

---

# **SWEN90006 Software and Security Testing**

**SWEN90006**

**Sep 08, 2021**



# CONTENTS

<b>I</b>	<b>LECTURE NOTES</b>	<b>5</b>
<b>1</b>	<b>Introduction to Software Testing</b>	<b>7</b>
1.1	Learning outcomes of this chapter . . . . .	7
1.2	Chapter Introduction . . . . .	7
1.3	Programs . . . . .	8
1.4	The Psychology of Software Testing . . . . .	15
1.5	Input Domains . . . . .	17
1.6	Black-Box and White-Box Testing . . . . .	18
1.7	Error Guessing . . . . .	19
1.8	Some Testing Laws . . . . .	20
1.9	Bibliography . . . . .	20
<b>2</b>	<b>Input Partitioning</b>	<b>21</b>
2.1	Learning outcomes of this chapter . . . . .	21
2.2	Chapter Introduction . . . . .	21
2.3	Equivalence Classes . . . . .	22
2.4	Partitioning and equivalence classes . . . . .	23
2.5	Domain Testing . . . . .	25
2.6	Equivalence Partitioning . . . . .	29
2.7	Test Template Trees . . . . .	36
2.8	Combining Partitions . . . . .	39
2.9	References . . . . .	41
<b>3</b>	<b>Boundary-Value Analysis</b>	<b>43</b>
3.1	Learning outcomes of this chapter . . . . .	43
3.2	Chapter Introduction . . . . .	43
3.3	Values and Boundaries . . . . .	44
3.4	Guidelines for Boundary-Value Analysis . . . . .	45
<b>4</b>	<b>Coverage-Based Testing</b>	<b>53</b>
4.1	Learning outcomes of this chapter . . . . .	53
4.2	Chapter introduction . . . . .	53
4.3	Control-Flow Testing . . . . .	54
4.4	Data-Flow Testing . . . . .	64
4.5	Mutation Analysis . . . . .	72
4.6	Comparing Coverage Criteria . . . . .	78
<b>5</b>	<b>Testing Modules</b>	<b>81</b>
5.1	Learning outcomes of this chapter . . . . .	81
5.2	Chapter Introduction . . . . .	81

5.3	State and Programs . . . . .	81
5.4	Testability of State-Based Programs . . . . .	82
5.5	Unit Testing with Finite State Automata . . . . .	84
5.6	Testing Object-Oriented Programs . . . . .	90
<b>6</b>	<b>Test Oracles</b>	<b>95</b>
6.1	Learning outcomes of this chapter . . . . .	95
6.2	Chapter Introduction . . . . .	95
6.3	Active and Passive Test Oracles . . . . .	96
6.4	Types of Test Oracle . . . . .	96
6.5	Oracle derivation . . . . .	98
<b>7</b>	<b>Testing-and-Integration</b>	<b>101</b>
7.1	Learning outcomes of this chapter . . . . .	101
7.2	Chapter Introduction . . . . .	101
7.3	Integrating the System . . . . .	102
7.4	Types and Levels of Testing . . . . .	103
<b>8</b>	<b>Security Testing</b>	<b>107</b>
8.1	Learning outcomes of this chapter . . . . .	107
8.2	Chapter Introduction . . . . .	107
8.3	Introduction to Penetration Testing . . . . .	108
8.4	Random Testing . . . . .	110
8.5	Mutation-based Fuzzing . . . . .	111
<b>II</b>	<b>APPENDIX</b>	<b>113</b>
<b>9</b>	<b>A Brief Review of Some Probability Definitions</b>	<b>115</b>
9.1	Random Variables . . . . .	116
9.2	Probability Density Functions . . . . .	117
9.3	Probability in Reliability Measurement . . . . .	118
<b>10</b>	<b>Maximum Likelihood Estimation</b>	<b>119</b>
<b>III</b>	<b>TUTORIALS</b>	<b>121</b>
<b>11</b>	<b>SWEN90006 Tutorial 1</b>	<b>123</b>
11.1	What is testing really? . . . . .	123
11.2	Important terminology . . . . .	123
11.3	Your tasks . . . . .	124
11.4	The programs . . . . .	124
<b>12</b>	<b>SWEN90006 Tutorial 2</b>	<b>127</b>
12.1	Introduction . . . . .	127
12.2	The Program . . . . .	127
12.3	Tasks . . . . .	128
12.4	Java Implementation . . . . .	128
12.5	LWIG Implementation . . . . .	128
12.6	JUnit test script . . . . .	129
<b>13</b>	<b>SWEN90006 Tutorial 3</b>	<b>133</b>
13.1	Introduction . . . . .	133
13.2	Working With the Program . . . . .	133

13.3	Your Tasks . . . . .	134
13.4	The Program . . . . .	134
13.5	JUnit test script . . . . .	136
<b>14</b>	<b>SWEN90006 Tutorial 4</b>	<b>139</b>
14.1	Introduction . . . . .	139
14.2	Working With the Program . . . . .	139
14.3	Your Tasks . . . . .	142
<b>15</b>	<b>SWEN90006 Tutorial 5</b>	<b>145</b>
15.1	Introduction . . . . .	145
15.2	Working With the Program . . . . .	145
15.3	Your Tasks . . . . .	151
<b>16</b>	<b>SWEN90006 Tutorial 6</b>	<b>153</b>
16.1	Introduction . . . . .	153
16.2	Working With the Program . . . . .	153
16.3	Your Tasks . . . . .	158



These notes are available as a [PDF download](#)

Software engineering is about exercising *control* over the software that you are developing. This can be control over the process that is used or the product. In this subject, we focus on the latter: processes and methods for controlling the quality of software.

Like any other part of software engineering, it is important to follow approaches for software testing. If we do a particularly good job of testing in a project, we want to know what processes and methods we used so that we can do a good job in future projects. If we approach software testing in an ad-hoc manner, it makes it harder to repeat our successes.

This subject is about processes and methods for testing the functional correctness and the security of software.

## Subject overview

The aim of this subject is to explore systematic methods for testing software, selecting test inputs to maximise testing coverage and to maximise the likelihood of founding faults.

The subject is largely divided into three parts:

1. Testing for functional correctness. This is what most people think about when they consider testing: running inputs and observing outputs to see if the software conforms to its requirements. We explore how to select good test inputs, how to measure whether these are sufficient, and how to determine if a test has passed (test oracles)
2. Testing for reliability. This is similar to functional correctness, but we consider reliability to be a statistical notion of correctness. We explore reliability models for software, reliability measures, reliability and testing, and how to use reliability engineering principles to achieve more confidence in the correctness of software.
3. Testing for security. Security testing leaves behind the idea of functional correctness, complete, and related topics, and aims to determine: how secure is our system? Here, we will explore common security vulnerabilities, how to test effectively to detect these, and advanced symbolic methods for detecting these. Security testing is different in that we are not testing requirements directly – we are looking for generic vulnerabilities.

## Intended learning outcomes

On completion of this subject the student is expected to:

1. Select appropriate methods to build in quality and dependability into software systems
2. Select and apply effective testing techniques for verifying medium and large scale software systems
3. Select and apply measures and models to evaluate the quality, reliability, and security of a software system
4. Work in a team to evaluate and apply different methods for quality, reliability, and security of a software system

## Generic skills

The subject is a technical subject. The aim is to explore the various approaches to building software with specific attributes into a system. The subject will also aim to develop a number of *generic skills*.

On completion of this subject, students should have the following generic skills:

1. Independent research skills, in order to develop your ability to learn independently, assess what you learn, and apply what you learn.
2. The ability to work as part of a team, following processes and that meet milestones as part of that team.

3. Interpret and assess of quantitative and qualitative data.
4. Critical analysis skills that will complement and round out what you have learned so far in your degree, and help you to think more deeply about all phases of software development.

### Expectations on Students

The subject has formal teaching through lectures and tutorials/workshops. There are also consultation times for you.

### Lectures and Lecture Notes

These subject notes are the main reference for the subject. They contain notes on all of the major topics contained in the subject, as well as a number of appendices that are included for interest and to round fill out the the material in the notes. *Material in the appendices will not be examined*

**You are expected to read the subject notes.**

In addition to the subject notes, additional material will be handed out in lectures. Lectures are aimed at providing you with another view of the material in the subject notes. Lectures will primarily consist of problem-solving exercises and discussions, so it is expected that you will gain a deeper understanding of concepts and material from lectures than from the lecture notes.

**You are expected to attend lectures.**

Material from other subjects such as Algorithms and Data Structures, Object-oriented Software Development, and Software Processes and Management is assumed in the subject notes and in the lecture slides. Reference books are reserved for overnight loan at the Engineering library and for certain topics papers may also be handed out in the lectures.

**The staff of SWEN90006 will expect you to be familiar with the material in the subject notes prior to tutorials and workshops and certainly for assignments and projects.**

### Tutorials/Workshops

The aim of the tutorials is to create a bridge between theory and practice. The tutorial questions are similar in standard to those in the assignments and the final exam. They are also aimed at getting you to think about the ways in which the theory can be applied to practical evaluation problems.

**The staff of SWEN90006 will expect you to actively engage in tutorials by conducting tutorial exercises and in engaging in the discussions.**

### Consultations

Consultations are for you to ask questions about assessment work or to seek a deeper understanding of the subject material. You are encouraged to come to consultations whenever you are having difficulty with the subject matter or the assessment work.

### Assessment



This subject is a blend of theory and practice. The practical assessment consists of three assignments and a team project.

Assessment	Topic	Type	Percentage
Assignment 1	Software testing	Individual	20%
Assignment 2	Choice by students	Individual	5%
Assignment 3	Security testing	Team	25%
Quizzes	All topics	Individual	15%
Exam	All topics	Individual	35%

## Assignments

There are three assignments in the subject. One is worth 5%, and the other two are worth 20% and 25% respectively.

The 5% assignment will require you to submit a multiple-choice question with a sample answer into the *PeerWise* repository: <http://peerwise.cs.auckland.ac.nz/>. The aim of these assignments is to encourage deeper thinking about what you have learned in the subject. It also provides the class with a repository of sample questions during revision, and allows class members to peer review and comment on each others questions and answers.

To get the full 5%, you will be required to demonstrate understanding of the subject content, to provide a quality question, and to provide a helpful answer.

The two larger assignments will require you to apply theory learned in lectures to the analysis and testing of programs. There are limits to testing and analysis of programs — so each of the assignments will ask some technical questions as well as some questions aimed at getting you to explore these limitations.

What we are looking for in assignment work is understanding as well as practical application. Your assignments will only be graded above 80% if you demonstrate both understanding and technical ability adequately.

### In short — we give marks if you show thinking

The aim of the group project is for you to build up your skills in security analysis, your evaluation skills, and your research skills. The group project is worth 25%.

The project will be group-based. You are expected to work as part of a team, and will be assessed as a team. This is to ensure that everyone has ‘skin in the game’ — which simply means that there is an incentive to work as a team instead of as a group pursuing individual interests.

## Quizzes and exams

We will hold 4 quizzes of the semester: week 3, week 6, week 9, and week 12, worth 3%, 4%, 4%, and 4% respectively. These will be held during the lectures and will test materials from the previous 3 weeks.

