing

In this section, the application of Black-Box Testing principles to system-level functional testing are described.

The key difference between Unit Testing and System Testing is that the test is run over the system interface, rather than the programming interface (except in the case of the development of a system which consists purely of libraries, where the only interface is the programming interface).

The Interface

System Functional Testing is performed over the system interface. This might take the form of one or more of the following: a command-line interface, a GUI interface, speech-based and gesture-based interfaces, web-interface, other network interfaces, or dedicated hardware interfaces to sensors and actuators for an embedded system.

The Test Environment

A generic test environment model for System Testing is shown in Figure 7.6. The Test Tool provides input (I/P) to the SUT, and receives output (O/P) from the SUT, over the system interface (I/F). In some cases the interface will be synchronous, where every input generates an output. In other cases the interface will be asynchronous, where an input may create a sequence of outputs over time, or the software may spontaneously generate outputs based on timers or other internal events.

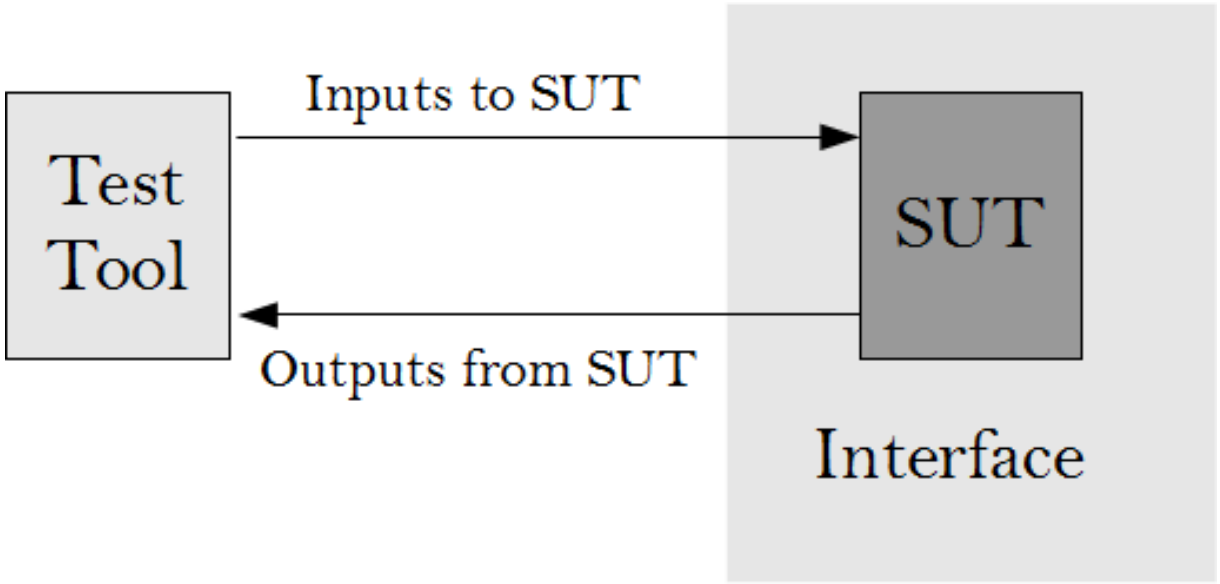


Figure 7.6: System Test Model

This generic model can be applied to different specific software interfaces as shown in Figure 7.7 for a GUI interface, Figure 7.8 for a network interface, and Figures 7.9 and 7.10 for a web-based interface. In Figure 7.9 the system is tested via a browser; and in Figure 7.10 the system is tested using HTTP directly over the network interface.

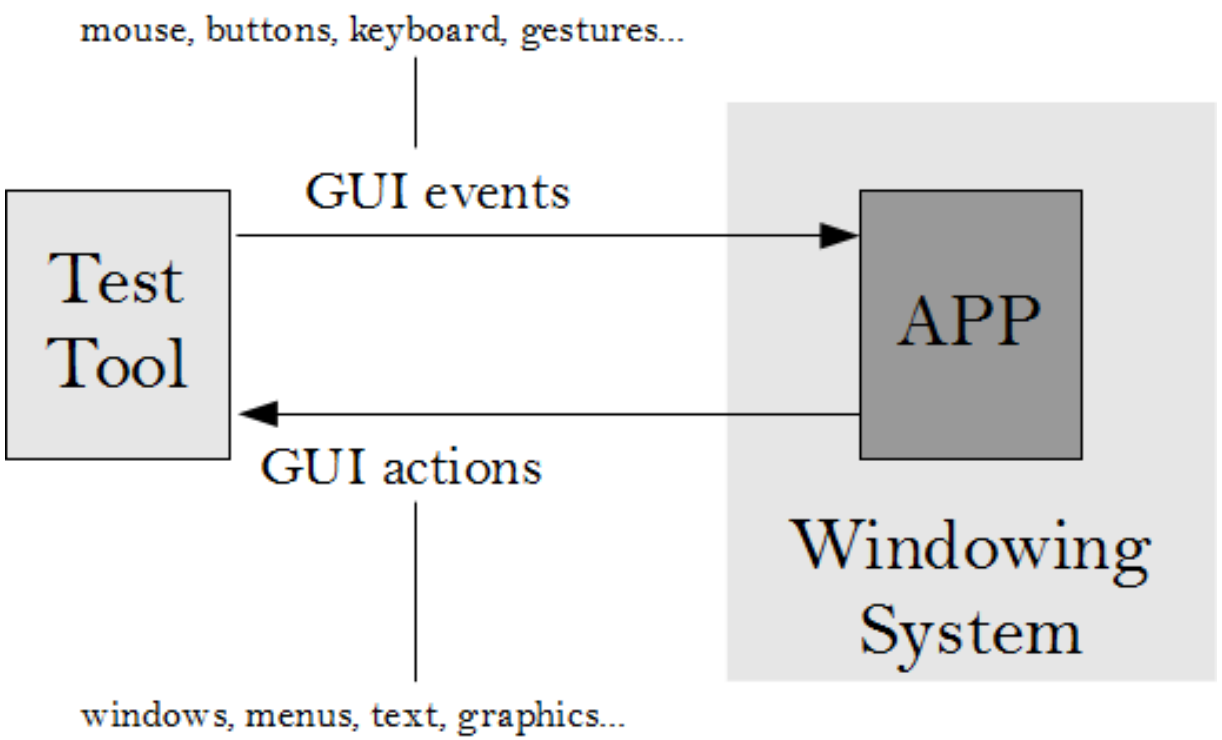


Figure 7.7: GUI Test Model

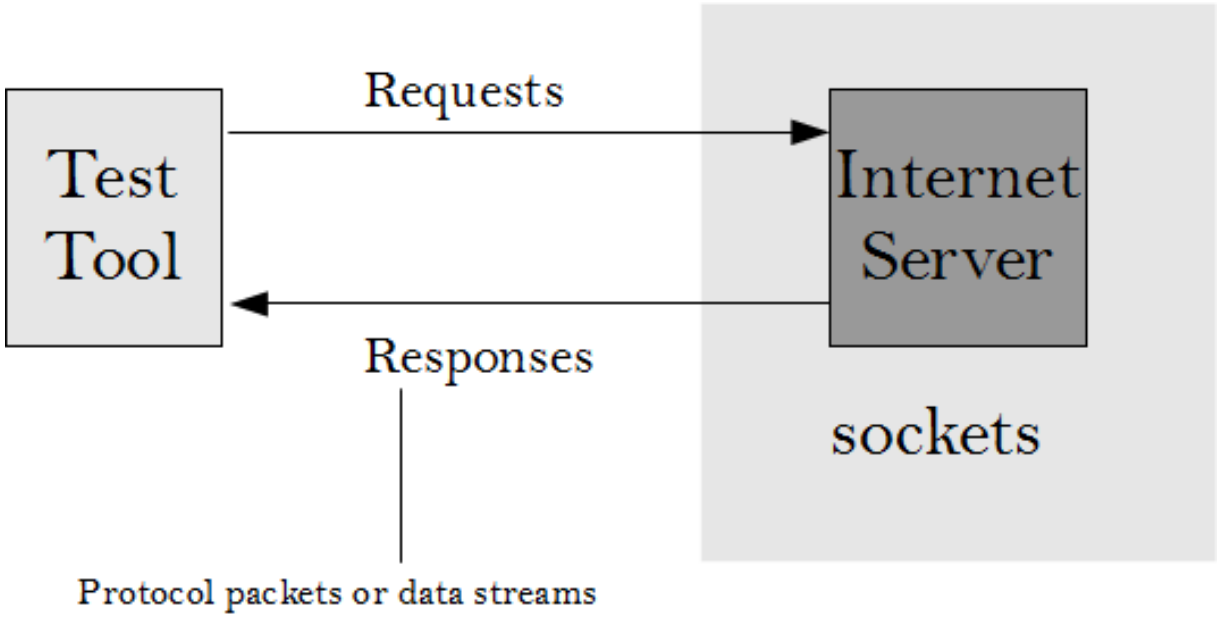


Figure 7.8: Network Test Model

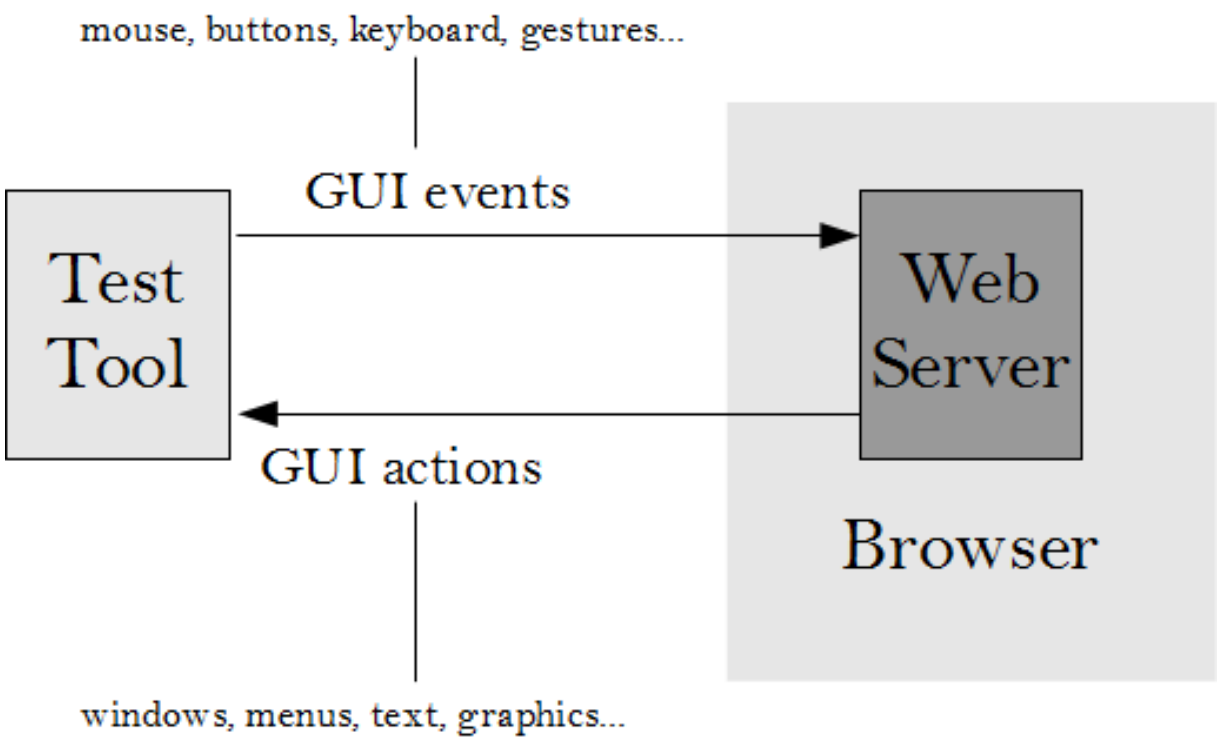


Figure 7.9: Browser Web Test Model

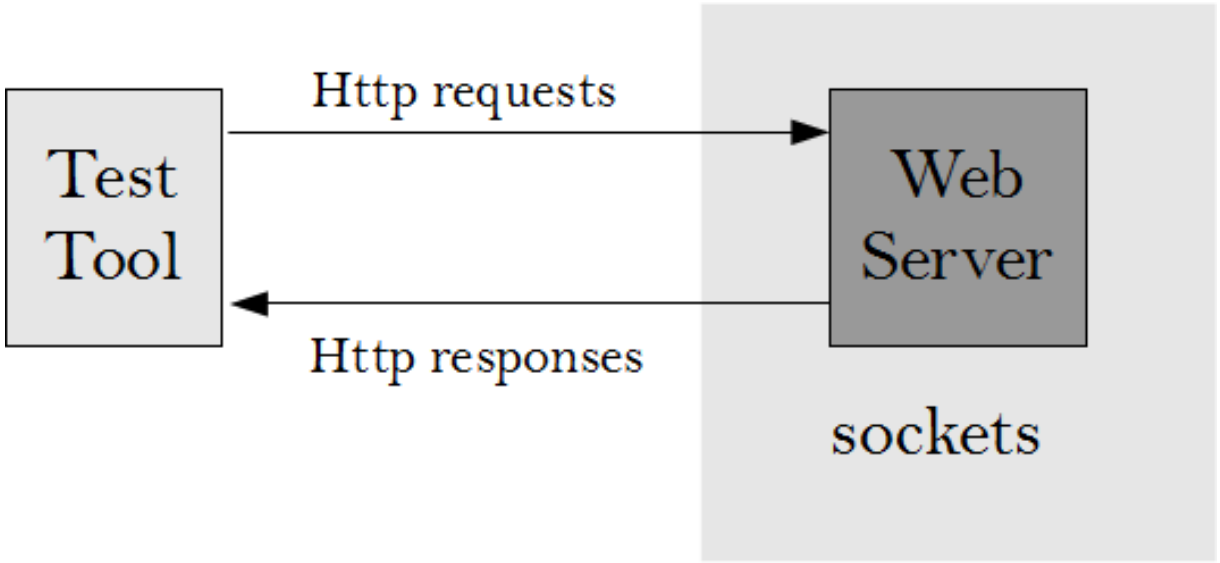


Figure 7.10: Direct Web Test Model

7.2.3 Test Cases

The selection of Test Cases is not as simple as for Unit Testing, as it depends on the particular interface being used for testing. But the general principles are the same as for Black-Box Testing: selecting a subset of all the possible input values that will provide reasonable assurance that the system operates correctly. All of the Black-Box Techniques described (Equivalence Partitions, Boundary Value Analysis, Truth Tables, and State-Based/Sequential Testing) can be applied to System Testing.

Application of Equivalence Partitions, Boundary Value Analysis, and Truth Tables is relatively straightforward. Based on the *Software Feature* being tested, Test Cases are selected using the technique being used, and then the tests specified by selecting specific values. The only difference is that instead of calling a method with the specified parameters, the data must be entered into the interface as appropriate, and then the results collected from the interface. Often, especially in the case of a GUI, the test will have to state how to get to the appropriate place in the interface to enter the data, and to get the responses (which may well be in different menus or windows of the application).

State-Based Testing can be used to test the interface itself. That is to make sure that it transitions through the different states correctly. For network-based protocols, the state transitions will be specified, and the techniques already described can be used with the data placed in network packets instead of parameters. For a GUI interface the technique can be used to ensure that the correct windows and menus appear (state) based on the user inputs (events).

7.2.4 GUI Example

From a testing viewpoint, a GUI consists of multiple *windows*, each containing various *elements*. The elements in turn can contain further elements. User input takes the form of input *events*, to which the system responds with *actions* which cause the results to be displayed to the user. A GUI invariably has a defined sequence of windows that can be displayed based on the user input events: for example, displaying a menu, displaying a sub-menu, displaying a new window on top of the existing window (perhaps to set preferences, or save work to a file).

System Testing of a GUI can take the following forms:

- Ensure the interface can be *navigated* correctly.
- Ensure the correct elements are on each screen.
- Verify the correct output *actions* occur for each input *event*.

Navigating the Interface Any of the described strategies for state-diagram coverage can be used. Normally *all transitions* is aimed for, ensuring that every user action which should cause a change in the user interface at every place causes that change correctly. An example would be (a) bringing up a pull-down menu, and (b) causing the menu to disappear.

Screen Elements If the screens are specified in detail, the system can be tested to ensure that they appear correctly. Some of the attributes that can be easily tested for are:

- Element type: output textbox, input textbox, button etc.
- Element placement: absolute position, relative position, other (automatic placement)
- Element values: initial, modified
- Element appearance: size, font, image, colour, etc.

Correct Behaviour As for Black-Box Unit Testing, aspects of the input parameters for each software feature form the Test Cases (using EP, BVA, Truth-Tables, etc.). Some additional factors to consider are:

- The inputs may be on a number of different windows or menus from the outputs.
- The expected outputs may be on a number of different windows.

The System

The example system is a simple interest calculator. There are two windows:

- The main window: this prompts for inputs and shows the result.
- Exit window: this confirms that the user wants to exit.

The specifications for these two windows is given graphically in Figures 7.11 and 7.12.

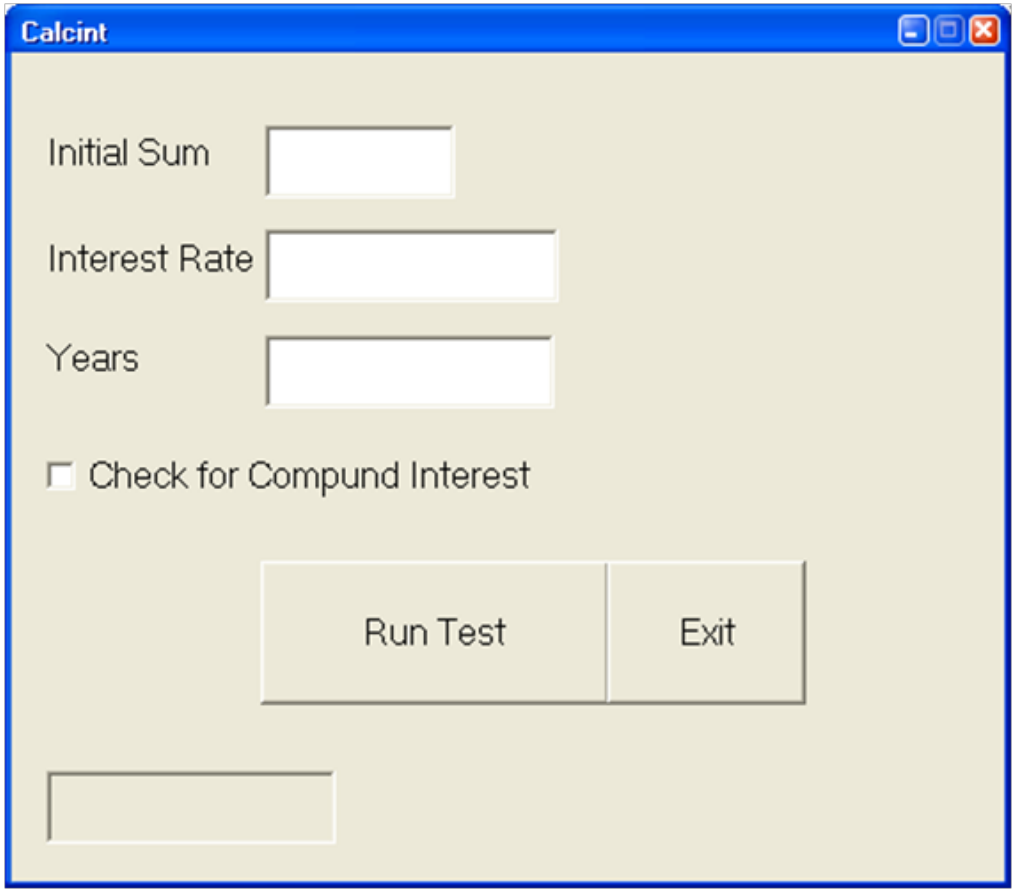


Figure 7.11: Calcint Main Window

Figure 7.12: Calcint Confirm Window

The interest calculation is defined as:

$$interest = \begin{cases} initial * (1 + rate) * years & \text{if not compound} \\ initial * (1 + rate)^{years} & \text{if compound} \end{cases}$$

calculated to the nearest euro.