The examination is a 2 hour exam based on the lecture, tutorial and assignment material. The tutorial questions are written in the same style and of similar difficulty to the exam questions. Being able to answer the assignment questions, tutorial questions, and project work means that you will be well prepared for the exam.

## Hurdles

**Important note:** This subject, like many in the school, has a hurdle on both exam and coursework. This means that to pass the subject, you will need to pass both the coursework and the exam.

Assessment component	Percentage of total	Hurdle
Coursework assignments	50%	25%
Quizzes + Final Examination	50%	25%

**Part I**

**LECTURE NOTES**



## INTRODUCTION TO SOFTWARE TESTING

### 1.1 Learning outcomes of this chapter

At the end of this chapter, you should be able to:

- Discuss the purpose of software testing.
- Present an argument for why you think software testing is useful or not.
- Discuss how software testing achieves its goals.
- Define faults and failures.
- Specify the input domain of a program.

### 1.2 Chapter Introduction

---

#### Footnotes

[1]: Recall from *Software Processes and Management* that the word *assurance* means having confidence that a program or document or some other artifact is fit for purpose. Later in these notes we will try and translate the informal and vague notion of *confidence* into a measure of probability or strict bounds.

---

Testing is an integral part of the process of *assuring*[1] that a system, program or program module is suitable for the purpose for which it was built. In most textbooks on software engineering, *testing* is described as part of the process of *validation* and *verification*. The definitions of validation and verification as described in *IEEE-Std. 610.12-1990* are briefly described below.

---

#### Definition

**Validation** is the process of evaluating a system or component during or at the end of the development process to determine if it satisfies the requirements of the system. In other words, validation aims to answer the question:

**Verification** is the process of evaluating a system or component at the end of a phase to determine if it satisfies the conditions imposed at the start of that phase. In other words, verification aims to answer the question:

---

In this subject, testing will be used for much more than just validating and verifying that a program is fit for purpose. We will use testing, and especially random testing methods, to *measure* the attributes of programs. **Note** that not all attributes of a program can be quantified.

Some attributes, like reliability, performance measures, and availability are straightforward to measure. Others, such as usability or safety must be estimated using the engineer's judgement using data gathered from other sources. For example, we may estimate the safety of a computer control system for automated braking from the reliability of the braking computers and their software.

Before looking at testing techniques we will need to understand something of the *semantics* of programs as well as the *processes* by which software systems are developed.

It is necessary to understand the *semantics* of programs so that, as a tester, we can be more confident that we have explored the program thoroughly. We discuss programs briefly in Section [Programs](#).

## 1.3 Programs

To be a good tester, it is important to have a sound conceptual understanding of the *semantics of programs*. When it comes to software there are a number of possibilities, each of which has its own set of complications.

Firstly, let's consider the simple function given in Figures [1.1](#) written in an imperative programming language like C.

```
void squeeze(char s[], int c)
{
    int i, j;
    for (i = j = 0; s[i] != '\0'; i++) {
        if (s[i] != c) {
            s[j++] = s[i];
        }
    }
    s[j] = '\0';
}
```

The function **squeeze** removes any occurrence of the character denoted by the variable **c** from the array **s**, and squeezes the resulting array together. The semantics of C are such that integers and characters are somewhat interchangeable, in that an element of the type **int** can be treated as a **char**, and vice-versa.

If the squeeze function were written in Haskell then its type would be

$\text{squeeze} : \text{string} \times \text{char} \rightarrow \text{string}$

The input type is  $\text{string} \times \text{char}$  and the output type is  $\text{string}$ . The output type is implicit because the parameter to squeeze is a call by reference parameter.

The set of values in the input type is called the *input domain* and the set of values in the output type is called the *output domain*. For functions like the squeeze function, the input domain is the set of values in the input type, and the output domain is the set of values in the output type.

### Footnotes

[2]: The Fibonacci numbers is a sequence of numbers given by  $1\ 1\ 2\ 3\ 5\ \dots\ f_n\ \dots$  where  $f_n = f_{n-1} + f_{n-2}$ .

The fibonacci function, shown in Figure [1.2](#), takes an integer  $N$  and returns the sum of the first  $N$  Fibonacci numbers [2].

```

unsigned int fibonacci(unsigned int N) {
    unsigned int a = 0, b = 1;
    unsigned int sum = 0;
    while (N > 0) {
        unsigned int t = b;
        b = b + a;
        a = t;
        N = N - 1;
        sum = sum + a;
    }
    return sum;
}

```

Again, if the fibonacci function were written in Haskell its type would be `fibonacci : unsigned int → unsigned int` and so the input domain for the **fibonacci** function is the set of values of type **unsigned int**. Likewise, the output domain for the **fibonacci** function is also the set of values of type **unsigned int**.

The *input domain* to a program is the set of values that are accepted by the program as inputs. For example, if we look at the parameters for the squeeze function they define the set of all pairs (s, c) where s has type array of char and c has type int.

Not all elements of an input domain are relevant to the specification. For example, consider a program that divides a integer by another. If the denominator is 0, then the behaviour is undefined, because a number cannot be divided by 0.

**Note** The input domain can vary on different machines. For example, the input domain to the fibonacci function is the set of values of type unsigned int but this can certainly vary on different machines. Table 1.1 shows how the set of values for parameters of type unsigned int and int respectively differ for different machine word sizes.

Word Size	unsigned int Set of Values	int Set of Values
8 Bit Integers	0 .. 255	-127 .. 128
16 Bit Integers	0 .. 65,535	-32,767 .. 32,768
32 Bit Integers	0 .. 4,294,967,295	-2,147,483,647 .. 2,147,483,648

The output domain for the **fibonacci** function is identical. The design specification of the **fibonacci** function should specify the range of integers that are *legal* inputs to the function.

Determining the input and output domains of a program, or function is not as easy as it looks, but it is a skill that is vital for effectively selecting test cases. That is why we will spend a good deal of time on input/output domain analysis.

Notice also that in the squeeze and **fibonacci** functions, for any input to the function (an element in the input domain) there exists a unique output (element output domain) computed by the function.

### Footnotes

[3]: Recall that the property that for all inputs there exists a unique output is the defining property of a mathematical function.

The function in question is *deterministic*. A function is deterministic if for every input to the function there is a unique output — the output is completely determined by the input[3]. In this case it is easier to test the program because if we choose an input there is only one output to check.

**BUT**, we do not always have deterministic programs. If a program can return one of a number of possible outputs for

any given input then it is *non-deterministic*. Concurrent and distributed programs are often non-deterministic. Non-deterministic programs are harder to check because for a given input there is a set of outputs to check.

Programs may *terminate* or *not terminate*. The *squeeze* and **fibonacci** functions both terminate. In the case of **fibonacci** an output is returned to the calling program and so to test **fibonacci** we can simply execute it and examine the returned value. The function *squeeze* returns a void value so even if it terminates we must still examine the array that was passed as input, because its value may change.

On the other hand a classic example of a non-terminating program is a control loop for an interactive program or an embedded system. Control loops effectively execute until the system is shutdown. In the same way as the *squeeze* function and the **fibonacci** function a non-terminating program may:

- generate observable outputs; or
- not generate any observable outputs at all.

In the former case we can test the program or function by testing that the sequence of values that it produces is what we expect. In the second case we need to examine the internal state somehow.

### 1.3.1 What is software testing?

Testing, at least in the context of these notes, means *executing* a program in order to measure its attributes. Measuring a program's attributes means that we want to work out if the program *fails* in some way, work out its response time or through-put for certain data sets, its mean time to failure (MTTF), or the speed and accuracy with which users complete their designated tasks.

Our point of view is closest to that of IEEE-Std. 610.12-1990, but there are some different points of view. For comparison we mention these now.

- Establishing confidence that a program does what it is supposed to do (W. Hetzel, *Program Test Methods*, Prentice-Hall)
- The process of executing a program with the intent of finding errors (G.J. Myers, *The Art of Software Testing*, John-Wiley)
- The process of analysing a software item to detect the difference between existing and required conditions (that is, bugs) and to evaluate the features of the software item (IEEE Standard for Software Test Documentation, IEEE Std 829-1983)
- The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component (IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990)

The viewpoint in these notes is deliberately chosen to be the broadest of the definitions above, that is, the last item in the list above. The theme for this subject is that testing is not just about detecting the presence of faults, but that testing is about *evaluating attributes of system and its components*. This means software testing methods are used to *evaluate* and *assure* that a program meets all of its requirements, both functional and non-functional.

To be more specific, software testing means executing a program or its components in order to assure:

1. The correctness of software with respect to requirements or intent;
2. The performance of software under various conditions;
3. The robustness of software, that is, its ability to handle erroneous inputs and unanticipated conditions;
4. The security of software, that is, the absence of vulnerabilities and its robustness against different kinds of attack;
5. The usability of software under various conditions;
6. The reliability, availability, survivability or other dependability measures of software; or



7. Installability and other facets of a software release.

#### Remark

- Testing is based on *executing* a program, or its components. Therefore, testing can only be done when parts (at least) of the system have been built.
- Some authors include V&V activities such as audits, technical reviews, inspections and walk-throughs as part of testing. We do not take this view of testing and consider reviews, inspections, walk-throughs and audits as part of the V&V process but not part of testing.

### 1.3.2 The Purpose of Testing

#### Footnotes

[4]: In *Notes on Structured Programming*.

*“Program testing can be used to show the presence of bugs, but never to show their absence!”*

— Edsger W. Dijkstra[4]

This quote states that the purpose of testing is to demonstrate that there is a discrepancy between an implementation and its specification, and that it cannot be used to show that the implementation is correct. The aim of most testing methods is to systematically and actively find these discrepancies in the program.

*Testing and debugging are NOT the same activity.* *Debugging* is the activity of: (1) determining the exact nature and location of a suspected fault within the program; and (2) fixing that fault. Usually, debugging begins with some indication of the existence of an fault. The purpose of debugging is to locate fault and fix them.

Therefore, we say that the aim of testing is to demonstrate that there are faults in a program, while the aim of debugging is to locate the cause of these faults, and remove or repair them.

The aim of *program proving* (aka *formal program verification*) is to show that the program does not contain any faults. The problem with program proving is that most programmers and quality assurance people are not equipped with the necessary skills to prove programs correct.

### 1.3.3 The Language of Failures, Faults, and Errors

To begin, we need to understand the language of *failures*, *faults* and *errors*.

#### Definition

- **Fault:** A fault is an incorrect step, process, or data definition in a computer program. In systems it can be more complicated and may be the result of an incorrect design step or a problem in manufacture.
- **Failure:** A failure occurs when there is a deviation of the observed behaviour of a program, or a system, from its specification. A failure can also occur if the observed behaviour of a system, or program, deviates from its intended behaviour which may not be captured in any specification.
- **Error:** An incorrect internal state that is the result of some fault. An error may not result in a failure – it is possible that an internal state is incorrect but that it does not affect the output.

*Failures and errors are the result of faults* – a fault in a program can trigger a failure and/or an error under the right circumstances. In normal language, software faults are usually referred to as “bugs”, but the term “bug” is ambiguous and can mean to faults, failures, or errors; as such, as will avoid this term.