The GUI *state diagram* is defined as shown in Figure 7.13. On entry, the *Calcint* window is displayed and the fields *Sum*, *Rate*, and *Years* are blanked. If the *Run Test* button is clicked, the *answer* is filled. If the *Exit* button is clicked, the *Confirm Exit* window is activated. If the *Cancel* button is clicked, the *Confirm Exit* window is deactivated, and focus returns to the *Calcint* window. If the *OK* button is pressed, the action *exit app* is taken to terminate the application.

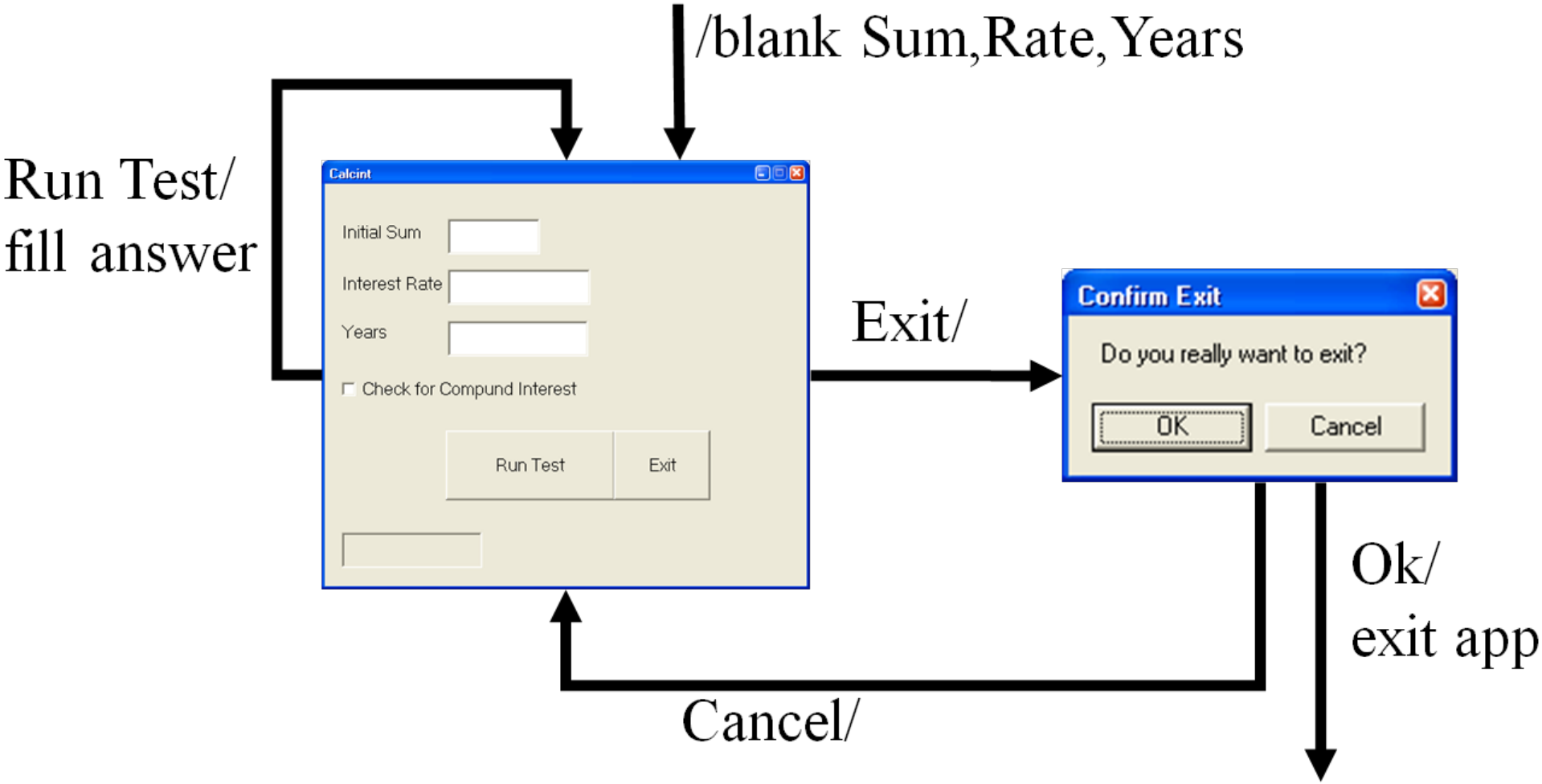


Figure 7.13: Calcint GUI State Diagram

Navigating The Interface

Using the *every transition* test strategy, each transition is a test case.

1. Entry transition
2. Run Test transition from Calcint window
3. Exit transition from Calcint window
4. Cancel transition from Confirm Exit window
5. OK transition from Confirm Exit window

This can be achieved in a single test as specified in Table 7.1. The inputs are the GUI events to drive the application, expressed in terms of components activated (here only button clicks are used). The Expected Outputs are the states of the system, expressed in terms of which window is active (for example: visible, on top, and having focus).

Notes:

- These tests do **not** test the functionality of the application – but that the navigation works correctly.
- The names of the GUI components may not be specified (they might, for example, be assigned automatically by the GUI design tool). In which case a GUI capture tool will have to be run to find the names.

Table 7.1: Navigation Tests for Calcint

ID	Test Cases Covered	Inputs	Expected Output
T001	1,2,3,4,5	Start application	Calcint window active
		Click Run Test	Calcint window active
		Click Exit	Confirm Exit window active
		Click Cancel	Calcint window active
		Click Exit	Confirm Exit window active
		Click OK	application exits

Appearance Tests

The exact details of the appearance tests will depend on how much detail has been specified for each window, and which windowing system being used (for example, Windows, X11, Java, etc.). In this section some example tests will be outlined to show the principles.

Every parameter for every component of every window is a test case. This includes some of the following representative parameters:

- x and y co-ordinates
- height and width
- contents (text, graphics)
- border width, style, and color
- background, foreground, and text colors
- font and font size

In this example, apart from the relative location of the text boxes and buttons, the specification states that the three text output boxes *Sum*, *Rate*, and *Years* should be initially blank, as shown in Figure 7.11. The tests for these are specified as follows in Table 7.2.

Table 7.2: Appearance Tests for Calcint

ID	Test Cases Covered	Inputs	Expected Output
T002	1,2,3,4,5	Start application	Calcint window active
		Sum.text()	empty string
		Rate.text()	empty string
		Years.text()	empty string

System Level Functional Testing

GUI-based system functional testing is very similar to OOT Black-Box Testing. In this case there is a single feature to test, which is simplified as it has all the inputs and outputs on the same window.

Using Equivalence Partitioning on the problem, the following are the *valid* test cases:

1. Initial Sum=100
2. Interest Rate=5
3. Years=10
4. Compound Interest=yes
5. Compound Interest=no

In practice, as for Unit Testing, the tests would begin with Equivalence Partitioning to ensure basic correctness, then move on to more complex techniques such as Boundary Value Analysis and Combinational Testing.

The tests for these test cases are specified in Table 7.3.

Table 7.3: Functional Tests for Calcint

ID	Test Cases Covered	Inputs	Expected Output
T003	1,2,3,4	Initial Sum=100	
		Interest Rate=5	
		Years=10	
		Compound Interest=unchecked	
		Run Test	answer=150
T004	1,2,3,5	Initial Sum=100	
		Interest Rate=5	
		Years=10	
		Compound Interest=checked	
		Run Test	answer=163

7.3 Field Testing and Acceptance Testing

Field Testing (commonly referred to as Alpha and Beta Testing) is used to verify that a system works correctly in a real operational environment. Alpha Testing is usually restricted in scope and carried out at the software developer's location. In Beta testing, the software is usually distributed for many volunteer users to run in their own environment. Beta testing is particularly good for identifying faults that occur during normal use - factors which might not have been considered by the development team may be identified and fixed prior to release of the product.

Some factors to consider in order to gain the maximum benefit from Beta Testing are:

- Making the test group as large and diverse as possible.
- Providing incentives for users to return reports.
- Keeping the user evaluations confidential.

Customer Acceptance Tests are used by a customer to ensure that the system that they ordered works properly. Typically they would be used prior to payment of a system, or they might be used in a comparative analysis of multiple tendered solutions. In some cases the tests, or a subset of the tests, are released to the software developer. In other cases, the tests are kept secret, and details only released of tests that fail.

Chapter 8

Software Test Automation

Manual software testing is slow, error-prone, and hard to repeat accurately. Yet software testing needs to be fast, accurate, and easy to repeat. It needs to be fast, so that it can be performed frequently without a time penalty. It needs to be accurate so that the test results can be relied on as a quality indicator. And it needs to be repeatable to allow for regression testing. Software test automation is therefore highly desirable.

Some of the testing tasks that can be automated relatively easily are:

- Automated execution of tests, or collections of tests
- Automated collection of results
- Automated evaluation of results
- Automated reports
- Automated measurement of test coverage

However, it can also be costly to automate testing as automation tools can be very specialized and their cost may even be higher than the cost of the system under test. Additionally, using the automation tool may be non-trivial, requiring an investment of time before the testers know how to use it properly. Ongoing developments are leading to new testing tools, and the tests and expertise developed may become obsolete.

There are essentially two types of test that can be easily automated: unit tests, and system tests. Unit tests are automated by writing code that calls the required methods with the specified test input data, and compares the actual output with the expected output. System tests are more complex to automate, as they depend on the details of the system interface.

In this chapter, automated unit testing is described in detail, using JUnit as an example test tool. The principles of automated system testing are then described in a more general manner.

8.1 Coverage Measurement

Most languages provide automated tools to measure at least statement coverage and branch coverage. For example, using Java and Eclipse[1], there are a number of plugins (such as EclEmma, CodeCover, and Cobertura) that provide this functionality. These tools can be used to verify that a White-Box test actually achieves the coverage it is

[1] The Eclipse Foundation

intended to achieve.

Coverage measurement tools can also be used to measure the component coverage of Black-Box tests. If low coverage is achieved, then there are untested components in the code, and achieving higher levels of coverage requires the use of White-Box testing techniques. Coverage measurements can be made for Unit Testing, Integration Testing, and System Testing.

8.1.1 Lazy Evaluation

Note that a particular problem arises with measuring Statement Coverage, which is *partial* execution of a line of source code. Apart from the simple example where there are two statements on the same line of source code, there is the more complex case where *lazy evaluation* may cause only some of the conditions in a complex decision to be executed.

In Java, the Conditional-Or (`||`) and Conditional-And (`&&`) operators are guaranteed to be evaluated left-to-right, and the right-hand operand is only evaluated if necessary[2].

Consider line 3 of `premium()`, as defined in Section 4.1.3:

```
((age<16) || (age>65) || (gender!='M' && gender!='F'))
```

If this is executed with the variable *age* having a value less than 16, then once the first condition `(age<16)` has been evaluated to true, the subsequent conditions `(age>65)` and `(gender!='M' && gender!='F')` are not evaluated, leading to only partial execution of the line.

Most coverage tools will identify that this has taken place, and use a separate color to indicate that a line has only been partially executed.

8.2 JUnit

JUnit[3] is presented here as a representative unit test tool. JUnit consists of a set of Java classes that automate the execution of unit tests, and collect and evaluate the results.

The best way to organize your JUnit class files, and to ensure that all the code is tested, is to have a test class for every program class. For example, class Demo in file Demo.java would have a test DemoTest in file DemoTest.java. You can use any naming convention you like – but be consistent. The accepted convention is to add the word "Test" either before or after your classname (as shown in the example above).

In the test class, the tests are implemented as methods. JUnit requires each test method to be identified using the Java "annotation" feature. The annotation for a JUnit test is "@Test" (see the example in Section 8.2.1). It is recommended that each test method test a single software feature for two reasons.

1. The method terminates as soon as a test fails. This means that any subsequent tests, in the same method, will not be executed. So, if you put multiple tests into a single method, the test results cannot indicate whether just one test failed, or a number of tests failed.

2. Ease of debugging – it is easier to find a fault if you know exactly which test failed.

The way to group multiple tests together is to put one test per method, and group the test methods into "test suites". To add to the flexibility, test suites may themselves be grouped into larger test suites.

[2] See The Java™ Language Specification, Third Edition

[3] JUnit.org

8.2.1 JUnit Example

The source code for a sample JUnit test is shown below:

```
import org.junit.Test;
import static org.junit.Assert.*;

Class DemoTest {
   @Test test1() {
      Demo d = new Demo();
      d.initialise(56);
      d.operate(44);
      assertEquals(d.result(),100);
   }
}
```

The key features of a JUnit test are as follows:

1. an object to test is created (in this case, of class Demo)
2. the object is initialised, or put into the right state for testing – in this example by calling initialise() with suitable data (*56* for example).
3. then the method under test is called – in this case method operate() with the input test data value – here *44* is used
4. the output data is collected – in this case by calling the result() method
5. the output value is compared with the expected output value by using the JUnit assertEquals() method – in this case with the expected output value *100*.

JUnit supports other assertion methods – the most commonly used ones are `assertEquals()`, `assertTrue()`, and `assertFalse()`. If the assertion passes, then execution continues to the next line in the method. If the assertion fails, then the test method terminates with an exception.

The JUuit tests are identified by a JUnit Test Runner which uses Java Reflection to find all the methods in the test class with the @Test annotation, and then calls them in turn, trapping exceptions, and keeping counters for the numbers of tests run, tests passed, and tests failed. JUnit comes with a default command-line test runner, but the easiest way to run JUnit tests is within an IDE, such as Eclipse, as described in the next section.

JUnit Test Suites

A *Test Suite* is a collection of tests (or a collection of Test Suites). In JUnit, the test class acts as a simple Test Suite - the JUnit test runner allows you to easily run all the tests annotated with `@Test` in the class.

You can also define your own Test Suites using the JUnit `TestSuite` class. This allows you to create suites as follows:

- create a suite of selected test methods using TestSuite.addTest()
- create a suite of selected test classes using the constructor TestSuite(Class[]), passing in an array of the required test classes.

Once a suite has been defined, it can be run using the `run()` method.

With this you can easily define and run suites for: all the tests, a subset of the tests for regression testing, a subset of the tests that failed last time, etc.

8.2.2 Test Documentation

When designing tests, the test documentation produced includes: analysis results, test case design, and test specifications. The test specifications include the actual test identifiers and data to be used in automating the test.

When automating software tests, there are two choices for where to store the test documentation: in separate files (perhaps using word processing and spreadsheet tools), or to store it as comments in the test programs. Experience shows that separate files invariably become out of date - so it is recommended in this text that tests are documented in the same file as they are implemented. An example of this, based on a problem in Chapter 4, is presented next.

Example

The following source code (SeatsTest.java) shows representative layout of a JUnit test for Partition Testing of seatsAvailable() as detailed previously. The layout of the test documentation must allow: (a) the tester to confirm that full coverage of the test cases has been achieved, (b) an external reviewer to check that the test designs are correct, and (c) easy addition of more tests.

```
[SeatsTest.java]
import org.junit.Test;
import static org.junit.Assert.*;
// Test Design Technique: Equivalence Partitions
// Test Author: Stephen Brown
// Version and Date: version 1.0.0, 13/4/2011
// Software under test: Seats.java, version 1.0.0
//
// [Test Cases] Partition {Test}
//    Inputs
//       freeSeats: [EP1] INT_MIN..-1                {3}
//                  [EP2] 0..seatsRequired           {1}
//                  [EP3] seatsRequired..totalSeats  {2}
//                  [EP4] totalSeats+1..INT_MAX      {4}
//       seatsRequired: [EP5] INT_MIN..0             {5}
//                      [EP6] 1..freeSeats           {2}
//                      [EP7] freeSeats+1..totalSeats{1}
//                      [EP8] totalSeats+1..INT_MAX  {6}
//    Outputs
//       return value: [EP9] true
//                     [EP10] false
// Test Data
//         Test Cases     Inputs                      Expected Outputs
//    ID   Covered        freeSeats   seatsRequired   return value
//
//     1   EP2,EP7,EP10     50          75            false
//     2   EP3,EP6,EP9      50          25            true
//     3   EP1*           -100          25            false
//     4   EP4*            200          25            false
//     5   EP5*             50        -100            false
//     6   EP8*             50         200            false
//
//   Note: * indicates an error test

class SeatsTest {
   @Test public void test1() {
      Seats s = new Seats(100);
      assertFalse( s.seatsAvailable(50,75) );
   }
   @Test public void test2() {
```

```
        Seats s = new Seats(100);
        assertTrue( s.seatsAvailable(50,25) );
    }
    @Test public void test3() {
        Seats s = new Seats(100);
        assertFalse( s.seatsAvailable(-100,25) );
    }
    @Test public void test4() {
        Seats s = new Seats(100);
        assertFalse( s.seatsAvailable(200,75) );
    }
    @Test public void test5() {
        Seats s = new Seats(100);
        assertFalse( s.seatsAvailable(50,-100) );
    }
    @Test public void test6() {
        Seats s = new Seats(100);
        assertFalse( s.seatsAvailable(50,200) );
    }
}
```

Note that all the tests within a file must have unique identifiers (here, they are numbered). Each test can then be uniquely identified using the filename and the identifier. It is important to include the version number of the class under test. If required, an edit history of the test file could also be kept (this could be automated under a version control system). There is no need to limit a file to one particular type of test: a test file might include tests derived using multiple techniques EP, BVA, TT, etc. Or, alternatively, each might be put into its own file. However, it would be usual to separate Black-Box and White-Box tests into separate files: the Black-Box tests remain valid, even if the implementation changes – but the White-Box tests do not.

8.3 JUnit Testing in an IDE

Eclipse is used as a representative IDE within which testing can be performed. Most IDE's either support unit testing, or have plugins to provide this feature.

The JUNIT plugin can be used with Eclipse to implement unit tests in Java. An example is given of how the Branch tests can be implemented for the problem covered in Section 2.1.

8.4 Regression/Inheritance Testing With JUnit

A subclass inherits the responsibilities of its superclass. In other words, whatever functionality a superclass has, the subclass should also have, along with additional features. So, when implementing subclasses, it is necessary to run the superclass tests against the subclass as a regression test. This verifies that the subclass works in "superclass context" before testing the new, subclass features.

You can cut-and-paste the superclass tests into a new class for the subclass, but this is cumbersome and error-prone. A better way is to inherit the superclass tests into a subclass test class! This requires the use of a "factory method" to return an object of the required class. An example is shown below for the test classes SuperClassTest and SubClassTest.

```
Class SuperClassTest {

   protected SuperClass factory(int x) {
      return new SuperClass(x);
   }

   @Test test_super_1() {
      SuperClass sut=factory(10);
      assertEquals(sut.getValue(),10);
   }

}

Class SubClassTest extends SuperClassTest {

   protected SubClass factory(int x) {
      return new SubClass(x);
   }

   @Test test_sub_1() {
      SubClass sut=factory(10);
      sut.zero();
      assertEquals(sut.getValue(),0);
   }

}
```

This code works as follows. First of all consider the JUnit test runner executing SuperClassTest as a JUnit test. A single annotated test, test_super_1 is found and executed.

- When test_super_1 runs, the call to factory() results in the method SuperClass.factory() being invoked, which returns an object of class SuperClass.
- The test test_super_1 is then run using a SuperClass object.

Now consider the test runner executing SubClassTest as a JUnit test. Two annotated tests are found, test_super_1 (inherited) and test_sub_1 (implemented in the class).

- When test_super_1 runs, the call to factory() results in the method SubClass.factory() being invoked, as it has overridden the inherited factory method – note that the method signatures must be identical. Thus factory returns an object of class SubClass. The test test_super_1 is then run using a SubClass object.

- When test_sub_1 runs, the call to factory() again results in the method SubClass.factory() being invoked, which returns an object of class SubClass. The test test_sub_1 is then run using a SubClass object.

This technique allows for any depth of inheritance to be supported, and all the inherited tests are run as well as the new tests for the new subclass.

8.5 Writing Your Own Test Runner

The standard JUnit and Eclipse test runners have limited functionality, in particular they do not save the results to a log file, which can be necessary when running large numbers of tests. Implementing your own test runner is very easy, you just have to do two things:

1. Define your own annotation
2. Write code to find and execute the annotated methods from a class

The following two code examples show how to do this. In file MyTest.java an "annotation" class MyTest is defined. The retention policy makes sure that the annotation is kept at runtime, the target specifies that this annotation can be used on a method, and the interface gives the name of the annotation (which must be the same as the name of the class file).

```
[MyTest.java]
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MyTest { }
```

In the following code example, file MyTestRunner.java, class MyTestRunner is defined. This uses *Java Reflection* to find the test methods (annotated with Test). Note that the input parameters to the program are the names of test classes to run - the class files must be on the CLASSPATH for this to work.

```
[MyTestRunner.java]
import java.lang.annotation.*;
import java.lang.reflect.*;
public class MyTestRunner
   private static void run(String tc){
      try {
         for (Method m:Class.forName(tc).getMethods())
            if (m.isAnnotationPresent(MyTest.class))
               m.invoke(null);
      } catch (Throwable ex) {
         // process exceptions here
      }
   }
   Public static void main(String args[]) {
      for (String s:args) {
         run(s);
      }
   }
}
```

The important method is `run()`, which accepts the name of a test class, and finds the actual class implementation using `forName()` – note: the class must be on your classpath for this to work. The list of methods in the class is found using a call to `getMethods()`. Using `isAnnotationPresent()` on each of these methods, allows the test methods to be identified. These are each called using `invoke()`.

The test runner can keep counters of the numbers of tests invoked, and count the test failures (in the exception handling code marked "// process exceptions here"). In addition, the name of the test class being run, the name of the test methods invoked, and the date and time of each test can be printed to a log file.

8.6 Mutation Testing

A number of tools are slowly becoming available to automate the process of mutation testing. The functions they perform are:

- Generate the mutants for a program (typically for a class).
- Execute existing tests against the mutants.
- Discard mutants where the output from the mutant is different to the output from the original code.
- Produce a report of mutants not identified by the tests.

One recent example is µJava (muJava) which automates mutation testing for Java classes, also available as a plugin for the Eclipse IDE (MuClipse). This supports an extended set of *method-level* and *class-level* mutations. The class-level mutations involve making small test changes to modify encapsulation, inheritance, polymorphism, and Java-specific features. The method-level mutations are similar to those discussed previously, making 11 changes to operators in 6 different categories (arithmetic, relational, conditional, shift, logical, and assignment).

8.7 Automated System Testing

Only a few examples are discussed here – but there are many more types of system interface, each of which may be required to be handled slightly differently.

The general architecture of an automated system test tool is shown in the previous chapter, in Figure 7.6. The test tool accesses the System Under Test (note: SUT is used for both System Under Test and Software Under Test) using the system interface to provide input, and to collect the output automatically.

Figure 7.7, also in the previous chapter, shows how this is implemented for a user application with a Graphical User Interface (GUI). The test tool inserts GUI events (such as mouse movements, mouse button clicks, and keyboard activity) into the application (APP) using the windowing system that the GUI supports. This might be Windows, the Java GUI libraries, X11 etc. depending on the application. The test tool then monitors the GUI actions (such as windows, menus, test and graphics drawn on the screen), and compares these outputs to the expected outputs.

A program with a command line interface (CLI) can also be system tested using automated test tools. These tend to be simpler that GUI-based tools, as all the input and output is in the form of text strings to and from the Java console. An example would be the Unix "expect" tool which can easily be used in scripts to form the basis of a CLI-based system test tool.

There are two ways that system test tools are run. One is using record-and-playback, where a test operator checks all the outputs manually the first time the test is run. Regression tests can then be run automatically by checking that exactly the same output is produced. The manual checking of results is very time consuming and error prone. Also, some fields, such as the date and time will change every time, and some parameterisation of the recorded output is required. Also the positioning, size, and appearance of windows can change, depending on the screen size, and the user preferences, which can cause problems using the record-playback approach.

The second way is to use the specification of all the system features, and write test scripts that provide the user inputs, and check the outputs. There are a large number of tools that do this, but they are all time consuming – both to configure and to program.

A frequently used integration of these approaches is to use a recording tool to capture one test in the form of a script, and then to modify this script for additional tests. This allows the names of the GUI components to be found automatically, while using the specification directly to produce the expected outputs.

An example harness for GUI system test is Abbot (which provides access to the GUI interface) and Costello (which provides GUI record-and-playback facilities)[4]. Abbot/Costello provides support for both *scripted* and *programmed* tests using Java.

An extract from an Abbot/Costello XML script is shown below – the key lines are:

- the `<action>` tags, which define
  - a mouse click (`"actionClick"`)
  - text entry (`"actionKeyString"`)
- the `<assert>` tag, which verifies the output using an assertion.

The identifiers "Length1", "Length2", "Length3", and "Result" are used to uniquely reference the four text fields used, and "Identify" to reference a button.
This test verifies that the input set {"10","10","10"} correctly produces the expected output "Equilateral".

```
<sequence>
 <action args="Length1,10" method="actionKeyString" />
 <action args="Length2,10" method="actionKeyString" />
 <action args="Length3,10" method="actionKeyString" />
 <action args="Identify" method="actionClick" />
</sequence>
<assert component="Result" method="getText"
                              value="Equilateral" />
```

The full script also includes a number of other tags, which are not shown in this extract: `<launch>` and `<terminate>` which run and terminate the application, and `<component>` which maps the names Length1, etc. to the actual Java Swing component identifiers.

Other commonly used automated system test tools include performance testing and stress/load testing. The tools tend to be customised to the interface - and are often customised to the application also.

[4]See http://abbot.sourceforge.net

Chapter 9

Testing in the Software Process

The actual activity of testing can be approached in two ways. The first is to wait until all the code has been written and then to test the finished product all at once. This is referred to as "Big Bang" development. On the surface, it is an attractive option to developers because testing activities do not hold back the progress towards completing the product. However, it is a very risky strategy as the likelihood that the product will work, or even be close to working can be very low, and is particularly dependent on program complexity and program size. Additionally, if tests do reveal faults in the program, it is much more difficult to identify their source. It is then necessary to search through the complete program to locate them.

A more modern approach is to test the software as it is being created. This is referred to as "incremental" development. Individual modules, or software features, are tested as they are written. This process continues as additional software increments are produced until the product is completed. While this may have the effect of slowing down the arrival of the final product, it should produce one that has fewer errors and that the development team will have much more confidence in.

Taking an incremental approach the various stages that the test team will perform have been named to reflect the activity. Figure 9.1 gives a timeline for the appearance of the different stages.

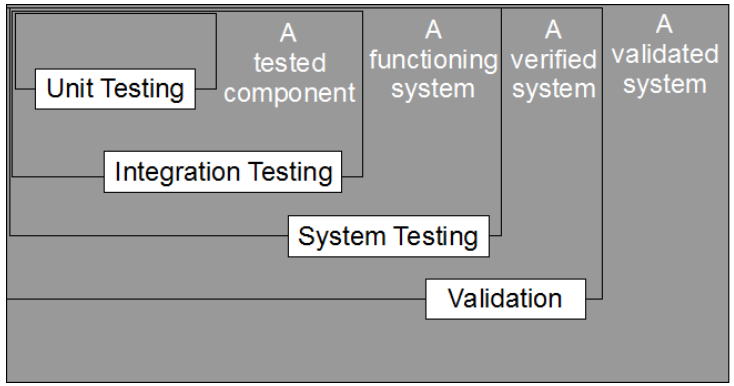


Figure 9.1: Timeline of the stages of incremental testing

This chapter describes the activities required to plan for software testing, and then examines how software testing fits into different styles of development process.

9.1 Test Planning

The time and resources required for software testing can be of a scale similar to those required for developing the software. The testing activity therefore needs to be planned and managed in a similar way. A typical Test Plan would include such information as:

- Items to be tested
- Tasks to be performed
- Responsibilities
- Schedules
- Required resources

The IEEE standard 892-1998 provides a formal framework within which a plan can be prepared[1]. Figure 9.2 shows some of the key documents it requires. In reality, only a very large or mission-critical project would use the full standard: most projects would use a subset, tailored for the project.

[1]This is due to be updated when 892-2008 is approved.

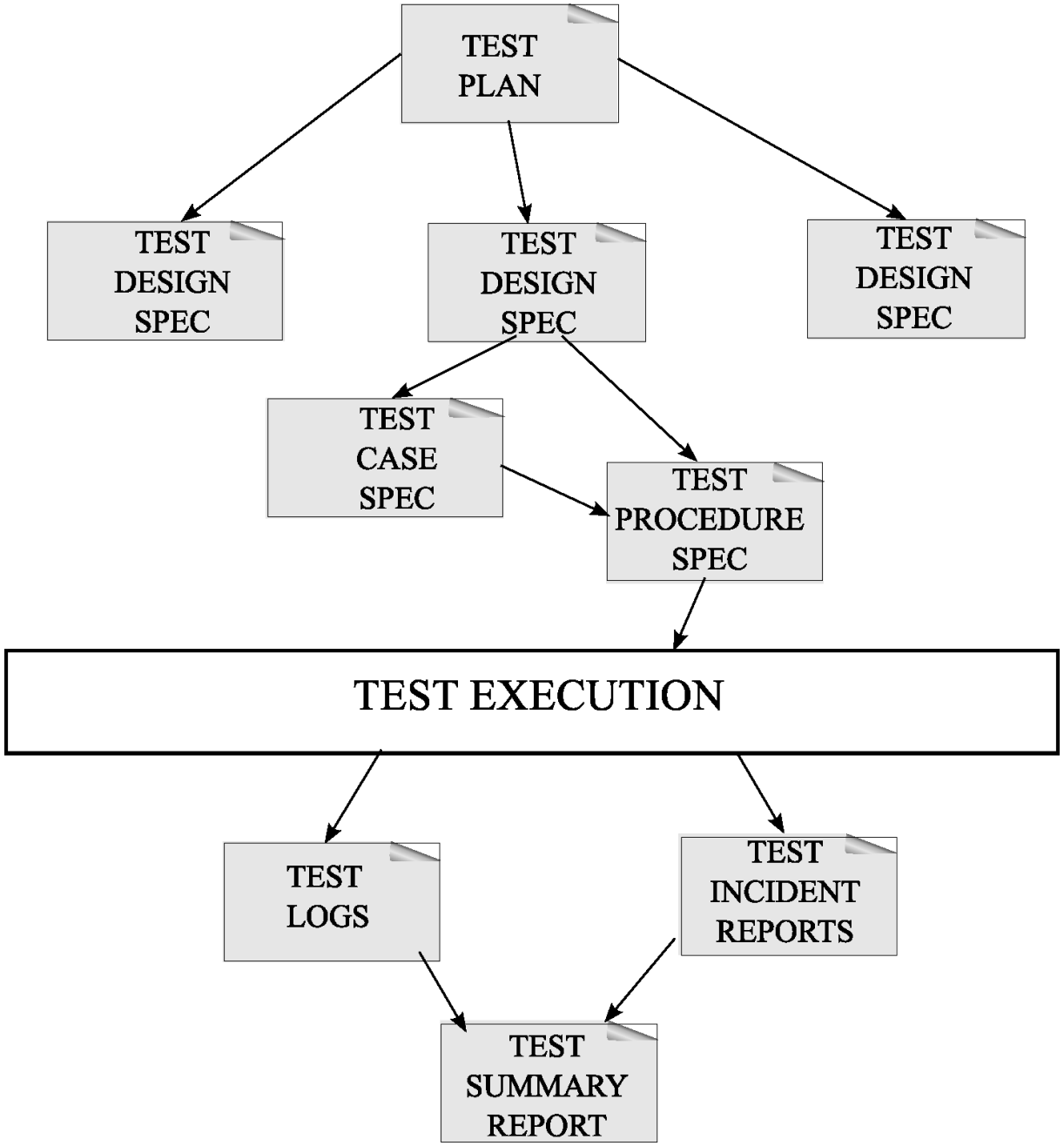


Figure 9.2: Key Documents in IEEE Standard 892-1998

9.2 Software Development Life Cycle

The Software Development Life Cycle is a structured plan for organizing the development of a software product. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process. The need for such planning arose with the growth in size and complexity of software projects. Unstructured approaches were found to result in cost overruns, late deliveries and uncertainties regarding quality. By adopting a plan for the development it was intended to create a repeatable and predictable software development process that would automatically improve productivity and quality. All models for development include software testing as part of the process but differ on the emphasis it is given. The next sections discuss some of the models and the role software testing has within them.

9.3 The Waterfall Model

This model visualizes the software development process as a linear sequence of phases beginning with a requirements analysis, followed by the system design, then coding, testing, and ending with system maintenance after the software has been deployed. Figure 9.3 shows the sequence of phases. All the planning is done at the beginning, and once created it is not to be changed. There is no overlap between any of the subsequent phases. Often anyone's first chance to "see" the program is at the very end once the testing is complete.

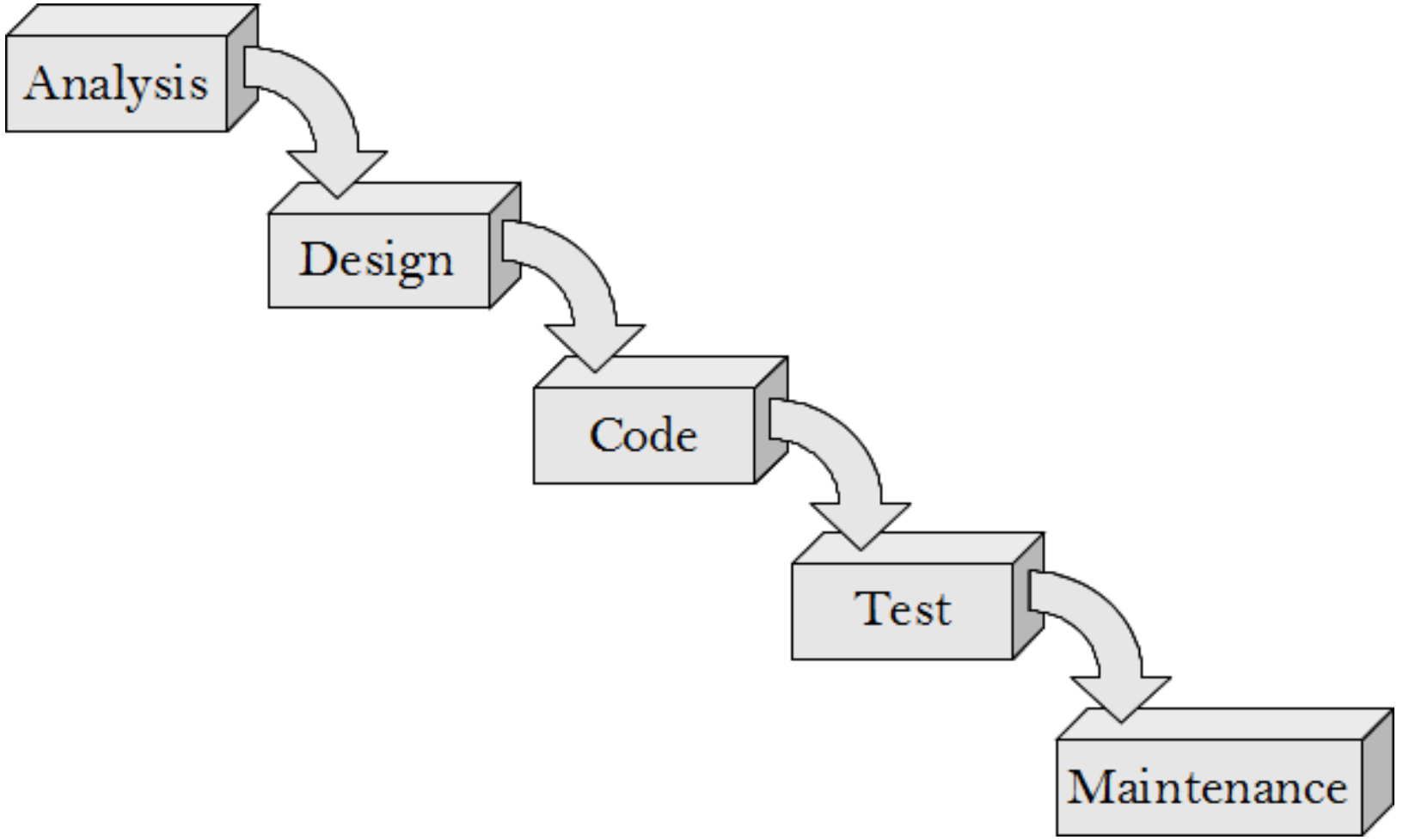


Figure 9.3: Waterfall Model

The idea behind the Waterfall model is that if time is spent early on making sure that requirements and design are absolutely correct then this will save much time and effort later. Thus, no time will be wasted creating a design from inaccurate requirements, or writing code based on a faulty design. Furthermore, there is an emphasis on documentation which keeps all knowledge in a central repository and can be referenced easily by new members joining the team. Criticisms it has received are that there are few visible signs of progress until the very end, and that it is not flexible to changes.

For example, the requirements may need to change during the project or a better approach to the design may become obvious during the coding phase. Additionally, it can be time-consuming to produce all the associated documentation. From a testing viewpoint, all the tests are carried out once the software is completed. This can cause a number of problems. Firstly, if there is any budgetary or time pressure on the project at this stage it could result in insufficient or incomplete testing being carried out. This

could be further exacerbated by much testing being done on the program as a whole rather than systematically progressing from Unit testing to System testing. Secondly, if testing does encounter many faults in a particular part of the program, while it could be more sensible to redesign it, this option would be excluded in favor of trying to fix the program followed by more testing. If some faults are very difficult to trace it could result in many iterations back and forth between fixing and testing. Lastly, once the customer receives the product, they may have a number of changes they want to be made to it. If there are many changes, this can mean a very long Maintenance phase, as alterations to the product will now be made without any proper process to control them.

In 2003, the IEEE software society carried out a survey of over 148 software projects with participants from India, Japan, the US and Europe. It found that only 36% used the Waterfall model as-is, while in the remaining 64% a variation on the Waterfall model was used (mixing it with features from other, "Iterative" practices).

9.4 The V-Model

This is an extension of the Waterfall model. Unlike, the Waterfall model this model emphasizes Verification & Validation by marking the relationships between each phase of the life cycle and testing activities. Once the code implementation is finished the testing begins. This starts with unit testing, and moves up one test level at a time until the acceptance testing phase is completed. Figure 9.4 shows a diagram of the progression of the V-model.