---

**Footnotes**

[5]: We will not worry about the range just yet.

---

Consider the following simple program specification: for any integer [5]  $n$ ,  $square(n) = n * n$ ; and an (incorrect) implementation of this specification in Figure 1.3

```
int square(int x)
{
    return x*2;
}
```

Executing  $square(3)$  results in 6 – a *failure* – because our specification demands that the computed answer should be 9. The *fault* leading to failure occurs in the statement `return x*2` and the first *error* occurs when the expression `x*2` is calculated.

However, executing  $square(2)$  would not have resulted in a failure even though the program still contains the same fault. This is because the behaviour of the **square** function of under the input 2 behaves correctly. This is called *coincidental correctness*. If 2 was chosen as the test input, then by *coincidence*, this happens to exhibit the correct behaviour, even though the function is implemented incorrectly.

The point here is there are some inputs that reveal faults and some that do not. While the above example is trivial, any non-trivial piece of software will display coincidental correctness for many test inputs.

*In testing we can only ever detect failures.* Our ability to find and remove *faults* in testing is closely tied to our ability to detect failures. The discussion above leads naturally to the following three steps when testing and debugging software components.

- Detect system failures by choosing test inputs carefully;
- Determine the faults leading to the failures detected;
- Repair and remove the faults leading to the failures detected; and
- Test the system or software component again.

This process is itself error-prone. We must not only guard against errors that can be made at steps (2) and (3) but also note that new faults can be introduced at step (3). It is important to realise here that steps (2) and (3) must be subject to the same quality assurance activities that one may use to develop code.

### 1.3.4 Test Activities

Ultimately, testing comes down to the selecting and executing *test cases*. A test case for a specific component consists of three essential pieces of information:

- a set of test inputs, or if the program under test is non-terminating, a set of sequences of test inputs;
- the expected results when the inputs are executed; and
- the execution conditions or execution environment in which the inputs are to be executed.

A collection of test cases is a *test suite*.

As part of the process of testing, there are several steps that need to be performed. These steps remain the same from unit testing to system testing.

#### Test Case Selection

Given the above definition of a test case, there are two generic steps to test case selection. Firstly, one must select the test inputs. This is typically performed using a *test input select technique*, which aims to achieve some form of *coverage criterion*. Test input selection is covered in Chapters 2–5 of these notes.

Secondly, one must provide the expected behaviour of every test input that is generated. This is referred to as the *oracle* problem. In many cases, this oracle can be derived in a straightforward manner from the requirements of the program being tested. For example, a test case that assesses performance of a system may be related to a specific requirement about performance in the requirements specification of that system.

Despite the amount of research activity on software testing, the oracle problem remains a difficult problem, and it is difficult to automate oracles or to assess their quality.

Like any good software engineering process, test case selection is typically performed at a high level to produce abstract test case, and these are then refined into executable test cases.

#### Test Execution

Once executable test cases are prepared, the next step is to execute the test inputs on the program-under-test, and record the actual behaviour of the software. For example, record the output produced by a functional test input, or measure the time taken to execute a performance test input.

Test execution is one step of the testing process that is generally able to be automated. This not only saves the tester time, but allows for regression testing to be performed at minimal case.

#### Test Evaluation

Compare the actual behaviour of the program under the test input with the expected behaviour of the program under that test input, and determine whether the actual behaviour satisfies the requirements. For example, in the case of performance testing, determine whether the time taken to run a test is less than the required threshold.

As with test execution, test evaluation can generally be automated.

## Test Reporting

The final step is the report the outcome of the testing. This report may be returned to developers so they can fix the faults that relate to the failures, or it may be to a manager to inform them that this stage of the testing process is complete.

Again, certain aspects of test reporting can be automated, depending on the requirements of the report.

### 1.3.5 Test Planning

Testing is part of quality assurance for a software system, and as such, testing must be *planned* like any other activity of software engineering. A *test plan* allows review of the adequacy and feasibility of the testing process, and review of the quality of the test cases, as well as providing support for maintenance. As a minimum requirement, a test plan should be written for every artifact being tested at every level, and should contain at least the following information:

- *Purpose*: identifies what artifact is being tested, and for what purpose; for example, for functional correctness;
- *Assumptions*: any assumptions made about the program being tested;
- *Strategy*: the strategy for test case selection;
- *Supporting artifacts*: a specification of any of the supporting artifacts, such as test stubs or drivers; and
- *Test Cases*: an description of the abstract test cases, and how they were derived.

Other information can be included in a test plan, such as the estimate of the amount of resources required to perform the testing.

### 1.3.6 Testability

The *testability* of software can have a large impact on the amount of testing that is performed, as well as the amount of time that must be spent on the testing process to achieve certain test requirements, such as achieving a coverage criterion.

Testability is composed of two measures: *controllability* and *observability*.

---

**Definition: Controllability and Observability**

1. The *controllability* of a software artifact is the degree to which a tester can provide test inputs to the software.
  2. The *observability* of a software artifact is the degree to which a tester can observe the behaviour of a software artifact, such as its outputs and its effect on its environment.
- 

For example, the squeeze from Figure 1.1 highly controllable and observable. It is a function whose inputs are restricted to parameters, and whose output is restricted to a return value. This can be controlled and observed using another piece of software.

Software with a user interface, on the other hand, is generally more difficult to control and observe. Test automation software exists to *record* and *playback* test cases for graphical user interfaces, however, the first run of the tests must be performed manually, and expected outputs observed manually. In addition, the record and playback is often unreliable due to the low observability and controllability.

Embedded software is generally less controllable and observable than software with user interfaces. A piece of embedded software that receives inputs from sensors and produces outputs to actuators is likely to be difficult to monitor in such an environment — typically much more difficult than via other software or via a keyboard and screen. While the embedded software may be able to be extracted from its environment and tested as a stand-alone component, testing software in its production environment is still necessary.

Controllability and observability are properties that are difficult to measure, and are aspects that must be considered during the design of software — in other words, software designers must consider *designing for testability*.

## 1.4 The Psychology of Software Testing

Testing is more an art form than a science. In this subject, we look at techniques to make it as scientific as possible, but skill, experience, intuition and psychology all play an important part of software testing.

Psychology? How does this impact software testing?

We will look at two things related to psychology: (1) the purpose of software testing; and (2) what a successful test case is.

### 1.4.1 The purpose of software testing

In Section *What is software testing?*, we defined testing and looked at its purpose. The famous Dijkstra quote that testing shows the presence of faults, but not their absence, is important.

What are some other commonly stated goals of software testing?:

- To show that our software is correct.
- To find ALL the faults in our software.
- To prove that our software meets its specification.

These may seem like great things to achieve, but should they be our goal in software testing?

Let's face some facts about software. *Software is complex – exceedingly complex*. Even medium-scale software applications are so much more complex than the very largest of other engineering projects. As such, every piece of software has faults. Even the most trivial programs contains faults; perhaps not ones that are likely to occur or have a large impact, but they do. Consider the standard binary search algorithm that we use in lectures throughout the semester. This algorithm was printed in textbooks, used, discussed, and presented for over four decades before it was noted that it contained a fault. The program contains about 10 logical lines of code! It is naive to think that modern web applications could have less than hundreds if not thousands of faults that the developers don't know about.

Further to this, another fact is that almost every program has an infinite number of possible inputs. Think of a program that parses HTML documents: there are an infinite number of HTML documents, but it needs to parse them ALL. No matter how many tests were run on this program, there is no guarantee that the very next test will not fail. As such, Dijkstra's quote is clear: we cannot test every possible input except for the most trivial program (even then, a program that takes two integers as input will take several years to test every combination), so, we cannot *prove* that a program is correct with testing. All we can do is prove that faults are in the software by running tests that fail. Then, we find and fix the faults, trying not to introduce new faults in the process, and hopefully the quality of our software has increased.

Why does psychology matter here? Let's take these two points: (1) *we can't prove a program is correct with testing*; and (2) *any program we test will have faults, and will continue to after we finish testing*.

If our goal is to find all faults or to show our program has no faults, then we are destined to fail at our goal. What is the point of aiming for a goal that we know that we cannot achieve? Psychologically, we know that people perform poorly when asked to do things they know are impossible.

If you task a team to "show that your software is fault free", what would the response be? Most likely, they would select tests that they know pass. This does not improve software quality at all. Quality is only improved when we find faults and fix them.

Psychologically, a much better goal is: *try to break our program*. If we assume that there are faults in the program and set out to find them, we will do a much better job of our goal. Further to this, we will be motivated each time we find a fault, because that was our goal.

*Testing should improve quality. By finding no faults, quality is not improved.*

Testing is a *destructive* process. We try to break our program.

Testing is **NOT** proof.

*Summary:* We must aim to find the faults are present. If we test to show correctness, we will subconsciously steer towards this goal. In addition, we will fail, because testing to show correctness is impossible.

### 1.4.2 What is a “successful” test case?

This is an interesting question. Our first response may be: “a test is successful if it passes”.

But given what we have just learnt, I hope you agree that: *a test is successful if it fails*.

Any test that does not find a fault is almost a waste. Of course, this is an exaggeration. If we are doing a good job of testing and trying to break our program, a passed test gives us some confidence that our program is working for some inputs. Further, we can keep our tests and run them later after debugging to make sure we do not introduce any new faults. This is called *regression testing* (Section 7.3).

Any test that fails (that is, any test that finds a fault), is a chance to improve quality. This is a successful test. A successful test is a more valuable investment than an unsuccessful test. Much of this subject deals with the question: how do we select tests that are more likely to fail than others.

Consider an analogy of medical diagnosis. If a patient feels unwell, and a doctor runs some tests, the successful test is one that diagnosis the problem. Anything that does not reveal a problem is not valuable. We must consider programs as sick patients – they contain faults whether we want them to or not!

*Summary:* A successful test is one that fails. This is how testing can be used to improve quality. Psychologically, we should be pleased when a test fails. It is not a bad thing when a test fails. The fault was there before the test was run, the test just found it!

### 1.4.3 Principles of software testing psychology

In the *The Art of Software Testing* (see reference below), Myers lists 10 principles of software testing. In this section, we will discuss three of these principles, as they are related to the psychology of software testing.

**Principle 1** — A necessary part of a test case is a definition of the expected output or result.

This means that a test case is not just an input. *Before* we run the test, we must also know what the expected output should be. It is not sufficient to run the test, see the output, and only then decide whether that output is correct.

Why not? Psychologically, we run into a problem. An erroneous result can be interpreted as correct because the mind sees what it wants to see. We may have a desire to see the correct behaviour because we don’t want to do any debugging. Or, we may consider that the person who wrote the code is much smarter than us, so surely they wouldn’t have made this error. Or, we may just convince ourselves that this is the correct answer somehow.

On the other hand, if we have the expected output before the test, we compare the actual output with the expected output, and now whether the test fails or not is completely objective. Either the observed output is the same as the expected, or it is not.

Of course, the expected output may be incorrect itself – we make mistakes in testing too. But at the very least, we will now check both the expected output and the program to see which is incorrect.

**Principle 2** — A programmer should avoid attempting to test his or her own program.

This is perhaps the most important of all principles!

First, let me say that I completely disagree with this principle. *A programmer should absolutely test their own code!* They should not be committing code to repositories that has never been testing. However, what the principle really means is: a programmer should avoid being the *only* person to test his or her own program.

Why? There are three main reasons for this; all linked to the psychology of software testing:

- If a programmer missed important things when coding, such as failing to consider a null pointer, or failing to check a divide by zero, then it is quite likely they will also not think of these during testing.

However, another person is less likely to make exactly the same mistakes. So, this duplication helps to find these types of ‘oversight’ faults.

- If a programmer misunderstands the specification they are programming to (e.g. they misunderstand a user requirement), they will implement incorrect code. When it comes to testing, they will still misunderstand the specification, and will therefore write an incorrect test using this misunderstanding. Both the code and the test are wrong, and in the same way. To the test will pass.

However, another person brings an opportunity to interpret the specification correctly. Of course, they may interpret it incorrectly in the same way, but it is less likely that both people will do this rather than just one.

- Finally, recall that testing is a *destructive* process. We aim to find faults; that is, showing that the software is ‘broken’. However, programming is a *constructive* process. We create the software and we try to make it correct.

As a programmer, once we create something that we work so hard on, we do not then want to turn around and break it! Consider some programming assignments that you have worked on. When you were running tests for it, did you hope they pass, or did you hope they fail? Now consider that you were running tests on someone else’s assignment. Would you care so much whether they pass or fail?

Put simply: someone testing their own code will struggle to switch from the constructive process of coding to the destructive process of testing if the code is their own. Psychologically, they will semi-consciously try to avoid testing parts of the code they think are faulty.

This principle works on a team level too. I have talked with software engineering teams in an organisation who test each others code. They take it competitively! They get great pleasure at breaking other teams’ code. They are motivated to find as many problems as possible. Could a team be so pleased at breaking their own code? I would say not.

**Principle 8** — The probability of the existence of more errors in a section of a program is proportional to the numbers already found in that section.

Faults seem to come in clusters. This is for several reasons: complex bits of code are harder to get right, some parts of code are written hurriedly due to time constraints, and just some software engineers are not as good as others, so their parts will be more faulty.

What does this mean? It means that as we test, we find many faults in one part of a program and fix them, and find comparatively fewer in another part and fix them. We should then invest our time in the more faulty regions.

This *may* seem counter-intuitive. After all, if we find only 1-2 faults in one section, there must be so many more out there than in the section we find 25! However, empirical evidence suggests this is not the case. Therefore, best investment is made in these error/fault-prone sections.

## 1.5 Input Domains

To perform an analysis of the inputs to a program you will need to work out the sets of values making up the input domain. There are essentially two sources where you can get this information:

- the software requirements and design specifications; and
- the external variables of the program you are testing.

In the case of white box testing, where we have the program available to us, the input domain can be constructed from the following sources.

- Inputs passed in as parameters;
- Inputs entered by the user via the program interface;
- Inputs that are read in from files;

- Inputs that are constants and precomputed values;
- Aspects of the global system state including:
  - Variables and data structures shared between programs or components;
  - Operating system variables and data structures, for example, the state of the scheduler or the process stack;
  - The state of files in the file system;
  - Saved data from interrupts and interrupt handlers.

In general, the inputs to a program or a function are stored in *program variables*. A program variable may be:

- A variable declared in a program as in the C declarations

```
int base;  
char s[];
```

- Resulting from a read statement or similar interaction with the environment, for example,

```
scanf("%d\n", x);
```

Variables that are inputs to a function under test can be:

- *atomic data* such as integers and floating point numbers;
- *structured data* such as linked lists, files or trees;
- a reference or a value parameter as in the declaration of a function; or
- constants declared in an enclosing scope of the function under test, for example:

```
#define PI 3.14159  
  
double circumference(double radius)  
{  
    return 2*PI*radius;  
}
```

We all try running our programs on a few test values to make sure that we have not missed anything obvious, but if that is all that we do then we have simply not covered the full input domain with test cases. Systematic testing aims to cover the full input domain with test cases so that we have assurance – confidence in the result – that we have not missed testing a relevant subset of inputs.

## 1.6 Black-Box and White-Box Testing

If we do have a clear requirements or design specification then we choose test cases using the specification. Strategies for choosing test cases from a program or function's specification are referred to as *specification-based* test case selection strategies. Both of the following are specification-based testing strategies.

**Black-box Testing** where test cases are derived from the functional specification of the system; and

**White-box Testing** where test cases are derived from the internal design specifications or actual code for the program (sometimes referred to as *glass-box*).

Black-box test case selection can be done without any reference to the program design or the program code. Black-Box test cases test only the functionality and features of the program but not any of its internal operations.



**Footnotes**

[6]: COTS stands for **Commercial Off The Shelf**

- The real advantage of black-box test case selection is that it can be done *before* the implementation of a program. This means that black-box test cases can help in getting the design and coding correct with respect to the specification.

Black-box testing methods are good at testing for missing functions and program behaviour that deviates from the specification. They are ideal for evaluating products that you intend to use in a system such as COTS [6] products and third party software (including open source software).

- The main disadvantage of black-box testing is that black-box test cases cannot detect additional functions or features that have been added to the code. This is especially important for systems that are safety critical (additional code may interfere with the safety of the system) or need to be secure (additional code may be used to break security).

White-box test cases are selected using requirements and design specifications and the code of the program under test. This means that the testing team needs access to the internal designs and code for the program.

The chief advantage of white-box testing is that it tests the internal details of the code and tries to check all the paths that a program can execute to determine if a problem occurs. As a result of this white-box test cases can check any additional code that has been implemented but not specified.

The main disadvantages of white-box testing is that you must wait until after designing and coding the program under test in order to select test cases. In addition, if some functionality of a system is not implemented, using white-box testing may not detect this.

The term “black-box testing” and “white-box testing” are becoming increasingly blurred. For example, many of the white-box testing techniques that have been used on programs, such as control-flow analysis and mutation analysis, are now being applied to the specifications of programs. That is, given a formal specification of a program, white-box testing techniques are being used on that specification to derive test cases for the program under test. Such an approach is clearly black-box, because test cases are selected from the specified behaviour rather than the program, however, the techniques come from the theory of white-box testing.

In these notes, we deliberately blur the distinctions between black-box and white-box testing for this reason.

## 1.7 Error Guessing

Before we dive into the world of systematic software testing, it is worth mentioning one highly-effective test strategy that is always valuable when combined with any other strategy in these notes. This technique is known as *error guessing*.

Error guessing is an *ad-hoc* approach based on intuition and experience. The idea is to identify test cases that are considered likely to expose errors.

The general technique is to make a list, or better a taxonomy (a hierarchy), of possible errors or error-prone situations and then develop test cases based on the list.

The idea is to document common error-prone or error-causing situations and create a defect history. We use the defect history to derive test cases for new programs or systems. There are a number of possible sources for such a defect history, for example:

- **The Testing History of Previous Programs** — develop a list of faults detected in previous programs together with their frequency;

- **The Results of Code Reviews and Inspections** — inspections are not the same as code reviews because they require much more detailed defect tracking than code reviews; use the data from inspections to create

Some examples of common faults include test cases for empty or null strings, array bounds and array arithmetic expressions (such as attempting to divide by zero), and blank values, control characters, or null characters in strings.

Error guessing is not a testing technique that can be assessed for usefulness or effectiveness, because it relies heavily on the person doing the guessing. However, it takes advantage of the fact that programmers and testers both generally have extensive experience in locating and fixing the kinds of faults that are introduced into programs, and can use their knowledge to guess the test inputs that are likely to uncover faults for specific types of data and algorithm.

Error guessing is ad-hoc, and therefore, not *systematic*. The rest of the techniques described in these notes are systematic, and can therefore be used more effectively as quality assurance activities.

## 1.8 Some Testing Laws

Here are some interesting “laws” about software testing. They are not really laws per se, but rules of thumb that can be useful for software testing.

**Dijkstra’s Law:** Testing can only be used to show the presence of errors, but never the absence of errors

**Hetzel-Myers Law:** A combination of different V&V methods out-performs any single method alone.

**Weinberg’s Law:** A developer is unsuited to test their own code.

**Pareto-Zipf principal:** Approximately 80% of the errors are found in 20% of the code.

**Gutjar’s Hypothesis:** Partition testing, that is, methods that partition the input domain or the program and test according to those partitions, is better than random testing.

**Weyuker’s Hypothesis:** The adequacy of a test suite for coverage criterion can only be defined intuitively.

## 1.9 Bibliography

G.J. Myers, *The Art of Software Testing*, John Wiley & Sons, 1979.

D.E. Knuth, *The Art of Computer Programming*, vol. 2: *Semi-numerical Algorithms*, 2nd Ed., Addison Wesley, 1981.

B. Beizer, *Software Testing Techniques*, 2nd ed., van Nostrand Reinhold, 1990.

E. Kit, *Software Testing in the Real World*, Addison-Wesley, 1995.

R. Hamlet, Random Testing, In *Encyclopedia of Software Engineering*, J. Marciniak ed., pp. 970–978, Wiley, 1994.

A. Endres and D. Rombach, *A Handbook of Software and Systems Engineering*, Addison-Wesley, 2003.

J. A. Whittaker, *How to Break Software: A Practical Guide to Testing*, Addison-Wesley, 2002.

## INPUT PARTITIONING

### 2.1 Learning outcomes of this chapter

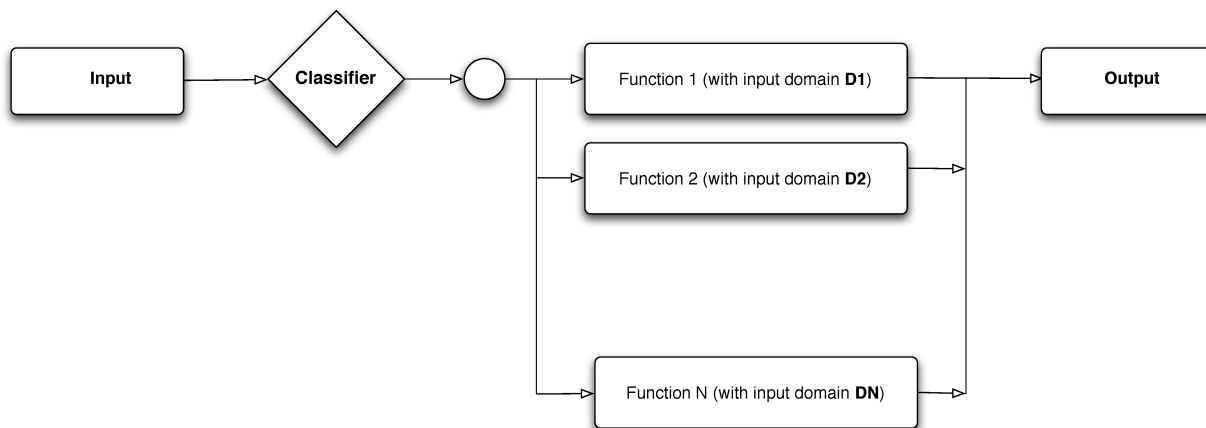
At the end of this chapter, you should be able to:

- Describe the purpose of equivalence partitioning.
- Define valid input domain, invalid input domain, and non-testable input domain Apply testing guidelines to define new equivalence classes for a program based on its functional specification.
- Using the test template tree approach to generate equivalence classes for a program based on its functional specification
- Given a set of equivalence classes, generate tests from those equivalence classes

### 2.2 Chapter Introduction

Recall from Section *Input Domains*, that each program has an *input domain* — the set of values that can be used as input to the program. Test input selection is all about selecting the values in this domain that have the highest likelihood of producing failures.