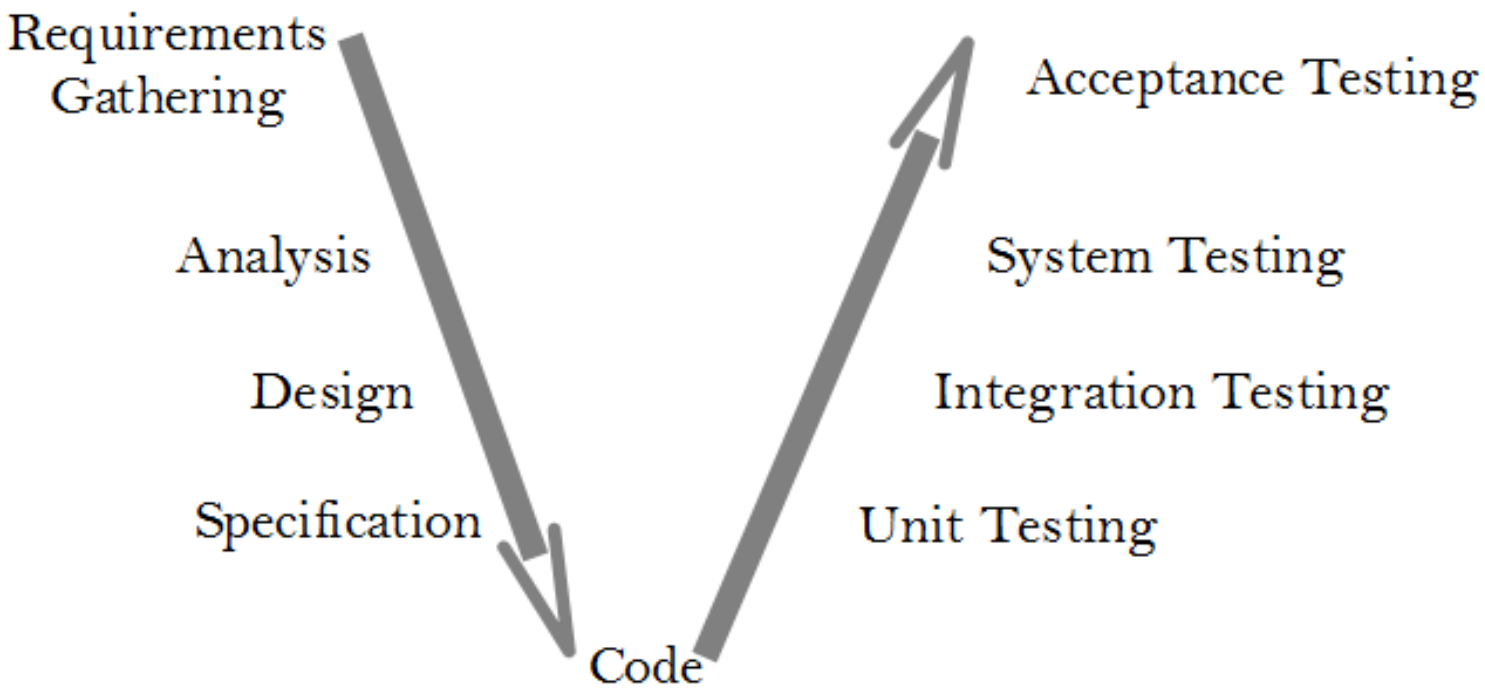


Figure 9.4: V-Model

Each document produced is associated with pairs of phases in the model. These are the (a) Detailed Design Specifications, (b) the System Design Specifications, (c) the System Requirements Specification, and (d) the User Requirements Specification.

As shown in Figure 9.5:

- Requirements Gathering produces the User Requirements Specification (URS), which is both the input to Analysis, and the basis for Acceptance Testing.
- Analysis produces the System Specification (SS) – also know as the Software Requirements Specification (SRS) – which is both the input for Software Design, and the basis for System Testing.
- Design produces the System Design Specification (SDS), which is both the input for the detailed Specification phase, and the basis for Integration Testing.
- The Specification activity produces the Detailed Design Specifications (DDS), which are both used to write the code, and also are the basis for Unit Testing.
- After code is produced, it then goes through Unit Testing, Integration Testing, System Testing, and finally Acceptance Testing.

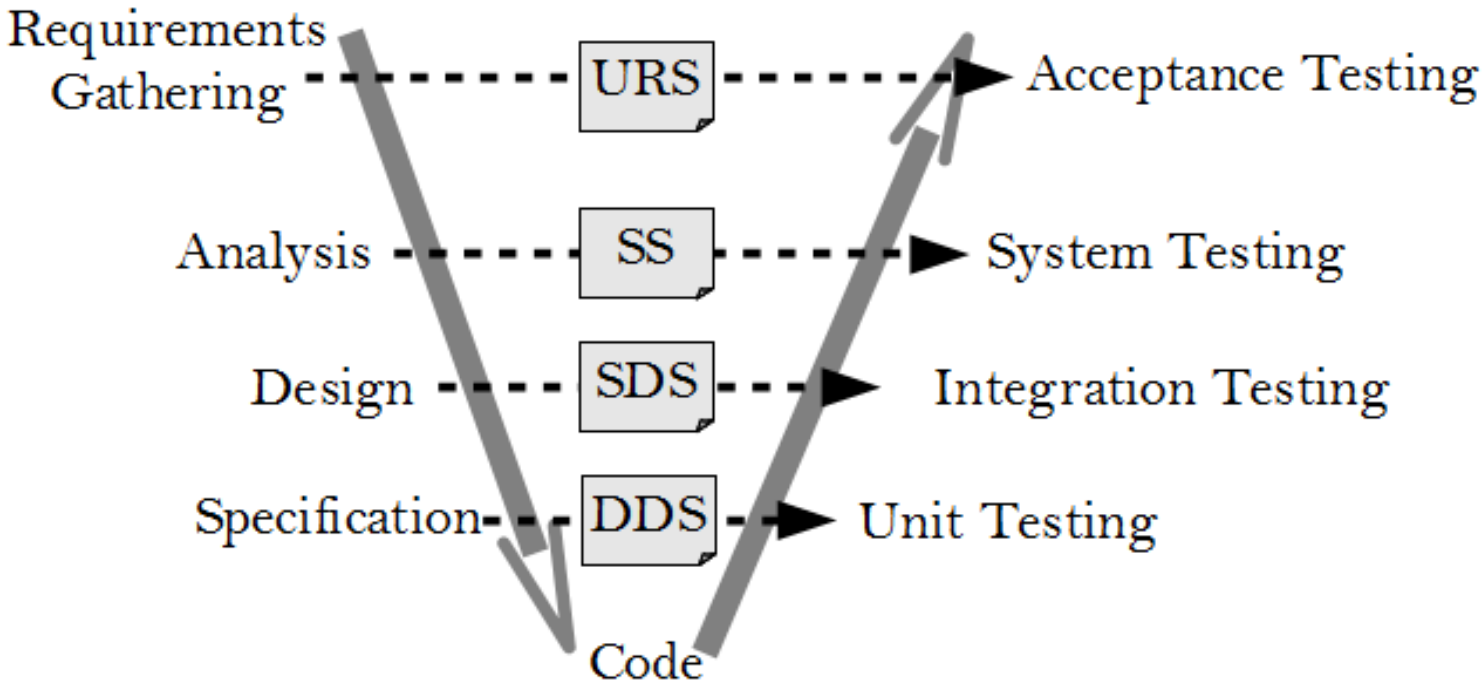


Figure 9.5: Documentation in the V-Model

The advantages of the V-model are that it is simple and easy to manage due to the rigidity of the model, and that it encourages Verification and Validation at all phases: each phase has specific deliverables and a review process. Unlike the Waterfall model it gives equal weight to testing rather than treating it as an afterthought. Its disadvantages

are that similarly to the Waterfall model there is no working software produced until late during the life cycle and it is unsuitable where the requirements are at a moderate to high risk of changing. It has been suggested too that it is a poor model for long, complex and object-oriented projects.

9.5 Incremental and Agile Development

In some software projects it is impossible to arrive at a stable, consistent set of system requirements at the beginning of the project. In these situations, the Waterfall and V Models of development are impractical, and an approach based on incremental or Agile models can provide much better results. Agile software development is associated with the Agile Manifesto[2].

The Manifesto outlines four key components:

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

Agile methods share with other incremental development methods an emphasis on building releasable software in short time periods. However, Agile development differs from the other development models in that its time periods are measured in weeks rather than months and work is performed in a highly collaborative manner.

One major difficulty that might appear for software testing in Agile development models is that, because the requirements can change quickly, the software tester's job can be much harder. They must continually keep up with changes to the software. If this were always the case, testing in Agile development could be almost chaotic. To make the tester's role effective in agile software development a number of rules need to be observed:

- When the developers "negotiate" the requirements for the upcoming iteration with the customers, the testers must be full participants in those conversations. This includes asking clarifying questions, noting where requirements are untestable and pointing out other traps that experienced testers will see.
- The testers immediately translate the requirements that are agreed upon in those conversations into test cases. Those test cases serve as the requirements documentation for the upcoming iteration. As soon as possible, the testers and developers collaborate in automating those test cases.
- When requirements change, testers are immediately involved because everyone knows that the test cases must be changed accordingly.

9.5.1 Incremental Development

The incremental model begins with a simple implementation of a part of the software system. With each increment the product evolves with enhancements being added every time until the final version is reached. A simple illustration of the process is shown in Figure 9.6. Testing is an important part of the incremental model and is carried out at the end of each iteration. This means that testing begins earlier in the development process and that there is more of it overall. Much of the testing is of the form of regression testing. However, the incremental nature of the process means that much re-use can be made of test cases and test data from earlier increments. The quality of the final product should, in theory, be better because of the increase in the frequency of test activities during the whole project.

A major advantage of the incremental model is that the product is written and tested in smaller pieces. This reduces risks associated with the process and also allows for changes to be included easily along the way. Additionally, by adopting an incremental model the customer or users have to be involved in the development from the beginning which means the system is more likely to meet their requirements and they themselves are more committed to the system because they have watched it grow. Thus, another advantage of the Incremental approach is an accelerated delivery of customer services: important new functionality has to be included with every iteration so that the customer

[2]Manifesto for Agile Software Development, Beck *el at*, 2001

can monitor and evaluate the progress of the product. Two primary disadvantages are that it can be difficult to manage because of the lack of documentation in comparison to other models and the continual change to the software can make it difficult to maintain as it grows in size.

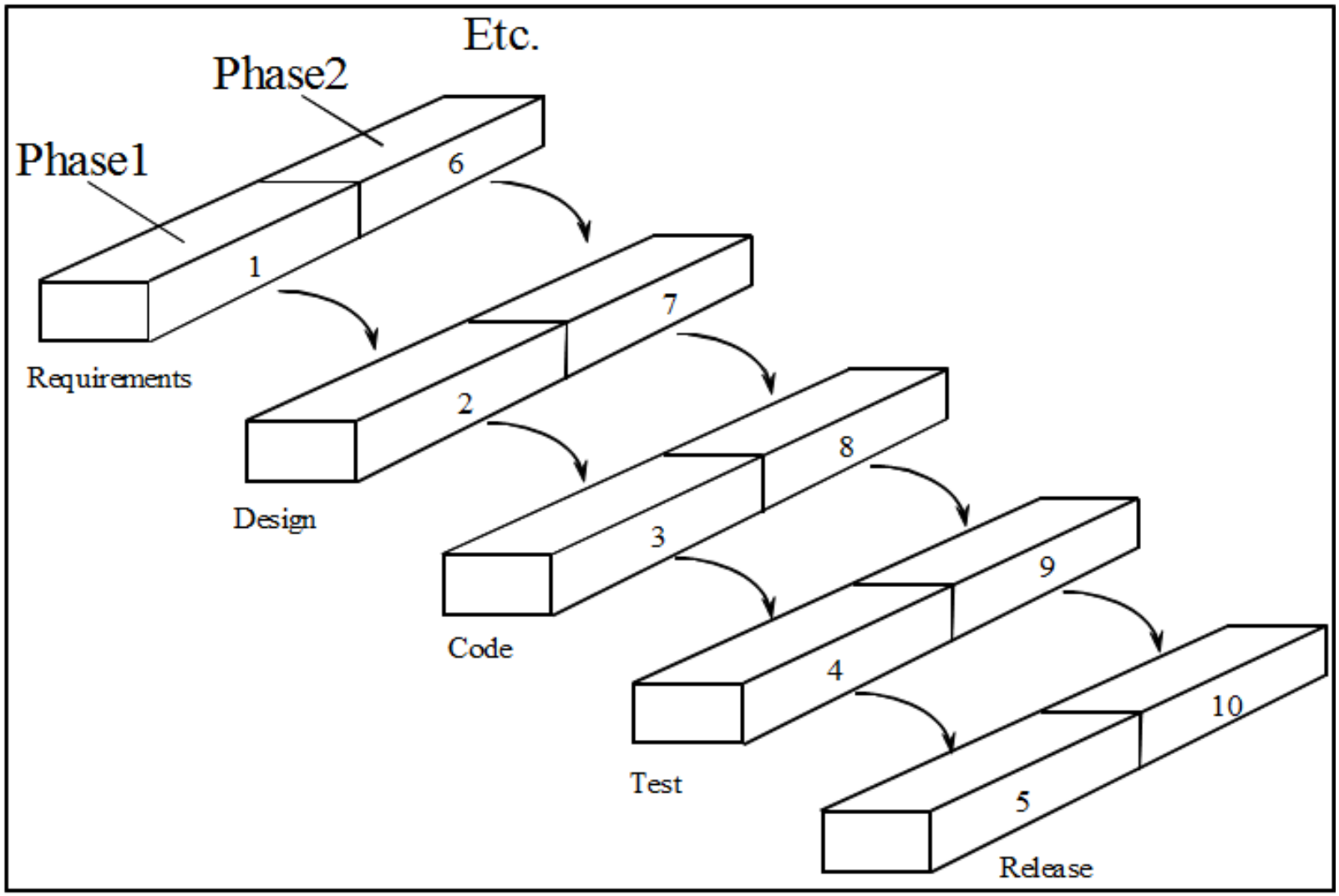


Figure 9.6: Incremental Development

9.5.2 Extreme Programming

Extreme Programming (XP) is a recent approach to software development. It is a subset of the philosophy of Agile software development. It emphasizes code reviews, continuous integration and automated testing, and very short iterations. It favours ongoing design refinement (or *refactoring*), in place of a large initial design phase, in order to keep the current implementation as simple as possible. It favours real-time communication, preferably face-to-face, over writing documents, and thus there is a lack of documentation compared to the traditional waterfall method. Working software is seen as the primary measure of progress. The methodology also emphasizes team work. Managers, customers, and developers are all part of a team dedicated to delivering quality software. Programmers are responsible for testing their own work; testers are focused on helping the customer select and write functional tests, and on running these tests on a regular basis.

XP has four key values:

- Communication: XP programmers communicate with their customers and fellow programmers
- Simplicity: they keep their design simple and clean
- Feedback: they get feedback by software testing from the start
- Courage: they deliver the system to customers as early as possible and implement changes as suggested, so they respond with courage to changing requirements and technology

A project begins by identifying a metaphor that describes the system. The metaphor acts as a conceptual framework, identifying key objects and providing insight into their interfaces. This intention is to have an idea that all people involved in the project will understand. The first iteration sets the initial skeleton of the project.

User Stories are written by the customers. These are the features of the application that the system needs to have. They are in the format of about three sentences of text written by the customer using non-technical language. These User stories will also drive the creation of the acceptance tests later on

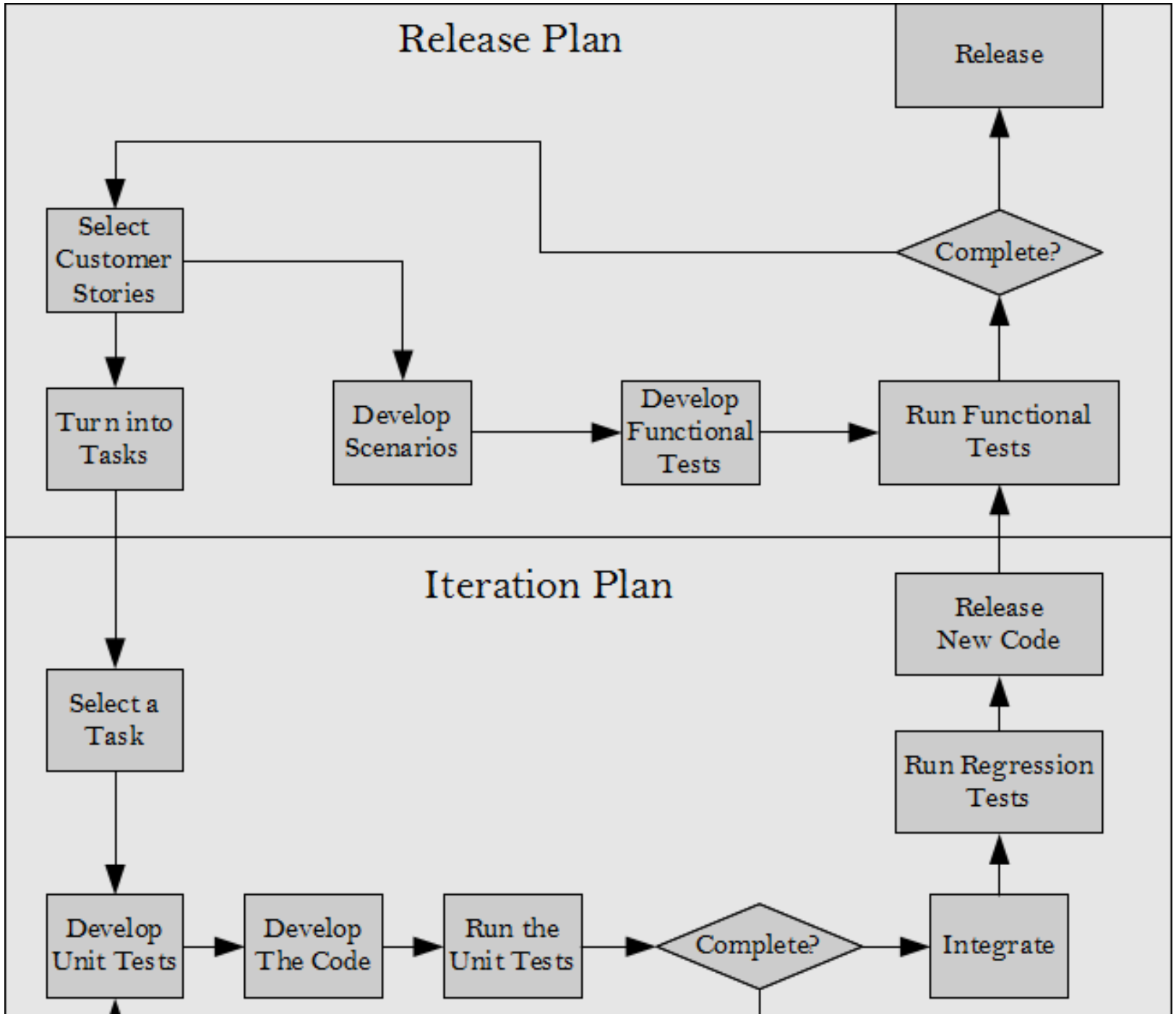


Figure 9.7 is a diagram of the XP process, highlighting the testing activities.

- A Release Plan is created from the User Stories. This plan sets out the overall project. The customer then decides what story or stories have the highest priority. Iteration plans are then created for each individual iteration, using development time estimates for each user story. The customer specifies scenarios to show that a user story has been correctly implemented. A set of functional (or acceptance) tests is developed based on these. The customers are responsible for verifying the correctness of the acceptance tests, and reviewing test scores to decide which failed tests are of highest priority. Acceptance tests are also used as regression tests prior to the release of a new version of the software.
- Each Iteration Plan is developed in detail just before the iteration begins and not in advance. Iterations are between 1 and 3 weeks in duration. User Stories are converted into implementation tasks, recorded on task cards. A programmer takes a task card, writes the unit test cases for the task, implements the code, and tests it. When the tests pass, the programmer then integrates the new code, runs regression tests, and releases the code for full functional testing. After this, there is a tested, working, software feature ready to demonstrate to the customer. Eventually, after all the iterations have been completed the product will be finished.

Testing is a key part of XP. Unit tests are created before the code. Doing this helps the developer to think deeply about what they are doing. The requirements are defined fully by the tests. Another benefit is that the code, influenced by the existing unit tests, is expected to be clearer to understand and easier to test. When a bug or error is found new tests are created to ensure it has been fixed.

The principles of eXtreme Programming are applied directly to the testing process in *Extreme Testing*.

9.5.3 SCRUM

SCRUM is a process for managing complex software projects and is a technique of Agile software development. It is similar to XP but there are a number of differences:

- Scrum teams work in iterations that are called sprints. These can last a little longer than XP iterations.
- Scrum teams do not allow changes to be introduced during the sprints. XP teams are more flexible with changes within an iteration as long as work has not started on that particular feature already.
- XP implements features in a priority order decided essentially by the customer, while in SCRUM there is more flexibility for additional stakeholders to influence the ordering.
- In XP unit testing and simple design practices are built in, while in SCRUM it is up to the team to organize themselves and adopt the practices they feel work best for themselves.

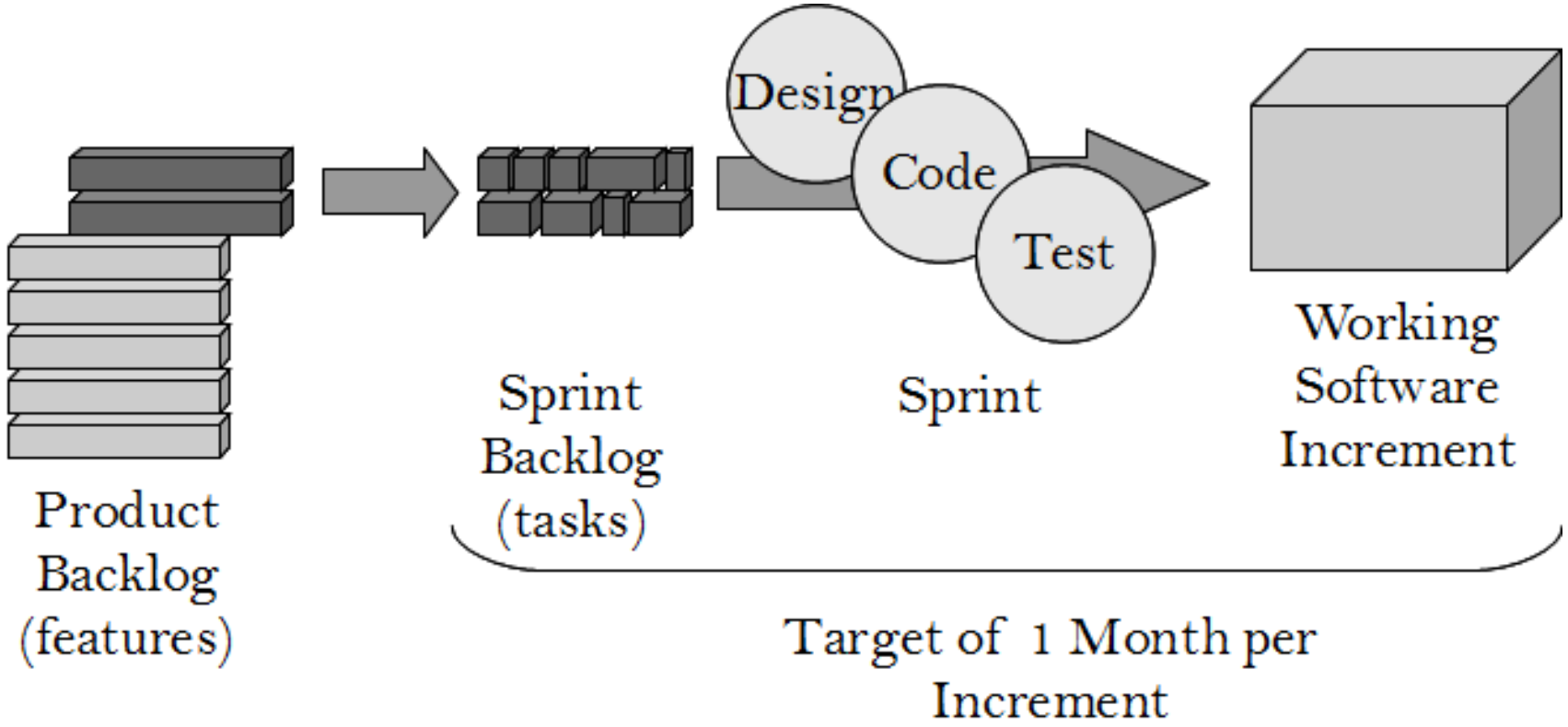


Figure 9.8: The SCRUM Process

As shown in Figure 9.8, SCRUM starts with the Product Backlog which is the prioritized list of all product requirements. Items most likely to generate value are at the top; the list is divided into proposed releases. The backlog items come from: Users, customers, sales, marketing, customer service, engineering, and anyone else that has an interest in the outcome of the project. The Product Backlog is never finalized, it emerges and evolves with the product.

Small, cross-functional SCRUM teams perform all development. SCRUM teams take on as much of the product backlog as they think they can turn into an increment of product functionality within a 30-day iteration. This is called a Sprint. The team maintains a list of tasks to perform during each Sprint that is called the Sprint Backlog. Multiple teams can take on product increments in parallel, all working from the same Product Backlog. A project will have multiple sprints. Before a sprint starts a meeting is conducted to decide what is going to be developed and delivered in that particular sprint. After the completion of the sprint, a meeting is held to collect feedback from the team. This feedback helps in planning and working on the next sprint. As the development team starts to work on the next sprint, the testing team carries out functional testing of the features developed in the last sprint. This approach gives good results as the testing team works with the developers from the start of the project.

9.5.4 Synch And Stabilize

This process model is associated with Microsoft. It has small parallel mini-teams with 3 to 8 developers each, or individual programmers, that work together as a large team. Each team has the freedom to evolve their design. However, since the teams have so much freedom the risk is that their work may become incompatible so it has to be ensured that they regularly synchronize what they are doing.

There are three phases: (1) Planning, (2) Development, and (3) Stabilization, as shown in Figure 9.9. In the planning phase the managers define the goals for a new product. An order of priority is then made for user activities that will be supported by product features. The planning phase output is a specification document, a schedule and team formation. Testers are also included in the teams as they work in parallel with the developers and the ratio of testers to developers is 1:1.

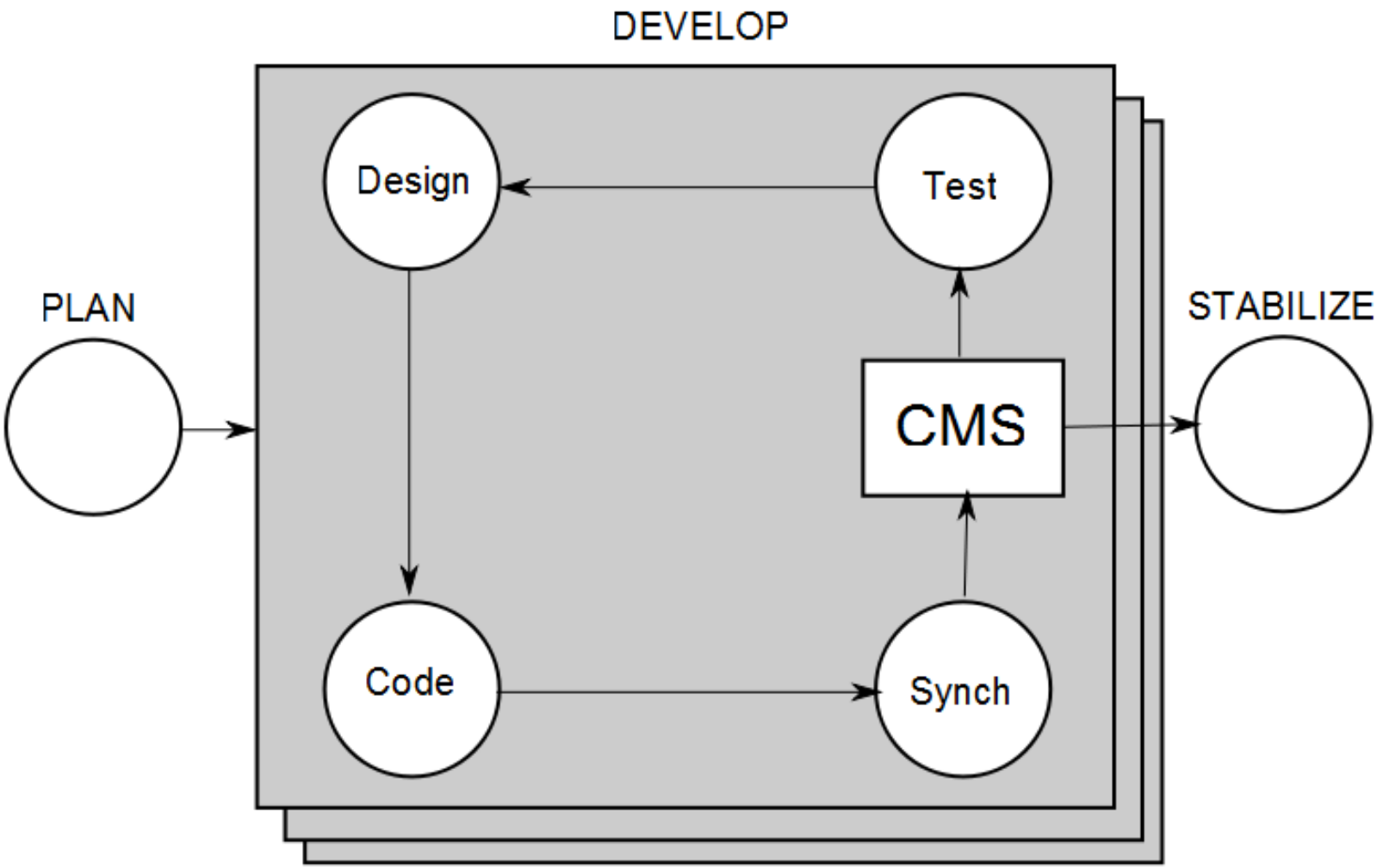


Figure 9.9: Synch and Stabilize

In the Development phase, feature development happens over 3-4 subprojects that last about 2-4 months each. Each subproject has design, code and testing activities. Feature teams synchronize work by building the product, i.e. putting together partially completed or finished pieces of the software, on a daily and weekly basis and finding and fixing errors. At the end of a subproject the product is stabilized.

The goal is for the process to produce zero defects in the final product. This is achieved by synchronising the activity of parallel development teams through continuously integrating the code, and periodically *stabilizing* the product. Code is checked into the Code Management System (CMS) by a specific deadline every day. Synchronization

is achieved using a daily (or nightly) build, with immediate testing, to ensure that small problems do not grow into large problems before they are found. Any code that *breaks* the build must be fixed immediately. During Stabilization, internal testing of the latest version of the complete product is performed. Once the product is finished, external testing is carried out by beta testing organizations, Independent Software Vendors (ISVs), and Original Equipment Manufacturers (OEMs). Finally, there is a release preparation which organizes everything before the final release and distribution of the product.

As a process model, Sync and Stabilize is good when it is desirable to ship preliminary versions, add features in the current or subsequent releases, and allow easier integration of pieces of products that may still be years away. It has a strong emphasis on testing as there is continuous testing of the product from development until release.

9.5.5 Process-Related Quality Standards and Models

There are a number of other models and quality-related standards that the tester should be aware of, though they do not have a direct relationship to testing. Two key ones are the Capability Maturity Model (CMM) and the ISO 9000 series of standards.

CMM

The *Capability Maturity Model* was developed initially by the Software Engineering Institute at Carnegie Mellon University in 1991 as a model based on best practices for software development. It describes an evolutionary method for improving an organisation from one that is *ad hoc* and immature, to one that is disciplined and mature. Subsequently, an updated model, CMMI (CMM Integrated), has been developed covering a broader selection of activities.

The CMM ranks software development organizations in a hierarchy of five levels, each with a progressively greater capability of producing quality software (see Figure 9.10):

- Level 1 – Initial (also referred to as Chaotic or *Ad Hoc*). Processes are typically undocumented and dynamic. They are driven in an uncontrolled, reactive manner by users or events.
- Level 2 – Repeatable. Processes are repeatable, possibly with consistent results. Process discipline is unlikely to be rigorous, but where it exists it may help to ensure that existing processes are maintained during times of stress.
- Level 3 – Defined. Defined and documented standard processes are established and subject to some degree of improvement over time. These processes are used to establish consistency of process performance across the organization.
- Level 4 – Managed. Using process metrics, management effectively controls the process. This includes identifying ways to adjust and adapt the process to particular projects without measurable losses of quality or deviations from specifications.
- Level 5 – Optimizing. The focus is on continually improving process performance through both incremental and innovative technological changes/improvements.

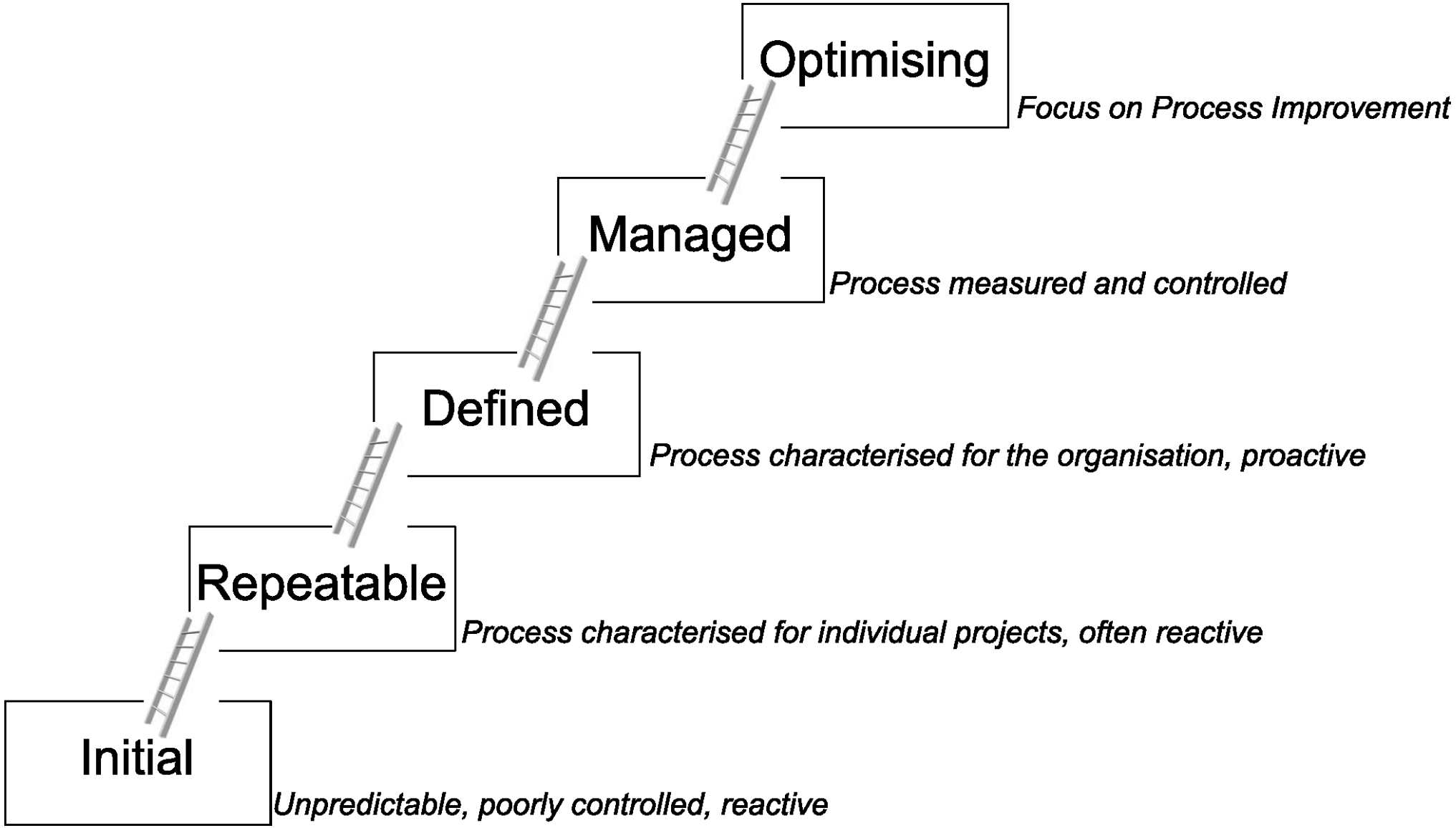


Figure 9.10: Levels of The Capability Maturity Model

CMMI defines a number of roles and Software Engineering process areas. Testing is mainly associated with the Software Quality Assurance (SQA) and Software Quality Control (SQC) roles in the CMMI model. The main test activities are in the following Software Engineering process areas:

- CMMI Technical Solution – Unit Testing
- CMMI Product Integration – Integration Testing
- CMMI Verification – System Testing

The test results provide a measure of the process quality, used as a significant input to the Process Assessment and Improvement activity.

ISO 9000 Series

The term "ISO 9000 Series" is used to refer to a family of quality standards, relating to quality management systems. The Series is designed to help organizations ensure they meet the needs of their customers and stakeholders. The first version of ISO 9001 was published in 1987, based on a series of standards proposed by the British Standards Institute (BSI), published in 1979.

The key standards in the ISO 9000 Series of relevance to software testing are:

- ISO 9000:2000 Quality Management Systems – Fundamentals and Vocabulary
- ISO 9001:2000 Quality Management Systems – Requirements
- ISO 9004:2000 Quality Management Systems – Guidelines for Performance Improvement
- ISO 19011 Guidelines on Quality and/or Environmental Management Systems Auditing
- ISO 10012-2:1997 Guidelines for Control of Measurement Processes

The series is not aimed at addressing technical issues associated with software testing, but the focus on process management does have a direct impact on the environment in which testers work, and of the expectation placed on their processes.

The ISO 9000 Series Process Model is based on eight quality management principles:

- Customer focus
- Leadership
- Involvement of people
- Process approach
- System approach to management
- Continual improvement
- Factual approach to decision making
- Mutually beneficial supplier relationships

Some of the key records in ISO9001:2000 which may be of direct relevance to testing are as follows:

4.2.3 Control of documents

4.2.4 Control of records

7.3.5 Design and development verification

7.3.6 Design and development validation

8.2.4 Monitoring and measurement of product

ISO9001 certification can be an important item in some software development environments. For example, the UK Board of Trade *TickIT* guidelines are an interpretation of ISO9000 produced to suit the processes of the IT industry, and especially software development. ISO/IEC 9003:2004 provides guidelines for the application of ISO 9001:2000 to computer software.

9.6 Repair-Based Testing

Following the discovery of faults, at any stage in the software development process, the faults are prioritised and repaired. Following this repair, additional tests are often developed to verify that the fault has been correctly repaired. This is especially important if existing tests have not discovered the fault (e.g. a fault reported by a customer).

There are generally three different levels of such tests that can be developed: specific, generic, and abstracted.