Using input partitioning, programs are considered as the composition of an *input classifier*, that classifies the input domain into one of a number of different classes where each input class computes one function of the overall program.



The canonical representation of a program is the set  $\{(D_1; F_1), (D_2; F_2), \dots\}$ , where  $D_i$  is the  $i^{th}$  domain and  $F_i$  is the corresponding function computed by the program. There is a one-to-one correspondence between the input domains and

the functions computed by a program. Each of the functions computed by a program occurs along a *program path*, that is, a path in the program that executes a sequence of statements for computing the function. In domain testing, an input domain is the subset of all inputs that will trigger a specific program function to be computed along a specific *program path*.

As an example, consider the program in Figure 2.2, which calculates the minimum of two integers,  $x$  and  $y$ . There are two paths in this program. The first executes the statements at lines 3, 4, 5, and 7, while the second executes at lines 3, 4, and 7. Each of these can be conceived to be its own function. The first one returns the value of  $y$ , while the second returns the value of  $x$ . The input classifier executes the first function if  $x$  is greater than  $y$ , and executes the second otherwise.

```
1. int min(int x, int y)
2. {
3.     int minimum = x;
4.     if (x > y) {
5.         minimum = y;
6.     }
7.     return minimum;
8. }
```

Based on this view of a program, there are two possible types of faults:

- *Computation faults*: where the correct path is chosen but an incorrect computation occurs along that path; and
- *Domain faults*: where the computation is correct for each path but an incorrect path is chosen.
- The possible causes for an incorrect path are: (1) the incorrect path is executed for the input domain; (2) the decisions that make up the path selection may contain a fault; and (3) the correct path (or a fragment of a path necessary for the computation) may simply be missing.

The aim of input partitioning is to derive test inputs that exercise each function of the program at least once. In this chapter, we present two methods for partitioning the input into *equivalence classes*.

## 2.3 Equivalence Classes

In functional testing, we can systematically derive equivalence classes and test inputs that are more likely to find faults than by using random testing. Using these techniques, we have a better idea of what constitutes an equivalence class.

---

### Definition

An *equivalence class* is a set of values from the input domain that are considered to be as likely as each other to produce failures for a program. Relating this to Figure 2.1, there are  $N$  equivalence classes: one for each function. Intuitively, this says that, for an equivalence class, each element in that equivalence class should execute the same statements in a program in the same order, but just with different values.

---

In other words, a single test input taken from an equivalence class is representative of *all* of the inputs in that class.

Recall the Min function from Figure 2.2. In that program, all inputs that satisfy the relation  $x > y$  will execute one path (inside the if statement), while the rest, which satisfy the relation  $x \leq y$ , will execute another (not executing inside the if statement). Therefore, all values that satisfy  $x > y$  form one equivalence class, while the rest form another.

### 2.3.1 Input conditions, valid inputs, and invalid inputs

Each equivalence class is used to represent certain *input conditions* on the input domain. Equivalence partitioning tries to break up input domains into sets of *valid* and *invalid* inputs based on these input conditions. Unfortunately, the terms “input condition”, “valid” and “invalid” are not always used consistently but the following definition will be used in these notes to give a consistent meaning of the terms for this subject.

---

**Definition**

- An *input condition* on the input domain is a predicate on the values of a single variable in the input domain. When we need to consider the values of more than one variable in the input domain, we refer to this as the *combination of input conditions*.
  - A *valid* input to a program is an element of the input domain that is the value expected from the program specification; that is, a non-error value according to the program specification.
  - An *invalid* input is an input to a program is an element of the input domain that is not expected or an error value that as given by the program specification. In other words, an invalid input is a value in the input domain that is not a valid input.
- 

The class of invalid inputs can be further broken down into two sub-classes: those that are *testable*; and those that are *not testable*.

A non-testable input is one that violates the *precondition* of a program. A precondition is a condition on the input variables of a program that is *assumed to hold* when that program is executed. This means that the programmer has made an explicit assumption that the precondition holds at the start of the program, and as such, the behaviour of the program for any input that violates the precondition is *undefined*. If the behaviour is undefined, then *we do not test for it*.

A testable invalid input is an invalid input that does not violate the precondition. Typically, these refer to inputs that return error codes or throw exceptions.

The space of inputs can be modelled as in Figure 2.3.

---

**Example 5**

As an example of the above, consider a *binary search* function, which, given a *sorted* list of numbers, determines whether or not a specific number is in that list. The efficiency of a binary search relies on the fact that the input list is sorted. Thus, the precondition of the function is that the list is sorted. There is little point writing tests with unsorted lists, because the behaviour of a binary search is undefined for an unsorted list.

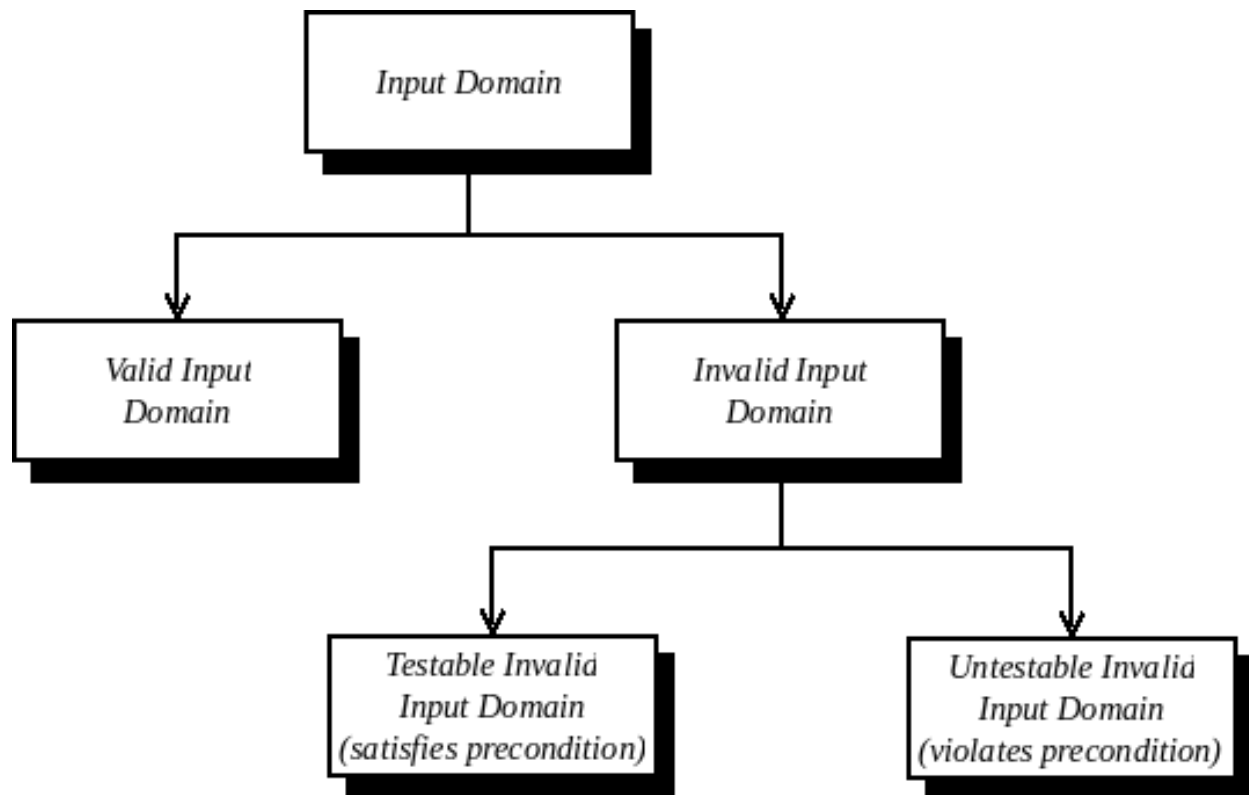
The input condition on the list is a predicate specifying that the list is sorted. Any sorted list is a valid input, while any unsorted list is an invalid and untestable input.

---

## 2.4 Partitioning and equivalence classes

Input partitioning is then a systematic method for identifying interesting input conditions to be tested. To make our use of the term input condition consistent with other literature we will assume that an input condition can be applied to set of values of a specific input variable, or a set of input variables as well.

The assumption behind equivalence partitioning is that all members of the class behave in the same way with respect to failures, so choosing one member of an equivalence class has the same likelihood of detecting a failure as any other member of the equivalence class. This assumption is not correct, but nonetheless, partitioning the input domain into equivalence classes is a valuable technique for testing when combined with other techniques.



There are two key properties equivalence classes for software testing. If we have  $n$  equivalence classes  $EC_1, \dots, EC_n$  for an input domain,  $ID$ , the following two properties must hold:

- for any two equivalence classes,  $EC_i$  and  $EC_j$ , such that  $i \neq j$ ,  $EC_i \cap EC_j = \emptyset$ ; that is,  $EC_i$  and  $EC_j$  are *disjoint*; and
- $\bigcup \{EC_1, \dots, EC_n\} = ID$ ; that is, the testable input domain is covered.

Together, these two properties say that the equivalence classes are disjoint and they cover the input domain. In other words, for any value in the input domain, that input belongs to *exactly one* equivalence class.

The key question to answer now is: *what constitutes a good equivalence class*? This chapter presents two systematic techniques for partitioning the input domain of a program into equivalence classes. The first one relies on the underlying structure of the program to determine which inputs execute the same program statements — therefore, it is a *white-box* testing technique. The second technique relies on the specified functionality of the program — therefore, it is a *black-box* testing technique. To avoid confusion between the two, we will refer to the former as *domain testing*, and the latter as *equivalence partitioning*.

These two techniques are related, but the underlying concepts are different. Domain testing determines the *actual* equivalence classes of a program, while equivalence partitioning, on the other hand, determines the *expected* equivalence classes of a program from its functional requirements.

## 2.5 Domain Testing

In this section, we present *domain testing*. Domain testing partitions the input of a program using the program structure itself. The test inputs are executed on the program, and the tester evaluates whether the output of the program is in the expected output domain.

To select test cases according to the domain testing strategy, the input space is first partitioned into a set of mutually exclusive equivalence classes. Each equivalence class corresponds to a program path. The idea is to select test cases based on the domain boundaries and to explore domain boundaries in order to work out whether or sub-domains are correct or not. More precisely, test cases are selected from: (1) the domain boundaries; and (2) points close to the domain boundaries.

We will need some standard terminology for domain testing.

- **Relational Expressions:** are of the form  $A \text{ rel } B$ , where *rel* denotes a relational operator, that is, one of  $>$ ,  $<$ ,  $\geq$ ,  $\leq$ ,  $=$  and  $\neq$ .
- **Predicates:** are points in a program where the control flow can diverge.
- **Simple Predicates:** are single relational expressions.
- **Compound Predicates:** are predicates composed of several simple predicates combined by Boolean operators such as AND ( $\wedge$ ), OR ( $\vee$ ) and NOT ( $\neg$ ).
- **Linear Simple Predicates:** are simple predicates of the form  $(a_1x_1 + \dots + a_nx_n) \text{ rel } k$ , where *rel* is a relational operator,  $x_1, \dots, x_n$ , and the  $a_i$ s and  $k$  are constants.
- **Path Condition:** is the condition that must be satisfied by the input data for that path to be executed.

### Example 6

As an example consider the program in Figure 2.4. The domains of this program are calculated by looking at the different paths in the program. In this example, there are two if/then/else statement, which gives us four possible paths. Table 2.1 outlines the four path conditions for each path, which correspond to the predicate in the branches.

```
int f(float x, y)
{
    float w, z;

    w = x + y;

    if (x + y >= 2) {
        z = x - (3 * y);
    }
    else {
        z = 2x - 5y;
    }

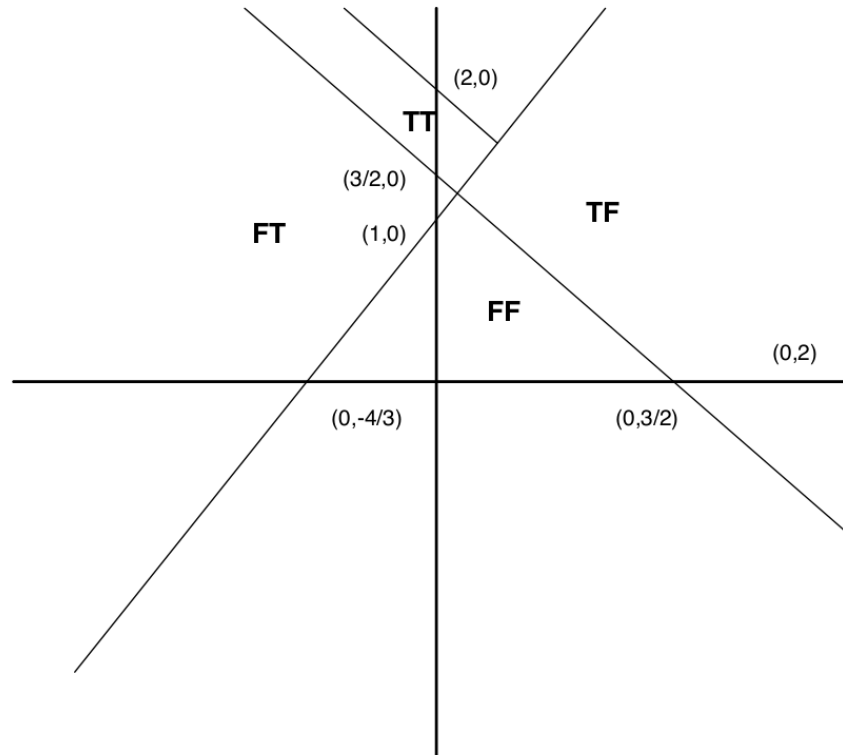
    if (z + w > 3) {
        z = f(x, y, z);
    }
    else {
        z = g(x, y, z);
    }

    return z;
}
```

Considering that a domain is the set of inputs that satisfy a path condition, then four path conditions gives us four domains. Each domain is defined by a *linear border*. These are shown Figure 2.5. In this figure, TT refers to the domain in which both branches in the program evaluate to true. TF, TF, and FF are defined similarly to other possible evaluations.

$$\begin{array}{ll} \text{C1a} & x + y \geq 2 \\ \text{C1b} & x + y < 2 \\ \text{C2a} \text{ (C1T)} & 2x - 2y > 3 \\ \text{C2b} \text{ (C1F)} & 3x - 4y > 3 \end{array}$$

If the branches of a program are such that certain paths are *infeasible*, then the domain for that path is empty.




---

**Remark**

We note that domain testing makes an assumption that programs are *linearly domained*.

---

Linear equalities are expressions that can be put into the form

$$a_1x_1 + \dots + a_nx_n = k \text{ or } a_1x_1 + \dots + a_nx_n \neq k$$

and linear inequalities are expressions that can be put into the form

$$a_1x_1 + \dots + a_nx_n \leq k \text{ or } a_1x_1 + \dots + a_nx_n \geq k$$

or

$$a_1x_1 + \dots + a_nx_n < k \text{ or } a_1x_1 + \dots + a_nx_n > k.$$



Linear borders require fewer test points to check and can be checked with more certainty.

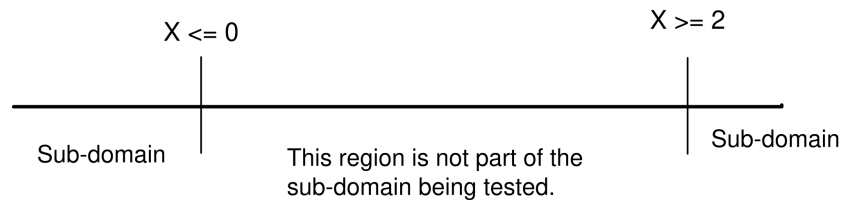
If a program violates this assumption then testing using this strategy may not be as productive at finding faults as would have been the case if the program did satisfy the assumptions, because solving non-linear equalities and inequalities is considerably more difficult than solving linear equalities and inequalities.

In the non-linear case a point may lie in a domain but be computationally too expensive to check accurately; for example, a polynomial:

$$a_1x_1^y + \dots + a_nx_n \leq k$$

## 2.5.1 Compound Predicates

Simple predicates contain just a set of variables and a relational operator such as  $\leq$ ,  $\geq$ ,  $>$ ,  $<$ ,  $=$  or  $\neq$ . Simple predicates create *simple boundaries*. When we use *compound predicates* to define boundaries then the situation becomes trickier to test. A compound predicate with a logical conjunction (**AND** operator) is straight-forward because it often defines a convex domain where all of the points lie within a boundary. A compound predicate with a logical disjunction (**OR** operator) can create disjointed boundaries, for example,  $x \leq 0 \vee x \geq 2$  creates the boundaries as in Figure 2.6



To overcome this problem, domain testing treats compound predicates such as logical disjunction as different paths. That is, in the example in Figure 2.6, the cases of  $x \leq 0 \vee x \geq 2$  are treated as two separate functions. Each compound predicate is broken into *disjunctive normal form*; that is, a disjunction of one or more simple predicates.

To do this, each compound predicate is reduced to a predicate using only disjunction and negation, using *DeMorgan's Laws*. For example, using the law

$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$

we can reduce the expression  $\neg(x \leq 0 \vee x \geq 2)$  to  $x > 0 \wedge x < 2$ .

## 2.5.2 Loops in Programs

If each program path corresponds to a function, then for some programs containing loops, we have an infinite number of paths, and therefore, an infinite number of equivalence classes. Selecting test inputs for each of these classes is clearly impossible, so we need a technique for making this finite.

A typical work around is to apply the *zero-to-many* rule. The zero-to-many rule states that the minimum number of times a loop can execute is zero, so this is one path. Similarly, some loops have an upper bound, so treat this as a path. A general guideline for loops is to select test inputs that execute the loop:

- zero times – so that we can test paths that do not execute the loop;
- once – to test that the loop can be entered and that the results for a single iteration are correct;
- twice – to test that the results remain correct between different iterations of the loop;

- N times (greater than 2) – to test that an arbitrary number of iterations returns the correct results; and
- N+1 times to test that after an arbitrary number of iterations the results remain correct between iterations.

In the case that we know the upper bound of the loop, then set N+1 to be this upper bound. Therefore, we are partitioning the input domain based on the number of times that a loop executes. We need only to instantiate the number, N, when selecting the test inputs that satisfy the equivalence classes.

### Example 8 – Selecting Test Cases for the Triangle Program

Consider a program, which takes as input, three integers, x, y, and z, each representing the length of one side of a triangle, and classifies the triangle as one of the following:

- *invalid*: at least one side does not have a positive length, or one side is greater than the sum of the other two sides;
- *equilateral*: all sides are the same length;
- *isosceles*: two sides are the same length; or
- *scalene*: all sides are of different length.

An implementation of this program is shown below.

```
type enum Triangle = {equilateral, isosceles, scalene, invalid}

Triangle categorise(int x, int y, int z)
{
    if (x > 0 && y > 0 && z > 0 &&
        x + y >= z && x + z >= y && y + z >= x) {
        if (x == y)
            if (y == z) return equilateral;
            else return isosceles;
        else if (y == z) return isosceles;
        else if (x == z) return isosceles;
        else return scalene;
    }
    else return invalid;
}
```

To derive the paths for this program, we look at the branches in the program source. For simplicity, we abbreviate the predicate in the first branch to  $\text{valid}(x, y, z)$ .

The valid paths in the program are straightforward to identify. The equivalence class that corresponds to an equilateral triangle is characterised by the predicate  $\text{valid}(x, y, z) \wedge x = y \wedge y = z$ . For the isosceles case, there are three equivalence classes:

- $\text{valid}(x, y, z) \wedge x = y \wedge y \neq z$
- $\text{valid}(x, y, z) \wedge x \neq y \wedge y = z$
- $\text{valid}(x, y, z) \wedge x \neq y \wedge y \neq z \wedge x = z$

Finally, the equivalence class for the scalene triangle is  $\text{valid}(x, y, z) \wedge x \neq y \wedge y \neq z \wedge x \neq z$

The invalid case is more difficult to partition. The false condition of the first branch is taken if  $\neg \text{valid}(x, y, z)$ , which is equivalent to

$$\neg(x > 0 \wedge y > 0 \wedge z > 0 \wedge x + y \geq z \wedge x + z \geq y \wedge y + z \geq x)$$

Using DeMorgan's laws to reduce this into disjunctive normal form, we are left with six cases: the negation of each of the simple predicates above.

Combining the invalid with the valid cases gives us 11 abstract test cases, which must be then instantiated with actual values. For example, choose the test inputs  $x = 3 \wedge y = 3 \wedge z = 3$  for the equilateral case.

---

## 2.6 Equivalence Partitioning

In this section, we present *equivalence partitioning*. Equivalence partitioning partitions the input of a program using the functional requirements of the program. The test inputs are executed on the program, and the tester evaluates whether the output of the program is in the expected output domain.

Equivalence partitioning is one of the oldest and still most widely used method for selecting test cases based on a partitioning of the input domain.

The aim is to minimise the number of test cases required to cover all of the equivalence classes that you have identified. The general method only needs three steps:

- Identify the initial equivalence classes (ECs);
- Identify overlapping equivalence classes, and eliminate them by making the overlapping part a new equivalence classes; and
- Select one element from each equivalence class as the test input and work out the expected result for that test input.