9.6.1 Specific Repair Test

The specific repair test verifies that the software now operates correctly for the specific data inputs that caused the failure. This will often consist of a White-Box test that ensures that executing the new/modified code provides the correct output.

9.6.2 Generic Repair Test

In order ensure that the repair works for other data values, and not just those that caused the failure, the conditions that caused the failure should be analysed, and Black-Box Tests developed that exercise these conditions. The generic repair test verifies that the repair works for different data values from those that caused the failure, where those values exercise the same failure condition.

9.6.3 Abstracted Repair Test

In order to broaden the value of finding a fault, it is useful to abstract the fault as far as possible. Then additional Black-Box Tests can be developed that attempt to find other places in the code where the same mistake may have been made by the programmer, leading to similar faults. If the fault has a characteristic signature (often caused by cut-and-paste operations), then this can be searched for in the code in order to determine whether it is worth the effort of developing these abstrated repair tests.

9.6.4 Example

Consider a program that at some point has to search a list of data for a particular entry. The data is kept in a linked-list, and due to a mistake by the programmer the code never finds penultimate entries in the list (i.e. entries which are second-from-last). This fault eventually causes a failure, which is reported by a customer. The fault is then located by replicating the customer's conditions and debugging the code. Subsequently the fault is repaired by rewriting a line of code.

To verify the repair, a White-Box Test Unit Test is developed that exercises the new line of code and verifies that the method it is in produces the correct result for the data that previously caused a failure. A Black-Box Unit Test and/or System Test is then developed that ensures that, for different sets of data, the second last entry can be found in this list.

Finally, additional Unit Tests and/or System Tests are developed that check that, for any other lists or collections of data in the program, the second-to-last entry can be located correctly.

9.6.5 Repair-Based Test Suites

These additional tests need to be run immediately after the repair to verify it. Then they can be added to the standard Unit Test and Regression Test Suites for the code. This ensures that the fault is not repeated – and that a developer does not by mistake revert to a previous version of the code.

Prior to release of a software product, the risk of repaired faults having been re-introduced can be reduced by re-running all the Repair-Based Tests in a separate Test Suite.

Chapter 10

Advanced Testing Issues

In this chapter some more advanced testing issues are identified, and solutions briefly discussed.

10.1 Philosophy of Testing

Software testing can used to achieve a number of different goals. Typically developers will use it to ensure that they have a low number of faults; testers may use testing to ensure that a system works correctly; quality auditors may use it to measure the mean time to failure; and customers may use it to decide when to pay.

A few quotes from well-known authors are shown here to demonstrate this - comparing their definitions with the IEEE definition:

IEEE Testing is "The activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component."

Glass Testing is "The process of executing computer software in order to determine whether the results it produces are correct."

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

Myers "If our goal is to show the absence of errors, we will find very few of them... If our goal is to show the presence of errors, we will discover a large number of them."

Dijkstra "Program testing can be used to show the presence of bugs, but never to show their absence!"

Hennell "The aim is not to discover errors, but to provide convincing evidence that there are none, or to show that particular classes of faults are not present."

Hetzel "Testing is the measure of software quality."

10.2 Test Technique Selection

One of the questions that cannot be answered in the general case is "*what is the best test technique to use?*". Even if a more satisfactory theory of testing is developed, it will not be able to answer this question. The best test technique is the one that finds the most faults in any given piece of software; and that will be dependent on the faults in the software.

This shows one of the benefits of collecting fault data and analysing the types of faults. By doing this, a fault profile data can be developed, and then tests developed targeted at the most likely faults.

10.3 Inserting Data Faults

As a complement to Mutation Testing, faults can be inserted into good data, and the behaviour of the code observed. As in inserting faults into the code, pre-existing tests are needed to exercise the software, and a test passes when the insertion of a fault results in a change in the output. The pre-existing tests are also needed to verify that the change in behaviour is correct.

10.4 Design For Testability (DFT)

For software to be tested thoroughly sometime requires it to be written with testing in mind. Some examples follow.

Adding Functionality to Enable Testing A good example of this is doing state-based testing of classes. Testing could be enabled by adding extra methods to enable the current state to be queried, or to provide read access to private attributes for test purposes.

Another example would be logging internal activity, such as messages (method calls) between objects. The log can be analysed after the fact, allowing the internal operation of the code to be checked. An example would be using a UML sequence diagram to verify that the correct methods are being called with the correct parameters in an object-oriented program.

Adding Built-in Test Built-in-test (using assertions for example) can be added to the code in order to improve the effectiveness of the testing. Normally built-in test features can be disabled at compile time or run-time in order to avoid any performance penalty associated with them. Sometimes a small number of built-in-tests will be enabled in the released software to provide increased confidence in its correct operation - this may be seen in the form of reported "internal consistency" errors.

Separating the GUI from the functionality This allows the GUI components to be tested separately from the software system. This is particularly valuable for complex interfaces, where the test tool can make sure that the correct software component is being called for each user action.

It also allows the functionality to be tested separately from the interface. This has two advantages. The first is that typically not all the software features can be reached from a GUI interface - the interface itself handling data conversion errors. The second is that double-faults can sometimes hide failures, where faults in the interface and the functional code cancel each other out.

In addition to making testing easier, this provides a number of other benefits. It allows the front-end to an application to be changed independently from the application. This enables the same functionality to be used with a web-based interface, a windows-based interface, and a command-line interface, without the need to change and test the functionality each time.

10.5 Testing with Floating Point Numbers

In many modern processors, and programming languages, real numbers are represented using the IEEE 754 floating point representation. The layout for 32-bit (Java float) and 64-bit (Java Double) numbers are shown in Figure 10.1. This consists of a sign bit, 8/11 bits for the exponent, and 23/52 bits for the coefficient. The coefficient is interpreted as a binary number between 0 and 1, and the overall value is calculated as follows:

$Value = (-1)^{sign} * (1.coefficient) * 2^{exponent}$

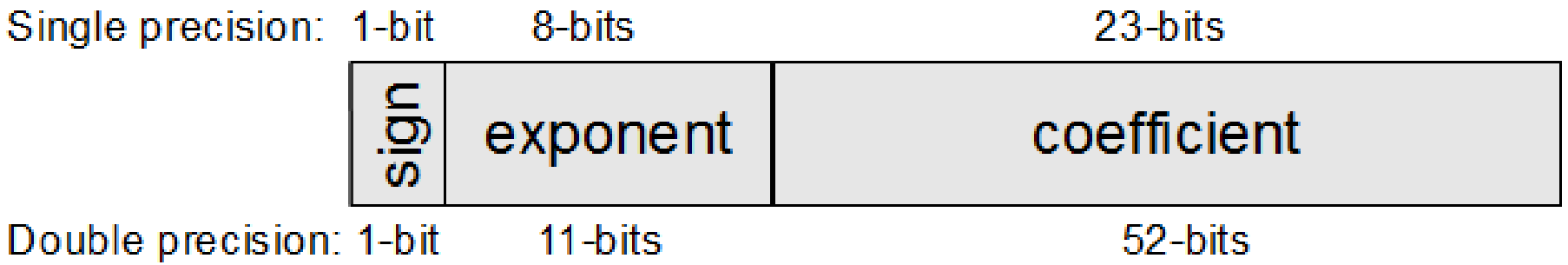


Figure 10.1: IEEE 754 Floating Point

Floating point numbers are much harder to handle than integers for testing purposes. One of the main reasons is that floating point arithmetic is seldom exact. For example:

- What are the ranges for a partition with `float` values from 0.3 to 2.7? What is the next floating point number less than 0.3? What is the next floating point number greater than 2.7?
- How do you test that a function returns the `double` value 148.45? How accurate must the answer be?

Just embedding these constants into the test code will not always work properly: many decimal numbers cannot be represented precisely in binary floating point (in the same way that $\frac{1}{3}$ cannot be accurately represented in decimal, 0.1 cannot be accurately represented in binary).

Also, floating point errors tend to accumulate within a program, and so floating point results will seldom be "exact" even if they can be represented. For example: the Java constant `0.3f` produces the value `0.30000001192092896`[1].

The following list shows a few problems that the software tester may experience with floating point, and suggests remedies for these.

Comparing to a Constant For the two reasons discussed above, comparing a returned floating point return value with a constant floating point value is likely to fail. Instead, it needs to be compared to a range of values, representing the minimum and maximum allowable values. For example, if the answer must be correct to 6 decimal places, then the allowable limits would be the expected value ±0.0000005. If the software is not specified with an accuracy margin, then strictly speaking it cannot be tested!

Finding Boundary Values Given a value between 0.0 and 1.0, what are the boundary values of the partition below 0.0 and the partition above 1.0? For the partition below, is it -0.1 or -0.0001 or -0.0000000001? And for the partition above is it 1.01 or 1.0001 or 1.0000000001?

If the programming language or libraries support it, then you can program features to find floating point numbers just below and just above a value. Otherwise, you can extract the three fields from the IEEE representation and calculate the value from those.

For example, in Java, the methods Java.lang.Math.nextUp() and .nextAfter() can be used to do this.

Handling Cumulative Errors Because not every number can be exactly represented, and values can get lost when subtracting or adding a very small number from a very large one, errors tend to accumulate in software that uses floating point. In order to handle this, the maximum allowable error needs to be specified for such software, and then as for comparing to a constant, this provides the upper and lower bounds on a correct output.

[1] System.out.printf("%30.28f",0.3f);

Input and Output It is easy to make a mistake when reading or writing floating point numbers. For example, the visible format of the number may change depending on its value (0.000001 vs 1.0000E-6). It is therefore very important to understand where these changes take place, and to test software input and output with very large and very small floating point numbers to make sure they are handled correctly.

10.6 Unit Testing of GUI Components

GUI components provide an interface for System Testing – but the GUI components can also be tested on their own. By using the same test tool as for System Test, which provides inputs and outputs via the GUI, but special test code in place of the application functionality, the GUI component can be tested for correctness. Such tests include:

- Verifying that input data is correctly converted from text to binary form.
- Verifying that the correct methods are invoked as *callbacks* by the GUI.
- Verifying that data from the correct input fields is passed to each input parameter.
- Verifying that the results are displayed in the correct output field(s).

An example set of test tools for implementing this is JUnit (as the test harness) and Abbot[2] (for the GUI interfacing).

10.7 Testing with Complex Data Structures

In the examples shown in this book, as in most books on testing, the data types have all been very simple (for example, integer or boolean). Handling more complex data types, which occur frequently in programs to be tested, is more complex. Some examples include:

- arrays
- graphs (for example, linked-lists, or class relationships)
- files
- databases

In reality, each element in a complex data structure is a separate variable. But it is not feasible to generate all the possible variations of elements and their values. In addition, complex data structures introduce new variables, or "meta-data", such as the size of the structure.

So, when testing code with complex data structures, the principles of testing must be applied to more than just individual data values. Some of the issues to be addressed are:

- Applying the principles to both *meta-values* and values. Examples of meta-values include array length, file name, file location, file size, graph size and depth.
- Applying the principles to both values, and the relationships between values. For example, an array sorted in increasing order, in decreasing order, and randomly sorted.

Files may have a defined structure, with an associated syntax. This may be representable as a graph or state machine, suitable for graph coverage or state-based testing. The keywords may be in text or binary form, suitable for testing using boundary values and combinations.

[2] See http://abbot.sourceforge.net

10.8 Automated Random Testing

Using two techniques already discussed, selecting input data at random and using assertions, fully automated random testing can be achieved. There are two problems with random testing: selecting useful input data, and working out the expected outputs automatically (note the problem: the program to calculate the expected output has exactly the same specification as the program you are testing – it is called a "Test Oracle" – so how do you test the Test Oracle?).

The first question (selecting useful input data) is more difficult to resolve. For example, consider a method to determine whether two lines are the same length. With randomly generated lines, it is highly unlikely that they will be exactly the same length. So the problem is to not just generate random data, but rather to generate random inputs. In this case it is the relationship between the lines which needs to be randomised, and so a random generator would need to generate random relationships, as well as random values for the actual line lengths. One way to support this is to provide random generators with every class as part of Design For Testability.

The second question (providing a Test Oracle) is resolved more easily. One approach is to use a higher-level, more mathematical language to specify functionality. This, in many cases, can allow software to generate expected outputs more accurately, though more slowly, than the low-level software written to execute the function. An alternative approach is to specify relationships, derived from the specification, that the output must have to the inputs. This allows an actual output to be checked for correctness without actually calculating the expected output value! This is often implemented as *assertions* in the code.

There are few working examples at the moment of fully automated runtime checking of pre-conditions, post-conditions, and invariants. One good example, though it is still a work in progress, is JML2[3]. JML provides support for *pre-conditions*, *post-conditions*, and *invariants*. These can be used for static design verification, but also for dynamic run-time testing using *runtime assertion checking* or RAC. The following code is a simple example of a method annotated in JML for testing purposes.

```
public class Demo {
   //@ ensures \result==x+1;
   public int inc(int x)
   {
      if (x>1)
         return x+1;
      else
         return x;
   }
}
```

A random number generator can be used in DemoTest.java to generate random input data, and the JML RAC raises an exception if the post-condition is not met. Note: `\result` refers to the value returned by the method. The notation `\old(x)` can be used to refer to the value of x on entry to the method.

10.9 Automated Statement and Branch Testing

There are many tools available to measure statement and branch coverage automatically (some representative Java examples are Emma, Cobertura, and Codecoverage). However, finding the correct input data to force a particular statement or branch to be executed is time consuming, manual work.

Recent developments in *directed* automated test data generation are addressing this problem. Using either heuristics, or analysis of the conditions, the code and branches followed can be monitored, and progressive changes made to the input data to try and achieve 100% coverage. One good example is PEX[4].

[3]Developed by the Department of Computer Science at ETH Zurich
[4]Developed by Microsoft

10.10 Overlapping and Discontinuous Partitions

Sometimes the specification will state different processing for the same range of values, and in this case the partitions and boundary values have to be carefully examined.

Consider the example `boolean isZero(int x)`. There is clearly one EP with the single value 0, and a second partition with all the values except 0.

The best way to handle overlapping ranges for input parameters is to separate out contiguous ranges of values.

So in this example, x should be treated as having the equivalence partitions:

x.EP 1 Integer.MIN_VALUE..-1

x.EP 2 0

x.EP 3 1..Integer.MAX_VALUE

And the boundary values:

x.BV 1 Integer.MIN_VALUE

x.BV 2 -1

x.BV 3 0

x.BV 4 1

x.BV 5 Integer.MAX_VALUE

The reasoning behind this is that the values -1 and +1 are special: they are immediate predecessors and successors to the value 0 which is treated differently. They are values which are likely to have faults in the software!

Consider a different example, with overlapping ranges for the return value in the method `int tax(int amount, boolean fixed)`. This method returns a value as follows:

- -1 indicating an error, if $amount < 0$
- 100, if fixed is true
- $amount/100$ (representing a 1% tax), otherwise.

The overlapping ranges for the return value are shown in Figure 10.2. The value 100 is both in a partition of its own, and also in the middle of a partition from 0..Integer.MAX_VALUE/100.

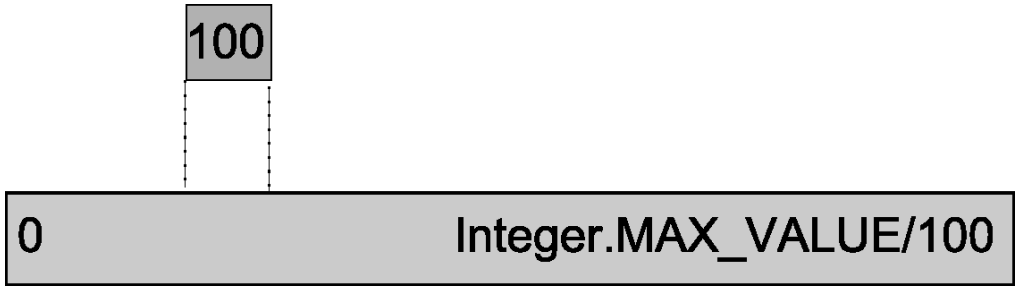


Figure 10.2: Overlapping Partitions

The best way to handle overlapping ranges for output parameters, is to ignore the overlap, and treat each range of values separately.
So in this example, x should be treated as having the equivalence partitions:

x.EP 1 -1

x.EP 2 0..Integer.MAX_VALUE/100

x.EP 3 100

Note that values less than -1 and greater than $Integer.MAX\_VALUE/100$ are not possible according to the specification, and thus are not valid partitions.
And the boundary values:

x.BV 1 -1 (from EP1)

x.BV 2 0 (from EP2)

x.BV 2 Integer.MAX_VALUE/100(from EP2)

x.BV 3 100 (from EP3)

The reasoning behind this is that the values 99 and 101 are not special: they are not values which are likely to have faults in the software. However, 100 is a special value, as it is produced by special processing – in this case by two different types of processing.
The rules for treating overlapping partitions are as follows:

- In general, break inputs into separate partitions.
- In general, treat outputs in as few partitions as possible, even if some are not contiguous.

10.11 Handling Relative Values

Another problem related to input (and output) parameters, is that not all partitions are defined by absolute values. Some partitions may be defined as relative to another input, or by a relationship between inputs.

Consider the method `int largest(int x, int y)`. There are three different cases to consider for x – see Figure 10.3:

1. $x < y$
2. $x == y$
3. $x > y$

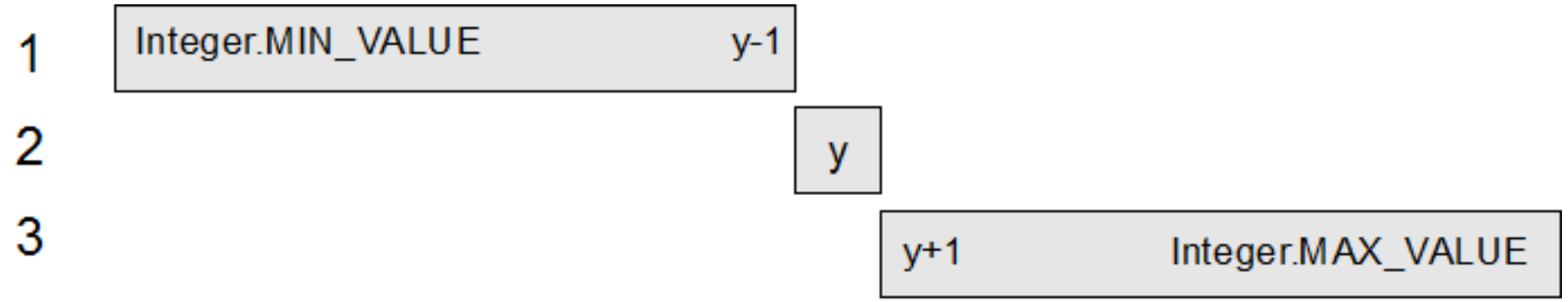


Figure 10.3: Relative Partitions

And three cases for y:

1. $y < x$
2. $y == x$
3. $y > x$

The best way to handle this is to regard x as a constant when considering y, and y as a constant when considering x. This gives the following partitions for x:

- Integer.MIN_VALUE..y-1
- y
- y+1..Integer.MAX_VALUE

and for y:

- Integer.MIN_VALUE..x-1
- x
- x+1..Integer.MAX_VALUE

The boundary values for x are:

- Integer.MIN_VALUE
- y-1
- y
- y+1
- Integer.MAX_VALUE

and for y:

- Integer.MIN_VALUE
- x-1
- x
- x+1
- Integer.MAX_VALUE

10.11.1 Classic Triangle Problem

A classic testing problem[5] is to classify a triangle as Equilateral, Isosceles, Scalene or Invalid based on the lengths of its sides. Let us consider the equivalence partitions for this problem, using the method

```
String classify(int x, int y, int z)
```

[5] Myers, The Art of Software Testing

where x, y, and z are the lengths of the three sides.