### 2.6.1 Step 1 - Identify the initial equivalence classes

Unfortunately, there is no clear cut algorithm or method for choosing test inputs according to equivalence partitioning. You will still need to build up some judgement and intuition. There are however, a good set of guidelines to get you going. Again, the guidelines are just suggestions for picking good equivalence classes – the guidelines work in many cases but there are always exceptions where you will need to exercise your own creativity.

The following set of guidelines will help you to identify *potential* equivalence classes. As you test a program you may need to choose extra test cases in order to explore different subsets of the input domain more thoroughly. The more you apply the guidelines and see the effect, the easier it will be for you to start choosing good classes for testing - practice will give you insight and experience.

### 2.6.2 Guidelines for identifying equivalence classes

#### Guideline 1: Ranges

If an input condition specifies a range of values, identify one valid equivalence class for the set of values in the range, and two invalid equivalence classes; one for the set of values below the range and one for the set of values above the range.

---

#### Example 9

If we are given the range of values 1..99, then we require three equivalence classes:

- The valid equivalence class 1..99; and
  - The two invalid equivalence classes  $\{x \mid x < 1\}$  and  $\{x \mid x > 99\}$ .
-

**Guideline 2: Discrete sets**

If an input condition specifies a set of possible input values and each is handled differently, identify a valid equivalence class for each element of the set and one invalid equivalence class for the elements that are not in the set.

---

**Example 10**

If the input is selected from a set of vehicle types, such as  $\{BUS, TRUCK, TAXI, MOTORCYCLE\}$ , then we require:

- Four valid equivalence classes; one for each element of the set  $\{BUS, TRUCK, TAXI, MOTORCYCLE\}$ ; and
  - One invalid equivalence class for an element “outside” of the set; such as *TRAILER* or *NON\_VEHICLE*.
- 

**Guideline 3: Number of inputs**

If the input condition specifies the number (say  $N$ ) of valid inputs, define one valid equivalence class for the correct number of inputs and two invalid equivalence classes – one for values  $< N$  and one for values  $> N$ .

---

**Example 11**

If the input condition specifies that the user should input exactly three preferences, then define one valid equivalence class for three preferences, and two invalid equivalence classes: one for  $< 3$  and one for  $> 3$ .

---

**Guideline 4: Zero-one-many**

1. If the input condition specifies that an input is a collection of items, and the collection can be of varying size; for example, a list or set of elements; define one valid equivalence class for a collection of size 0, one valid equivalence class for a collection of size 1, and one valid equivalence class for a collection of size  $> 1$ .

Additionally, if the collection has *bounds*, such as a maximum number  $N$ , define one valid equivalence class for a collection of size  $N$ , and one invalid equivalence class for a collection of size  $> N$ . Similarly for lower bounds, if they exist.

---

**Example 12**

If the input condition specifies that the user should input a list of their preferences, with no bound on the list, then define valid equivalence classes for lists of size 0, 1, and 5.

If the input condition is further constrained by a maximum size of 10, then define valid equivalence classes for lists of size 0, 1, 5, and 10, as well as an invalid equivalence class for a list of size 11.\*

---

**Guideline 5: Must rule**

If an input condition specifies a “must be” situation, identify one valid equivalence class and one invalid equivalence class.

**Example 13**

If the first character of an input *must be* a numeric character, then we require two equivalence classes – a valid class

$$\{s \mid \text{the first character of } s \text{ is a numeric}\}$$

and one invalid class

$$\{s \mid \text{the first character of } s \text{ is not a numeric}\}$$

**Guideline 6: Intuition/experience/catch-all!**

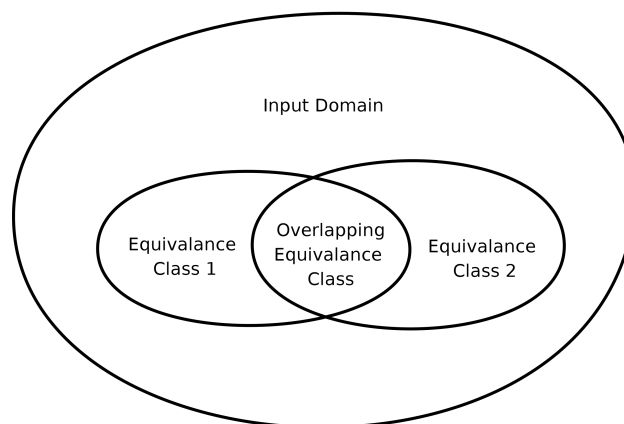
If there is any reason to believe that elements in an equivalence class are handled in a different manner than each other by the program, then split the equivalence class into smaller equivalence classes.

This is a sort of “default” catch statement, which really says that you can derive tests based on intuition or your understanding of the program and domain, where none of the other guidelines fit.

Notice that this particular set of guidelines has some very general rules such as the last one. In short, you will need to build up some judgement and experience in order to become good at selecting test cases.

**2.6.3 Step 2 - Eliminating overlapping equivalence classes**

If we apply the guidelines in Step 1, it is likely that we will end up with overlapping equivalence class. Figure 2.7 shows an input domain that has two equivalence classes that overlap.



Returning to our example of a binary search function, we may identify the following equivalence classes (in fact, we should identify even more):

- an empty list;
- a list with exactly one element;
- a list with at least two elements;
- a sorted list; and

- a unsorted list (an invalid input).

Equivalence classes A, B, and C are disjoint, as are D and E. However, a list with at least two elements can both sorted or unsorted, so there is overlap between classes C and D, as well as C and E.

Once these overlapping classes have been identified, they are eliminated by treating them as equivalence classes themselves. So, for the equivalence classes in Figure 2.7, the overlapping class becomes an equivalence class of its own, and the other two equivalence classes are re-calculated by removing the values in the overlapping class.

In the binary search example, we identify the overlap, and re-calculate the new classes. As it turns out, a list with two or more elements must be either sorted or unsorted, so the set of values in class C is equal to the set of values in the union of D and E. As a result, we end up with the following equivalence classes:

1. an empty list;
2. a list with exactly one element;
3. a unsorted list with at least 2 elements;
4. a sorted list with at least 2 elements;

In this example, we end up with *less* equivalence classes, however, in other cases we can end up with more.

## 2.6.4 Step 3 - Selecting test cases

Once the equivalence classes have been identified, selecting the test cases is straightforward. Any value from an equivalence class is identified to be as likely to produce a failure as any other value in that class, therefore *any element of the class serves as a test input*, and selecting any arbitrary element from the class is adequate.

---

### Example 14: Selecting Test Cases for the Triangle Program

Recall the triangle program from Example 8. Consider the following *informal specification* for that program:

The program reads three integer values from the standard input. The three values are interpreted as representing the lengths of the sides of a triangle. The program then prints a message to the standard output that states whether the triangle is

1. *invalid*: at least one side does not have a positive length, or one side is greater than the sum of the other two sides;
2. *equilateral*: all sides are the same length;
3. *isosceles*: two sides are the same length; or
4. *scalene*: all sides are of different length.

The triangle classification program requires three inputs to the program as well as the form of the output. Further we require that each of the inputs “must be” a positive integer. Table 2.2 summarises the input conditions.

Requirement	Input Condition
Input three integers	Let the three integers be $x$ , $y$ and $z$ ; then $x \in \text{int}$ , $y \in \text{int}$ and $z \in \text{int}$ are the input conditions.
Integers must be positive	$x > 0$ , $y > 0$ and $z > 0$

Now we can use the guidelines and the specification to determine valid and invalid equivalence classes for the input conditions. Guideline 1 is appropriate as we have a range of values. If the three integers we have called  $x$ ,  $y$  and  $z$  are all

greater than zero then there are three valid equivalence classes:

$$\begin{aligned} EC_{valid_1} &= \{x \mid x > 0\} \\ EC_{valid_2} &= \{y \mid y > 0\} \\ EC_{valid_3} &= \{z \mid z > 0\} \end{aligned}$$

for the input conditions  $x > 0$ ,  $y > 0$  and  $z > 0$ . However, these are not disjoint. After eliminating the overlapping cases, we are left with only one equivalence class:

$$EC_{valid} = \{(x, y, z) \mid x > 0 \wedge y > 0 \wedge z > 0\}$$

Adding the constraint that each side cannot be greater than the sum of the other two sides, and eliminating the equivalence classes leaves us with only one equivalence class again:

$$EC_{valid} = \{(x, y, z) \mid x > 0 \wedge y > 0 \wedge z > 0 \wedge x + y \geq z \wedge x + z \geq y \wedge y + z \geq x\}$$

The output domain consists of isosceles, scalene, equilateral and invalid. Now we will need to apply guideline 3 because different values in the input domain map to different elements of the output domain. For example, the equivalence class for equilateral triangles would be

$$EC_{equilateral} = \{(x, y, z) \mid x = y = z\}$$

If we create one equivalence class for each of the different values in the output domain, we split the initial equivalence classes into three disjoint classes.

For the invalid classes we need to consider the case where each of the three variables in turn can be negative, or longer than the other two sides:

$$\begin{aligned} EC_{Invalid_1} &= \{x \mid x \leq 0\} \\ EC_{Invalid_2} &= \{y \mid y \leq 0\} \\ EC_{Invalid_3} &= \{z \mid z \leq 0\} \\ EC_{Invalid_4} &= \{(x, y, z) \mid x + y \leq z\} \\ EC_{Invalid_5} &= \{(x, y, z) \mid x + z \leq y\} \\ EC_{Invalid_6} &= \{(x, y, z) \mid y + z \leq x\} \end{aligned}$$

Again, there is overlap here. It is possible that both  $x$  and  $y$  can be less than 0. So, the overlapping cases need to be eliminated. This is problematic, because there are a number of combinations here. This is discussed further in Section [Combining Partitions](#). For now, we will add just one test case from each of the (overlapping) classes above.

According to the equivalence partitioning method we only need to choose one element from each of the classes above in order to test the triangle program. Table 2.3 gives the test cases that are obtained after **Step 3** has been completed.

Equivalence class	Test Input	Expected Outputs
$EC_{equilateral}$	(7, 7, 7)	equilateral
$EC_{isosceles}$	(2, 3, 3)	isosceles
$EC_{scalene}$	(3, 5, 7)	scalene
$EC_{invalid_1}$	(-1, 2, 3)	invalid
$EC_{invalid_2}$	(1, -2, 3)	invalid
$EC_{invalid_3}$	(1, 2, -3)	invalid
$EC_{invalid_4}$	(1, 2, 5)	invalid
$EC_{invalid_5}$	(1, 5, 2)	invalid
$EC_{invalid_6}$	(5, 2, 1)	invalid

Selecting just *any* value in an equivalence class may not be *optimal* for finding faults. For example, consider the faulty implementation of the square function shown in Figure 1.3, Section 1.3.2. Recall from the discussion in Section [The](#)

*Language of Failures, Faults, and Errors* that the test input 2 for this function would not reveal a failure due to coincidental correctness. In the absence of any other information about the square function such as the maximum integer input or the largest possible result, the input domain and output domains for square are simply int and the valid equivalence class is some bounded subset of int. Selecting a test input of does not reveal a fault in the function, but the tester cannot know this.