Clearly negative numbers are invalid, and we will consider 0 as invalid as well. We will also consider the data invalid if the points are co-linear (on the same straight line), or if the sum of the shortest two sides is less than the longest side (leaving a gap) – see Figure 10.4.

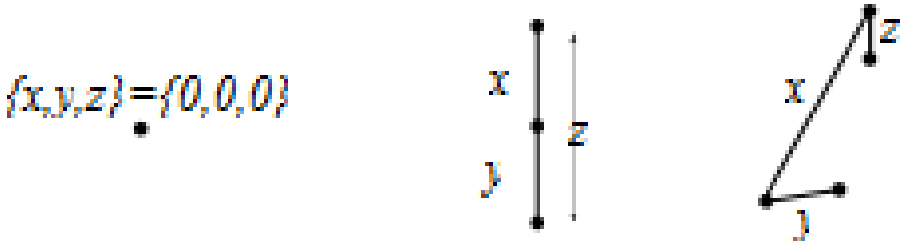


Figure 10.4: Invalid Triangles

Using the principles of overlapping partitions we have considered, this gives the following partitions:

x.EP 1 Integer.MIN_VALUE..0

x.EP 2 1..y-1

x.EP 3 y

x.EP 4 y+1..y+z-1

x.EP 5 1..z-1

x.EP 6 z

x.EP 7 z+1..y+z-1

x.EP 8 y+z..Integer.MAX_VALUE

And similar partitions for y and z.

10.12 Pairwise Testing

If there are a large number of combinations to be tested for, then the number of tests can be significantly reduced using *pairwise-testing*. This reduction technique can be applied to both Black-Box and White-Box tests, and at both the Unit Test and System Test level. Pairwise Testing reduces both the time for test design and coding (as there are fewer tests to design), and the time for test execution (as there are fewer tests to run).

The technique involves identifying every pair of combinations - and then combining these into as few tests as possible.

Consider, for example, exhaustive testing of a Truth Table with four independent, Boolean parameters. There are 16 rules, and thus 16 Test Cases to consider, and 16 tests to design and implement, as shown in Table 10.1.

Table 10.1: Truth Table for 4 Booleans

Test Cases															
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F
T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F
T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F
T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F

Using *Pairwise Testing* this is reduced to 6 Test Cases, as shown in Table 10.2. For each pair of parameters, every possible combination of {T,T},{T,F},{F,T},{F,F} occurs at least once.

Table 10.2: Pairwise Test Cases for 4 Booleans

Pairwise Test Cases					
1	2	3	4	5	6
T	T	F	F	T	F
T	F	T	F	T	F
T	T	F	T	F	F
T	F	T	T	F	F

This leads to a significant reduction in the number of tests. There are a small number of published research results showing that the additional benefits of 3-way, 4-way, N-way, and all combinations are not cost effective.

10.13 Testing Boundary Crossings (BX)

The technique of Boundary Value Analysis, as described in almost all the text books and published papers, requires generating test data to minimise the number of tests.

Consider the following example: method `negProduct(int x,int y)` which returns true if the product of two integers is negative.

The standard BVA input Test Cases are:

Parameter x Integer.MIN_VALUE, -1, 0, Integer.MAX_VALUE

Parameter y Integer.MIN_VALUE, -1, 0, Integer.MAX_VALUE

Giving the following tests:

Table 10.3: BX Tests for negProduct()

ID	**Test Cases Covered**	**Inputs**		**Expected Output**
		x	y	return value
1	1,5	Integer.MIN_VALUE	Integer.MIN_VALUE	false
2	2,6	-1	-1	false
3	3,7	0	0	false
4	4,8	Integer.MAX_VALUE	Integer.MAX_VALUE	false

Note that many combinations of boundary values have not been tested. One approach to resolve this is to test the "corners of the box". This means making sure that all the minimum-maximum combinations of independent parameters are tested. For two parameters this means the four corners of each box defined by the minimum and maximum values for each parameter partition. And for three parameters, this would be the eight corners of a cube, etc.

Using this approach, for the box defined by the corners:

{Integer.MIN_VALUE,Integer.MIN_VALUE}, and

{-1,-1}

This gives the following Test Cases:

Box 1.Corner 1 x=Integer.MIN_VALUE, y=Integer.MIN_VALUE

Box 1.Corner 2 x=Integer.MIN_VALUE, y=-1

Box 1.Corner 3 x=-1 , y=Integer.MIN_VALUE

Box 1.Corner 4 x=-1 , y=-1

There would be corresponding Test Cases for the other three boxes, defined by the corners:

{-1,0},{Integer.MIN_VALUEInteger.MAX_VALUE}

{0,0},{Integer.MAX_VALUE,Integer.MAX_VALUE}

{0,-1},{Integer.MAX_VALUE,Integer.MIN_VALUE}

Another approach is to test every boundary crossing separately. So for the example above, the test cases are the boundaries crossings B1, B2 and B3. And for each boundary, there are two tests: one at each side of the boundary. This gives the following Test Cases:

Boundary 1 $y<0$, x:-1$\rightarrow$0 (with $y<0$, x transitions from -1 to 0)

Boundary 2 y≥0, x:-1→0

Boundary 3 x<-1, y: -1→0

Boundary 4 x≥0, y:-1→0

This is shown by the arrows marked A,B,C,D in Figure 10.5.

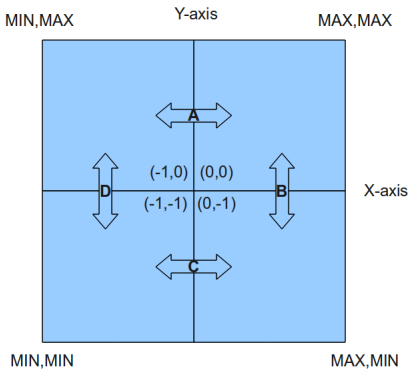


Figure 10.5: Boundary Crossings

As for other test techniques, the effectiveness can only be measured by how well the techniques finds faults. If a particular team of developers produces a high incidence of boundary-related faults, this technique could be valuable in finding them.

10.14 Varying Input Parameters

Normal scientific practice is to modify one parameter at a time when designing experiments to observe the behaviour of a system. This provides an alternative approach to designing tests: rather than trying to minimise the number of tests by modifying as many parameters as possible between tests, only change one parameter at a time.

This could be applied to Black-Box test techniques (such as EP, BVA, and TT) by only varying one input parameter value between tests. It could be applied to White-Box test techniques by only varying one covered component between tests (e.g. one different line of code, or one different branch).

There is no published research on the effectiveness of this approach (i.e. does it find enough extra faults to make the additional effort of designing tests and the additional time to execute the tests worthwhile).

10.15 Extended Combinational Testing

As normally defined, Combinational Testing (using *Truth-Tables* or *Cause-Effect Graphing*) uses "don't-care" conditions to reduce the number of rules. These "don't-care" conditions represent parts of the specification where the same output *effect* should be produced for different input *causes*.

However, this is a likely area for a programmer to make mistakes. In a way that is similar to boundary values (where the incorrect handling of these will cause faults), the incorrect identification and handling of "don't-care" conditions is also likely to be a source of faults. And unless the specification is provided in the form of a Truth-Table, it is highly likely that a programmer will make mistakes, especially for complex conditions.

There are two possible approaches to improving Combinational testing: not using "don't-care" conditions, and testing each "don't-care" condition individually.

10.15.1 No "Don't-Care"s

Fully expand out all the rules in the table, giving 2^N rules for N *causes*. Then each rule is a Test Case, and must be tested in a separate test.

10.15.2 Test "Don't-Care"s Individually

The alternate approach is to add extra tests for potential faults (where "don't-care" conditions may not have been handled correctly by the code).

Develop the Truth-Table, including "don't-care" conditions. Each rule is a Test Case, and must be tested in a separate test.

Subsequently, add extra Test Cases for the "don't-care" conditions. One value for each "don't-care" condition has already been tested (either true or false), so this adds one extra test for each "*" in the table (the opposite value from that already used).

While not providing full coverage of all the possible *combinations* of "don't-care" conditions, this does provide significant enhancement of them, ensuring that each "don't-care" has been tested at least once for both true and false values.

10.16 Including Testing In The Build Procedure

Standard build scripts provide two automated features: they recompile only the required source code based on dependencies, and they provide the correct parameters to the compiler and/or linker to build the software. Examples would be *make* (widely used under UNIX, and also used in Microsoft Visual Studio and Eclipse), and Ant (a Java-based make utility).

A *build* succeeds if the compiler and the linker produce no errors. Note that the compiler tests primarily for syntax errors (though modern compilers can also test for some semantic errors, such as the use of uninitialised variables).

A simple example is shown below, using GNU make syntax. The first line states that the file `Demo.class` depends on the file `Program.java`. This means that if the .java file is more recent than the .class file, then the command on the following line should be executed to bring the .class file up to date (the source file has been edited since it was last compiled).

```
Demo.class: Demo.java
	javac Demo.java
```

To integrate testing into the build procedure, the program can be made dependent on not only it passing the compiler, but also passing unit tests. If the unit test program `DemoTest` produces the output file `Demo.success` only if the tests pass, then the Demo.class file can be made dependent on both the input source code file and this test success file. The unit tests can then be added to the sequence of commands to be run to bring the class file up to date, as shown below:

```
Demo.class: Demo.java Demo.success
	javac Demo.java
	java DemoTest Demo
```

Now the build not only requires the source code to "pass" the compiler, but also the compiled code to pass the unit tests. If the test fails, then the .class file "Demo.class" should be automatically deleted to prevent it being used.

10.17 Testing Concurrent and Parallel Software

Concurrency is widely used in large software systems, using multiple threads of execution for reasons including performance, responsiveness, modularity, reliability, robustness, supporting multiple clients, scalability, and security. Examples include: graphical user interfaces, databases, web servers, etc. Parallel programming is used almost exclusively for performance reasons – and typically runs on arrays of processors. Examples include: engineering and scientific simulations, weather forecasting, financial analysis, etc. In both cases, *nondeterminacy* in the order of execution of interacting tasks makes both implementation and testing difficult (there being typically a very large number of possible permutations and combinations in the order of execution). The inclusion of timing as an additional dimension to testing causes the number of test cases to increase exponentially ("state explosion").

The key additional feature of these systems, from a testing viewpoint, is the need to test the co-ordination between different executing entities. This testing tries to expose any timing or sequencing "windows" that may exist. A timing/sequencing window, in this context, is a section of critical code which is *not* properly protected. Thus, under the right conditions (sometimes referred to as *Race Conditions*), two different executing entities can access the same data at the same time (leading to "data corruption"), or can enter a state where both tasks are waiting for the other to complete (a "deadlock" if permanent, a "livelock" if only caused by a continuous load). For parallel processing, with common access to shared memery (such as in multi-core threading), this access can literally be at the same time. For non-parallel cases, this access can be when a change of context takes place in a critical region (such as between the read and write when incrementing a counter, which can result in an increment getting lost).

There are generally two dynamic approaches (in addition to the Black-Box and White-Box testing techniques already described). One is to control the sequence of task execution in order to force every possible combination and permutation of task ordering – or, if this is infeasible, to exercise classes of ordering (selected using the principles of Equivalence Partitioning and Boundary Value Analysis). The other is to try and cause as many random ordering conditions as possible – usually by stress testing the software.

Another problem with testing these systems is repeatability. Due to the nondeterministic nature of software activity, it may be difficult to repeat a test exactly in order to "debug" a fault (find and repair it). One approach to this, which is not needed for most other forms of software, is to use a record-and-playback technique, allowing the exact sequence of actions to be replicated (as well as providing the exact input data that caused the failure).

10.17.1 Unit Testing

This is generally difficult to achieve, but can lead to a higher confidence in the results. The tests need to try and interrupt one activity with another, and ensure that both activities complete correctly. This may be done by careful control of the ordering of execution.

10.17.2 System Testing

Typically the input load will be varied to try and expose timing and sequencing windows. The load may be provided either by running multiple tests in parallel, or by adding an extra load while running a test. In addition to Functional Testing, Performance Testing is important in ensuring that the performance goals have been met.

10.17.3 Static Analysis

The difficulty of confirming correct operation through testing makes design verification and static analysis of the code particularily valuable techniques for concurrent and parallel software.

10.17.4 Tools

Typical examples of tools used in Concurrency Testing are ConTest and jCute.

ConTest

The main use of ConTest is to expose and eliminate concurrency-related bugs in parallel and distributed Java programs.

ConTest systematically and transparently schedules the execution of program threads such that program scenarios that are likely to contain race conditions, deadlocks, and other intermittent defects (collectively called synchronisation problems) are forced to appear with high frequency.

ConTest is applied by instrumenting[6] the bytecode of the application in places that are likely to be involved in concurrency faults. Then the tester or developer can run existing tests to verify concurrency.

The ConTest run-time engine is called by this instrumentation. The engine adds heuristically-controlled conditional sleep and yield instructions, which help reveal concurrency bugs. Each existing test can be run many times: ConTest causes different behaviours every time, and new faults may surface. ConTest also collects run-time information for coverage statistics, and to aid in location detection for debugging.

jCUTE

The jCUTE tool is used to test multi-threaded Java programs, combining Testing and Model-Checking (dynamic and static verification). For concurrent programs, jCUTE can catch race-conditions and deadlocks, and generate optimal test inputs and schedules for path coverage, branch coverage, or error coverage.

jCute can also be used to test non-concurrent programs: it can generate JUnit tests with the optimal number of test inputs for path coverage, branch coverage, and error coverage.

jCUTE produces detailed branch coverage statistics. For aid in debugging, jCUTE can record and replay the execution of a program.

10.18 Testing Embedded Software

Embedded Systems are often used for device control (Control Software) with very specific timing requirements (Real-Time Software). These systems are often used for life-critical applications (such as medical devices, transport, the space program etc.), or mission-critical applications (such as satellites, banking, tax collection, security, etc.). This leads to very high quality requirements. These may be met by rigorous analysis and design practices and process, and are generally verified by extensive testing (note: formal verification can also play significant role in such systems).

Testing embedded software is significantly more difficult than testing a *host* application. There is often no access to file systems, local storage space is often limited (and may have read-only access), CPU speeds tend to be lower (for example, in the MHz range rather than the GHz range), and memory space tends to be limited.

Some of the key features of Embedded Systems that are likely to result in faults, especially for less experienced programmers are:

- Hardware interfaces, which often have complex, state-based behaviour
- Interrupts, which open the opportunity for synchronisation errors in critical regions
- Limited memory space, often requiring complex design optimisations
- No memory protection, opening the opportunity for memory corruption, especially in many systems where addresses above the end of available RAM will often wrap back to the start address
- The use of languages, such as C, with little 'protection' from semantic errors (in particular array size protection, and invalid pointer protection). Even if the language does have protection features, these are frequently disabled for space or performance reasons
- Direct access to CPU features, often involving bit-set operations
- Special hardware, which may be not be as well characterised or specified as off-the-shelf hardware (and may even contain faults which require software workarounds)

[6] Instrumentation modifies the compiled bytecode by adding special markers at every branch and method invokation. Later, during execution of the instrumented source code, these markers are tracked and counted to provide data for reports

- Direct access to hardware, especially communication devices, often requries the same program to deal with both *little-endian* and *big-endian* data
- The programmer often has to consider word-size and data alignment issues, especially to optimise the use of memory space
- Special memory architectures are often used, for example requiring separate access to I/O space, RAM, and executable code
- Embedded code often suffers from *resource leaks*, where resources are not properly returned after use (for example, dynamically allocated memory). In any software, this leads eventually to hung programs (blocked waiting for resources), or software failure (unable to continue execution). In embedded software, this can lead to lockup, or to resets, both of which impact on the integrity and reliability of the system.

There are a number of approaches for testing embedded systems:

- Design For Testability – design the code so that the components can be unit tested individually. This generally requires structuring the code so that calls to the embedded operating system are not distributed throughout the code. Generally *processing* and *decision* components can then be unit tested, but *interface* components cannot. An often used technique is to provide *hardware-dependent* and *hardware-independent* software layers to improve portability. This layering can improve testability, allowing the hardware-independent layers to be tested on a host system.
- Use of assertions during test – this allow for testing of components or functionality that cannot be unit tested. These assertions are usually disabled for production software.
- Using test hardware or software that allows the software to be tested over its operational interface – this might include robotic arms to activate buttons or switches, cameras to read displays, etc.
- Emulating the embedded operating system on a host computer – this generally requires providing virtual device implementations for the various devices used by the software.
- Simulating the embedded operating system on a host computer – this generally involves simulating the execution of each instruction, providing a very accurate match to the real execution environment. This can be relatively slow, and more critically, the various hardware devices used by the software are often difficult to simulate.
- Timing problems are notoriously difficult to find – anything that changes the relative timing of the software (such as running on an emulator or in a simulator for testing/debugging purposes) can change the behaviour of a fault. As a consequence, timing-related tests often need to be run on the real hardware.
- Memory corruption is also a problem on embedded systems – typically there is no memory management on small devices. Both errant pointers (in C) and stack overflow can easily occur. Unit testing the components on a host help to find these. Also, during system test, checksums can be used to detect if areas of data have been changed when they should not have been. One approach is to enable all the compiler checking and safety features during testing only. Another is to add data integrity checking, which again only runs during testing.
- Often a non-production communications channel (often RS-232) can be used during testing. This allows input data to be generated on a host, the tests to run on the embedded system, and the results to be communicated back to the test host. The tests are managed from the host, but run on the device.
- Verifying the use of dyanmic resources, by stress-testing software that makes use of such resources.
- Fault classification can be an effective tool, as embedded systems require a broad range of skills to be applied (thus exposing any weaknesses in the development teams). Analysis of the fault patterns of individual developers, or of teams, can often lead to both improved effectiveness of testing, and also to an improved level of design and code quality.

Much embedded software is also concurrent and/or parallel - see Section 10.17 for a discussion on related testing issues.

10.19 Testing Network Protocol Processing

There are two key forms of network protocols: text-based (such as HTML), and binary (such as IP). Binary protocols are typically tightly constrained, with the ordering and size of each protocol element fixed. Text protocols typically allow more flexibility, and have to handle more “unexpected” cases, such as duplication of protocol entries and

contradictory protocol entries. Protocols invariably have state-based behaviour, but this is sometimes not stated in the form of a state-machine (making both implementation and testing more difficult).

10.19.1 Text-Based Protocols

Text-based protocols are typically less constrained that binary protocols, and allow more freedom in how the protocol elements are included. For example, data fields often have a length specified separately from the actual data; numeric data is presented in readable form, requiring conversion to binary in the software; and various protocol elements may not be required to be in a particular order.

Text-based protocols are notoriously difficult to implement – and thus require rigorous testing. Some of the key problems that code often fails to handle properly are:

- Length values being different from the length of the actual data. This can cause software to hang, or memory corruption if when a small buffer is written with a larger amount of data.
- Buffer overruns – where the data length is larger than the software can handle (especially where the protocol has no specified limit), leading to memory corruption (many security problems have been caused by stack overruns where too much data is written to a local vriable).
- Contradictory protocol elements – a simple example would be two different length values given for a data field.
- Missing protocol elements – protocol processing software often uses unspecified default values, leading to incorrect processing.

The parsers for these protocols are often hand-crafted, as often the protocols are not in the form of context-free grammars. Error testing of these protocols often concentrates on making sure that the software handles all the different types of invalid input data correctly – at least without crashing or exposing security flaws, and ideally by producing the correct error result.

10.19.2 Binary Protocols

Binary protocols are in general easier to implement, and easier to test. The different fields are usually fixed size (or may have a number of different fixed-size options), and the order of the fields is usually also fixed.

System testing can be done using the principles of State Testing with field values selected using Equivalence Partitioning to provide a basic test of correct operation. Then, boundary values and combinational testing can be used to try and find faults in the protocol processing.

As for most other forms of testing, code and branch coverage can be measured during these Black-Box tests, and the tests the augmented with data values selected to force the required level of coverage. For protocl processing, it would not be unreasonable to aim for 100% coverage.

10.19.3 Protocol Stacks and DFT

Protocols are often implemented in the form of a stack, with each layer in the protocol calling layers above it and below it in the stack. This can make Unit Testing very difficult - requiring the use of many test stubs. By using Design for Testability (DFT), and separating the protocol processing from the inter-layer communication software, Unit Testing of the protocol processing can be made significantly easier.

10.20 Research Directions

As in most areas of Software Engineering, research into software testing is a wide and varied field. This section gives a brief overview of some selected research areas. To develop a broader understanding of the current state-of-the-art, the reader should review papers from recent conferences in Software Testing.

The following conferences are a good a starting point:

CAST Conference of the Association for Software Testing

EUROSTAR European Software Testing Annual Conference

ICSE International Conference on Software Engineering

ICSQ International Conference on Software Quality

ICST International Conference on Software Testing, Verification and Validation

ISSTA International Symposium on Software Testing and Analysis

QUEST Quality Engineered Software & Testing Conference

STAREAST and STARWEST Software Testing Analysis & Review Conference

STPCON Software Test Professionals Conference

In addition, many journals and books provide a source for reading on more established research results. A number of indicative research fields, selected by the authors, are outlined below.

Search-Based Software Testing Random Testing is often used to try and achieve White-Box coverage criteria, such as code coverage. It is, in general, an intractable problem to determine the input values to be used to cause specific lines of code to be executed. Research is directed at search-based techniques to achieve particular coverage criteria. Some examples include: random search, local search, evolutionary algorithms, ant colony optimization and particle swarm optimization.

Mutation Analysis The idea of Mutation Testing has mainly been applied at the source code level. Recently, the idea has been applied to test different artifacts, including research into the notation used. Examples include: automating mutation testing, discarding equivalent mutants, mutation testing in multi-threaded code.

Regression Test Reduction Running Regression Tests is often the most time consuming testing activity. When software is changed, or extended, checking that the existing functionality still works correctly is at least as important as checking that the new functionality works correctly. Techniques to reduce this time are often heuristic – even if one particular line of code has changed, it may result in failures elsewhere in the program. Research is directed at finding the optimal set of tests to re-run, often based on interpreting the structure of the code and identifying tests that execute that structure.

Model-Based Testing Recent years have seen increasing interest in the use of models for testing software, for example UML. Research includes formal verification of model transformations into code, adding debug support for model-based testing, automatic test data generation based on the models.

Automated Software Verification Formal specifications provide the basis for automating testing, either at runtime, or statically. These usually take the form of assertions stating pre-conditions, post-conditions, and invariants that must hold for the software to match its specification. Recent research has started to provide working tools that do both static and dynamic evaluation of these.

New Technologies New software technologies and architectures require the application of test principles in new ways. Examples include research into effective testing for SOA (Service Oriented Architecture), Virtualisation, Dynamic Software Systems, GUI (Graphical User Interface), Cloud Computing, etc.

Software Process and Tools Testing is a key part of the software process, not only to measure software quality, but also to measure process quality. Research is active into the relationship between testing and the other process activities, the development of test-driven development processes, and the effectiveness of testing as a quality measure.

Appendix A

Terminology

Acceptance Test This is the testing of a software system to ensure that the software meets the user requirements (that is, it solves the user problem).

Actual Output This is the actual data value which the SUT has produced during execution.

Black-Box Testing Black box testing (also referred to as Specification-Based or Function Testing) is based on deriving test cases from the program specification, and uses no knowledge of the inner workings of the program code.

Boundary Value This is the value at one end of a partition. For non-continuous partitions, the boundary values must be carefully selected according to the specification.

Branch This is represented by the execution of an edge in a CFG.

Condition This is a single predicate, or logical expression, contained in a decision.

Control Flow Graph This is a directed graph representing the flow of control through a program.

Decision This represents the reason for the execution of an edge in a CFG, where there are multiple outputs from a node. A decision is composed of one or more conditions.

d-u pair This is an execution path from the definition to the use of a variable or attribute.

Dynamic Testing A Dynamic Test is one which does require the execution of the software.

Expected Output This is the output data value which the SUT is expected to produce for particular input values according to the specification.

Integration Test This is the testing of multiple units or sub-systems of a software system. The purpose is to ensure that these components operate correctly together.

Partition This is a range of data values for which the specification states that the processing is identical. The range need not be continuous.

Path This is a start-to-finish path of execution through a piece of software.

Software Testing *"An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component."* [IEEE Standard 829-2008]

Specification This is a statement of the requirements that software must meet.

State Diagram This is a diagram specifying the required behaviour of software in terms of the states it can be in, the events which can be raised, the state transitions that result from these, and the actions to be taken on each transition.

Statement This may represent a line of source code, or a node in a CFG (an uninterruptible sequence of statements is equivalent to a single statement from a coverage viewpoint).

Static Analysis A Static Test is one which does not require the execution of the software.

Sub-System Test This is the testing of a *sub-system* of software, usually considered to be a block of software that can be executed separately from other sub-systems.

Subsume A test technique subsumes another when it produces a superset of the tests.

SUT Software or System Under Test: the software, or system, being tested.

System Test This is the testing of a complete software system. Test data is passed through the system interface (for example, a GUI or a socket). The purpose is to ensure that the system meets the system specification.

Test Code or Procedure For an automated test, this is the code that implements the test. For a manual test, this is the actions that should be taken by a tester to execute the test.

Test Case This specifies a particular aspect of the software under test to be exercises, based on the criteria of the test technique.

Test Data This is actual data values to be used to provide inputs to the SUT, or to be used to compare with actual outputs from the SUT.

Test Oracle Software that provides the correct output for the software under test based on its specification.

Test Suite A collection of tests that can be run together. Note that test suites can in turn be collected into larger test suites.

Truth Table This is a table specifying the required operation of software in terms of input causes and output effects expressed in predicate logic.

Unit Test This is the testing of a *unit* of software, usually considered to be a function or a method. Test data is passed through the programming interface of the unit.

White-Box Testing White-Box testing is based on deriving test cases from the internal details of the program code. The test data, however, will require use of the specification.

Appendix B

Exercises

The following are selected exercises to help the reader gain an understanding of applying the principles of software testing, and to use examples of testing tools.

B.1 JUnit and Eclipse

This example works with the Helios version of Eclipse and the standard JUnit plugin. You will need to modify these commands if you are using a different version of Eclipse, a different unit test plugin, or a different IDE.

1. From the Eclipse File Menu choose New->Java Project ensuring to click on "Create separate folders for sources and class files" under Project layout.
2. Next, in the src folder create a new package called sa
3. Now, create a new class in this package called seatsAvailable and copy the code in Figure B.1 into this template file.
4. Next, within the same package sa, create a new "JUnit Test Case" clicking on the button "New JUnit 4 test". The "class under test:" will be seatsAvailable. Copy the code below for the test into the black template.
5. You can now run the test with sample data. Use the Eclipse menu option "Run as junit test" to run the tests. A "green bar" will indicate a successful outcome and a "red bar" an unsuccessful outcome.

```
public class Seats {

   public static Boolean seatsAvailable(int freeSeats, int seatsRequired)
   {
      boolean rv=false;

      if ( (freeSeats>=0) && (seatsRequired>=1) &&
      (seatsRequired<=freeSeats) )
      rv=true;
      return rv;

   }

}
```

Figure B.1: Seats.java

```
import static org.junit.Assert.*;
import org.junit.Test;

public class SeatsTest {

   @Test public void testseatsAvailableYes() {
      boolean b = Seats.seatsAvailable(50,25);
      assertTrue(b);
   }

   @Test public void testseatsAvailableNo() {
      boolean b = Seats.seatsAvailable(-50,25);
      assertTrue(b);
   }
}
```

Figure B.2: SeatsTest.java

B.2 Unit Test - Exercise 1

For a transport system, a small (1 m^3 volume) container can hold a load with a maximum weight of 1 tonne, a medium (10 m^3) container can hold a maximum of 5 tonnes, and a large (20 m^3) container can hold a maximum of 10 tonnes.

Use the Black-Box techniques of (a) Equivalence Partitions, and (b) Boundary Value Analysis. For each, derive the boundary values, test cases, and test data for the method checkContainer(), defined below, which checks that a container is suitable for a given load. Write down any assumptions that you make. Clearly separate normal and error tests.

```
public static boolean checkContainer(
   containerSize size,
   int weight,
   int volume)

Specification:
   INPUTS:
      containerSize
         - type: enum Sizes {Small, Medium, Large}
         - the size of the selected container
      weight
         - type: int
         - the weight of the load to be carried in Kg
           (1000Kg=1 tonne)
      volume
         - type: int
         - the volume of the load to be carried
           (in cubic metres)

   OUTPUTS:
      Return false if invalid inputs, or the container
         cannot hold the load
      Return true, if the container can hold the load
```

B.3 Unit Test - Exercise 2

Draw a Control Flow Graph (CFG) for the method checkContainer() shown below (as specified as in Exercise 1) and derive test cases and test data to provide (a) Statement Coverage, (b) Branch Coverage, and (c) End-to-end Path Coverage.

```
 1  public static boolean checkContainer(
 2     containerSize size,
 3     int weight,
 4     int volume)
 5  {
 6     boolean ok  = true;
 7     if ((weight<0)||(volume<0))
 8        ok = false;
 9     else if ((size==Small)&&((weight>1000)||(volume>1)))
10        ok = false;
11     else if ((size==Medium)&&((weight>5000)||(volume>10)))
12        ok = false;
13     else if ((size==Large)&&((weight>10000)||(volume>20)))
14        ok = false;
15     return ok;
16  }
```

B.4 Unit Test - Exercise 3

Complete the Truth Table for class SimpleContainer as specified in Example 1, and then develop Combinational tests from the table. Rules 1-3 have been already completed - you must complete rules 4-8. Clearly specify the methods to be called, the parameters passed for the test data, and the expected outputs, for each test.

```
class SimpleContainer {
   public enum Sizes {Small, Large};
   // Class-wide truth table for valid data inputs.
   //                                  RULES
   //                          ------------------------
   //   CAUSES                  1  2  3  4  5  6  7  8
   //       weight <= 1000      T  *  T
   //       weight > 10000      F  F  F
   //       volume == 1         T  *  F
   //       volume > 20         F  F  *
   //       size == Small       T  F  T
   //       size == Large       F  T  F
   //   EFFECTS
   //       check succeeds      T  T  F
   //       tooLarge            F  F  T
   //       tooHeavy            F  F  F
   //
   // Create a Container of the selected size (in cubic metres)
   SimpleContainer(Sizes containerSize);

   // Return true if the container can hold the load
   public boolean check(int weight, int volume);

   // Return true if the load is too large for the container
   public boolean tooLarge();

   // Return true if the load is too heavy for the container
   public boolean tooHeavy();
```

B.5 Unit Test - Exercise 4

Code the method useWindPower() as defined below.

Then develop unit tests, starting with Equivalence Partitions, and adding extra tests using Boundary Value Analysis and Truth Tables.

```
enum Usage {NONE,LOW,HIGH}

// Determine the amount of wind power to use
// Return NONE for invalid inputs
// Use no wind power if windSpeed<20 or windSpeed>120
// Use low wind power if windSpeed between 20 and 120,
//    and hydro>=100, and load<=1000
// Otherwise, use high wind power

public Usage useWindPower(int windSpeed,int hydro,int load)
```

B.6 Unit Test - Exercise 5

Draw a Control Flow Graph (CFG) for the method startGenerating() shown below, identify the test cases, and derive test data to provide Branch Coverage. Clearly identify all the test cases covered, and include expected outputs. Clearly identify any untestable branches, stating why they cannot be tested.

Specification: a pump storage power station should start generating if:

- capacity>75 and demand>=trigger, or
- capacity>75 and demand>90
- capacity>50 and demand>95

```
 1   public boolean startGenerating(int capacity,
        int demand, int trigger, int current)
 2   {
 3      boolean start=false;
 4      if (capacity>75) {
 5         if (demand>=trigger)
 6            start = true;
 7         else if (demand>(current*9/10))
 8            start = true;
 9      }
10      else if (demand>(current*95/100)) {
11         if (capacity>50)
12            start = true;
13      }
14      else if ((demand>current) && (capacity>80))
15         start = true;
16      return start;
17   }
```

B.7 Unit Test - Exercise 6

A touch-sensitive lamp cycles between off, dim, medium bright, and full brightness each time it is touched. It also has an off button, and a full button.

Draw a State Diagram for class Lamp defined below, and number the transitions. Use the diagram to derive All Transitions tests for the class. Only test transitions that cause a change of state. Clearly show the test cases covered and provide the test data.

```
enum Brightness {OFF, DIM, MEDIUM, FULL};

class Lamp {

   private Brightness currentSetting=Brightness.OFF;

   // if OFF, go to DIM
   // if DIM, go to MEDIUM
   // if MEDIUM, go to FULL
   // if FULL, go to OFF
   public void next()

   // always go OFF
   public void off()

   // if DIM or MEDIUM, go to FULL
   public void full()

    // return the current lamp setting
   public Brightness setting()

}
```

B.8 Unit Test - Exercise 7

A method "convert()" provides support for converting a string into a Java int, subject to limits on its minimum and maximum value.

Use Equivalence Partitioning and Boundary Value Analysis to develop tests for the method convert() defined below.

Then complete coding the method convert().

Finally, run the tests.

```
class DataInput {

   private int value;

   // If s represents a valid Java int, which is >= min and <= max,
   //    then convert sets "value" to the value represented by s and
   //    returns true.
   // Otherwise, value is undefined, and convert() returns false.

   public boolean convert( String s,
                           int min, int max ) {
      // To be completed
   }

   public int getValue() {
      return value;
   }

}
```

B.9 Unit Test - Exercise 8

Devlop tests to provide (a) statement coverage, and (b) branch coverage for check().

```
01 // return true if all the characters in s
02 // are valid numeric characters (0..9)
03
04 boolean check(String s)
05 {
06    boolean ok=true;
07    if (s.length()<1)
08       ok=false;
09    for (int i=0; i<s.length(); i++) {
10       if (s.charAt(i)<0)
11          ok = false;
12       if(s.charAt(i)>9)
13          ok = false;
14    }
15    return ok;
16 }
```

B.10 Unit Test - Exercise 9

Devlop DUP tests for the attribute counter in class CountedString as defined below.

```
01 class CountedString {
02
03    private String value;
04    private int counter;
05
06    // create a counted string with count=0
07
08    CountedString(String s) {
09       value = new String(s);
10       counter = 0;
11    }
12
13    // count how often c appears
14
15    void count(char c) {
16       counter = 0;
17       for (int i=0; i<value.length(); i++)
18          if (value.charAt(i)==c)
19             counter++;
20    }
21
22    // return the count
23
24    int getCounter() {
25       return counter;
26    }
27
28    // reset the counter to zero
29
30    int zeroCounter() {
31       counter = 0;
32    }
33
34 }
```

B.11 Exercise 10 - Test Projects

Complete the following projects.

1. Develop a small GUI program to calculate the area and circumference of a triangle. Use unit tests on the code. Then develop GUI/system tests and run them on the application.

2. Write a method with a fault in that you can't find using EP tests. Write the tests, and confirm that the tests pass. Then add BV tests, and if necessary modify the fault so it cannot be found. Do the same with TT and then Statement Coverage and Branch Coverage.

3. Write a method that accepts two strings, and counts the number of times each character in the second string appears in the first. For example

   count("this is a test","atz")

   would return the array {1,3,0}. Then write black and white box tests to verify it (use EP, BV, SC, and BC).

   Try adding faults to the code in a way that means that the tests still pass (you may need to write extra white box tests to ensure statement coverage or branch coverage). Execute your tests to confirm this. Then try and identify extra tests that catch these faults.

4. Develop a class with two simple methods, one to set the value of an attribute, and another to get its value. Write Black-Box tests for the set method, and White-Box tests for the get method. Then add faults into the code (commission for set, and omission for get) so that the tests continue to pass!

5. Develop a method with a fault that can't be found using EP, SC or BC tests, but which can be found using end-to-end path testing (hint: you need at least two if statements, and a fault which only shows up when a particular pattern of the if statements is executed that might not be executed using Statement or Branch Coverage). Run the tests and confirm it works properly.

6. Design a class that calculates the area of various shapes using floating point (use the double data type). Develop black box tests and execute them. Try and resolve the various problems associated with floating point.

7. Develop your own test runner, using the annotation @SpecialTest, and passing the name of the test class on the command line.

8. In teams, specify and develop a small GUI program using each of the different processes discussed in the book. Suggestion: develop slightly different versions of the program for each process.

Select Bibliography

The following are selected texts that have provided much of the background information for the courses supported by this book:

- P. Ammann and J. Offutt, *Introduction to Software Testing.* Cambridge University Press, 2008
- K. Beck, *eXtreme Programming.* Addison-Wesley, 2000
- R.V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 2000
- P. Hamill, *Unit Test Frameworks.* O'Reilly Press, 2004
- G.J. Myers, *The Art of Software Testing.* John Wiley and Sons, 2004
- S.L. Pfleeger and J.M. Atlee, *Software Engineering: Theory and Practice, 4/E.* Pearson Higher Education, 2010
- M. Roper, *Software Testing.* McGraw-Hill, 1994

The following additional works are cited in the book:

- J.B. Goodenough and S.L. Gerhart, "Toward a theory of test data selection", in *Proceedings of the international conference on Reliable software.* ACM, 1975. pp. 493–510
- ISO/IEC Standard 9126: Software Engineering – Product Quality, part 1. International Organization for Standarization, 2001
- R.B. Grady, *Practical Software Metrics for Project Management and Process Improvement.* Prentice-Hall, 1992
- O. Vinter and P-M. Poulsen, "Experience-driven Software Process Improvement", in *Proceedings of the Conference on Software Process Improvement (SPI 96)*, International Software Consulting Network, 1996
- A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, and C. Zapf, "An Experimental Determination of Sufficient Mutant Operators", *ACM Transactions on Software Engineering Methodology*, Vol. 5, No. 2 (April). ACM, 1996
- J.G.Ganssle, *A Guide to Code Inspections.* The Ganssle Group, 2001

ERRATA

1. Page 4 – "confirm to" should read "conform to"
2. Page 8 – "expected out" should read "expected output"
3. Page 79 – for an output of 300, "16 $\leq$ age $<$ 46" should read "16 $\leq$ age $<$ 45"
4. Page 99 – for tests 56 and 57, "500" should read "300"
5. Page 153 – "MIN(freeSeats+1,totalSeats)" should read "freeSeats+1"
6. Page 62 – "MIN(freeSeats+1,totalSeats)" should read "freeSeats+1"
7. Page 62 – "the MIN() function is used to represent the situation that if freeSeats<totalSeats, then seatsRequired has a partition from freeSeats+1..totalSeats. Otherwise, it just has s single value partition at totalSeats."

   should read

   "If seatsRequired==totalSeats, then freeSeats has a single value partition at totalSeats.

   If freeSeats==totalSeats-1, then seatsRequired has a single value partition at totalSeats. If freeSeats==totalSeats, then this partition does not exist."
8. Page 63 – "MIN(freeSeats+1,totalSeats)" should read "freeSeats+1"
9. Page 64 – "MIN(freeSeats+1,totalSeats)" should read "freeSeats+1"