*Is there a better way to select test inputs from equivalence classes to give a greater likelihood of finding faults?*

Clearly, we can select *more than one input* from an equivalence class, which reduces the chances of coincidental correctness, but increases the size of our test suite. Hierons discusses how, in some cases, coincidental correctness in *boundary-value analysis* (see Chapter 3) can be avoiding using information in the specification. However, his ideas do not work for the square function.

---

### Example 15: Selecting Test Cases for the GetWord Function

Consider the following specification of a function, GetWord, that returns the next word in a string:

1. The GetWord function must accept a string and return two items: (1) the first word in the string; and (2) the string with the first word removed.
2. Words are considered to be a string of characters not containing a blank character.
3. The blank characters between words can be spaces, tabs, or newline characters.
4. Strings can be a maximum of 200 characters in length.

An algorithm for GetWord is given in Figure 2.8

There are at least three steps in implementing a specification:

- Step 1: choose a design that will implement all of the requirements in the specification;
- Step 2: choose algorithms and data structures that implement the functionality in the design;
- Step 3: write the actual program, using some programming, that implements the algorithm.

In step 2 and step 3 there are choices and sometimes the programming language constructs that are chosen to implement an algorithm can introduce subtle behaviours that the specification writer did not anticipate.

The algorithm in Figure 2.8 makes some *design choices* about how to achieve the requirements. Requirement (1) requires a function that takes strings and returns a pair of strings: one for the word, and one for the rest of the line.

The algorithm implements this requirement by a function that takes a string as an input argument, returns a string (the first word), and the input string (the parameter line) to return the input with first word removed. Doing this is a distinct design choice.

```
string GetWord(string line)
    while (line[next] is a blank character) do
        next = next + 1;
    if ( line[next] is not the end of the line ) then
        while ( line[next] is not a blank character ) do
            word[i] = line[next];
            next = next + 1;
            i = i + 1;
        return word;
        line = line[next . . . length(line)];
    else
        return "";
```

Equivalence partitioning requires only the specification of the program to derive test cases. This makes it ideal for system level testing and integration level testing. However, there are times when we have more information available to us than



just the specification. Sometimes we have an algorithm that can be used to define “better” input conditions than those given by the specification alone, and to choose better equivalence classes.

Let us assume that the input domain for the GetWord function is the set of strings of length 200 characters or less. The output domain is the set of pairs strings of 200 characters or less. Note that, as the input string is a maximum of 200 characters, any of its substrings will be a maximum of 200 characters in length.

Using the guidelines from Section 2.4 in combination with the specification for GetWord, we now have an input condition that the length of the input string must be in the range 0 ... 200 characters. Using guideline 1 we have one valid equivalence class and two invalid classes; the valid class consists of strings of length between 0 ... 200 and an invalid class of length > 200. The other invalid class consists of strings of length < 0 which is infeasible because we cannot supply a string with a negative length. So we have:

$$\begin{aligned} EC_{valid} &= \{s \mid s \text{ is a string and } \text{length}(s) \leq 200\} \\ EC_{invalid} &= \{s \mid s \text{ is a string and } \text{length}(s) > 200\} \end{aligned}$$

But we are not yet done. The class  $EC_{valid}$  can be partitioned into smaller classes — the algorithm in Figure [fig:GetWord-*alg*] specifies a number of possibilities. This is the creative part of testing — thinking about what can go wrong and finding test cases that show that a program *does* go wrong.

Strings can be empty, they consist of one word or multiple words, there can be a single blank between words or a number of blanks between words, and words can contain legal characters or illegal characters. Using this information we can decompose  $EC_{valid}$  into smaller ECs as follows.

$$\begin{aligned} EC_{valid_1} &= \{s \mid s \text{ is empty}\} \\ EC_{valid_2} &= \{s \mid s \text{ has a single word}\} \\ EC_{valid_3} &= \{s \mid s \text{ has more than one word}\} \\ EC_{valid_4} &= \{s \mid \text{words are separated by single blank}\} \\ EC_{valid_5} &= \{s \mid \text{words are separated by 2 or more blanks}\} \end{aligned}$$

Notice that the union of all of the smaller ECs covers our original valid EC, that is,  $\bigcup_i EC_{valid_i} = EC_{valid}$ .

As with the triangle function in Example 14, there is overlap in the equivalence classes. In the GetWord case, strings with blanks must also have more than one word, so  $EC_{valid_3}$  is made redundant by  $EC_{valid_4}$  and  $EC_{valid_5}$ .

The test cases that we obtain from these equivalence classes are given in Table 2.4. The invalid class consists of strings of more than 200 words in length.

Equivalence Class	Input	Expected Output
		(word, line)
$EC_{valid_1}$	“”	(“”, “”)
$EC_{valid_2}$	“word”	(“word”, “”)
$EC_{valid_4}$	” two words”	(“two”, ” words”)
$EC_{valid_5}$	” two words”	(“two”, ” words”)
$EC_{invalid}$	” Story time ...” (200+ chars)	Error value

## 2.7 Test Template Trees

One complication of using equivalence partitioning is detecting and eliminating overlapping equivalence classes. However, taking a more structured approach to this using *test template trees*, we can avoid this problem by not introducing overlaps in the first place.

A test template tree is an overview of an equivalence partitioning, but importantly, it is *hierarchical*. When applied correctly, it provides a graphical overview of equivalence classes and their *justification*, as well as avoiding overlap.

The process for deriving a test template tree for a program is similar to doing equivalence partitioning.

### 2.7.1 Start

We start with the testable input domain, and aim to break this into smaller equivalence classes that make good tests.

### 2.7.2 Repeat

At each step, we choose an equivalence partitioning guideline to apply to one or more *leaf nodes* in test template tree, breaking the leaf nodes into multiple new leaf nodes, each once more specific than its parent node, which is no longer a leaf node.

When we partition each leaf node, we ensure that the partitioning is: (a) disjoint — the partitions do not overlap; and (b) they partitions cover their parent — that is, if we combine the new partitions, that combination will be equivalent to their parent. These two properties are the same as listed in Section *Equivalence Classes* for equivalence classes. By avoiding creating overlap and by covering each parent, we ensure (inductively) that the resulting equivalence classes at the end of this entire process also do not overlap and their cover the root node (the input domain).

### 2.7.3 End

The process ends when either: (a) there are no more sensible partitionings to apply; or (b) our tree grows too large and we can no longer manage the complexity (although there are ways to mitigate this as we will discuss soon).

### 2.7.4 Result

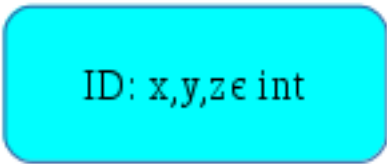
To construct the equivalence classes, we simply take each leaf node and derive the equivalence from that leaf node to the root of the tree.

The resulting equivalence classes do not overlap and they cover the entire input domain. This is because, at each partition, we ensure this property held locally. This local property means that non-overlap and coverage also hold globally, which can be demonstrated inductively.

---

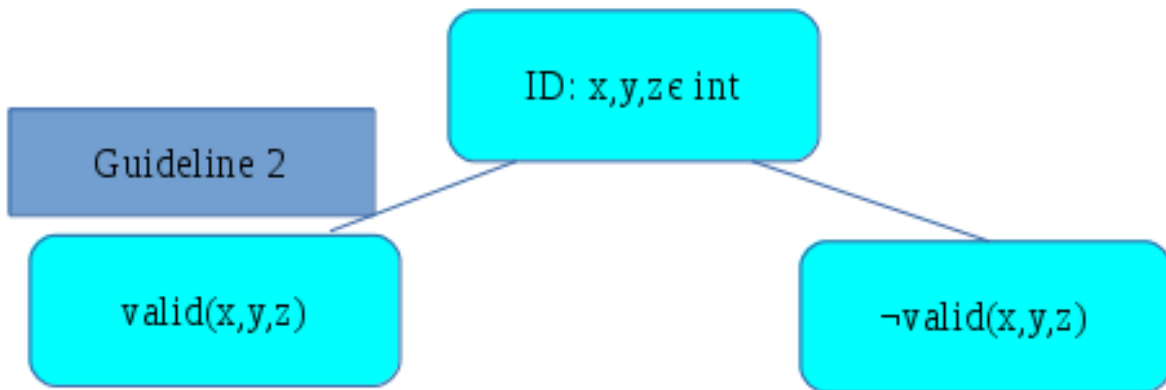
#### Example 16

*As an example, consider the triangle program. At the start, we have the testable input domain, which is just three integers:*

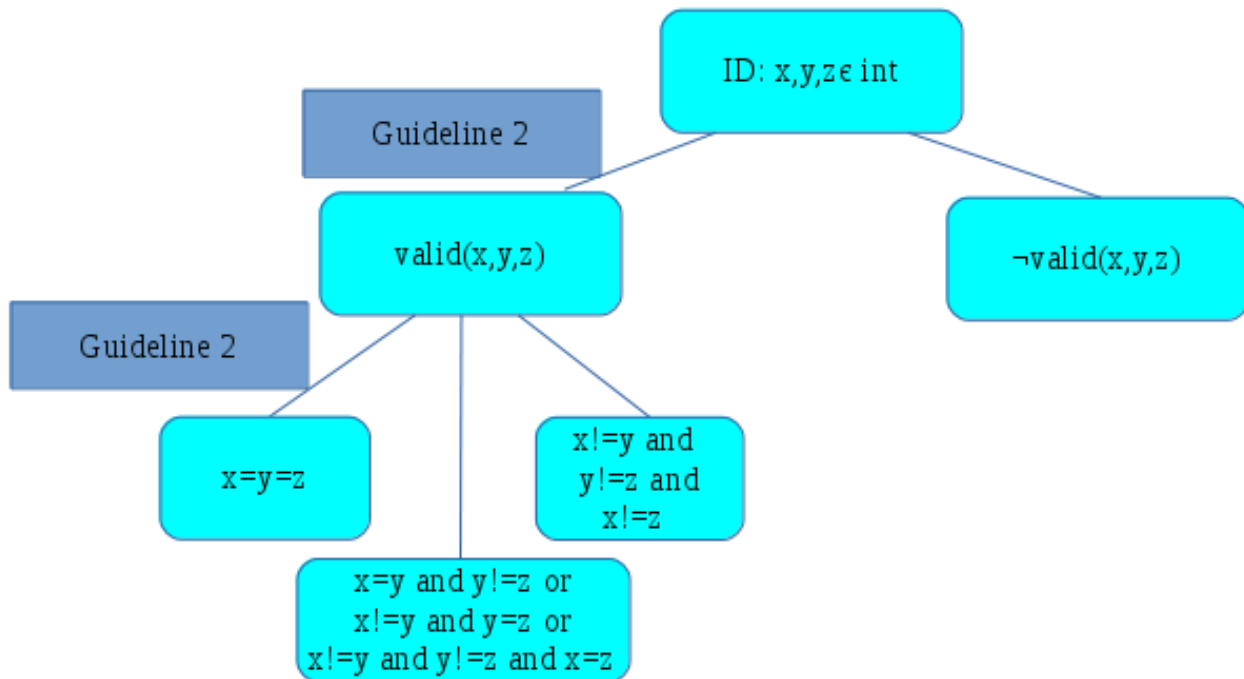


ID:  $x,y,z \in \text{int}$

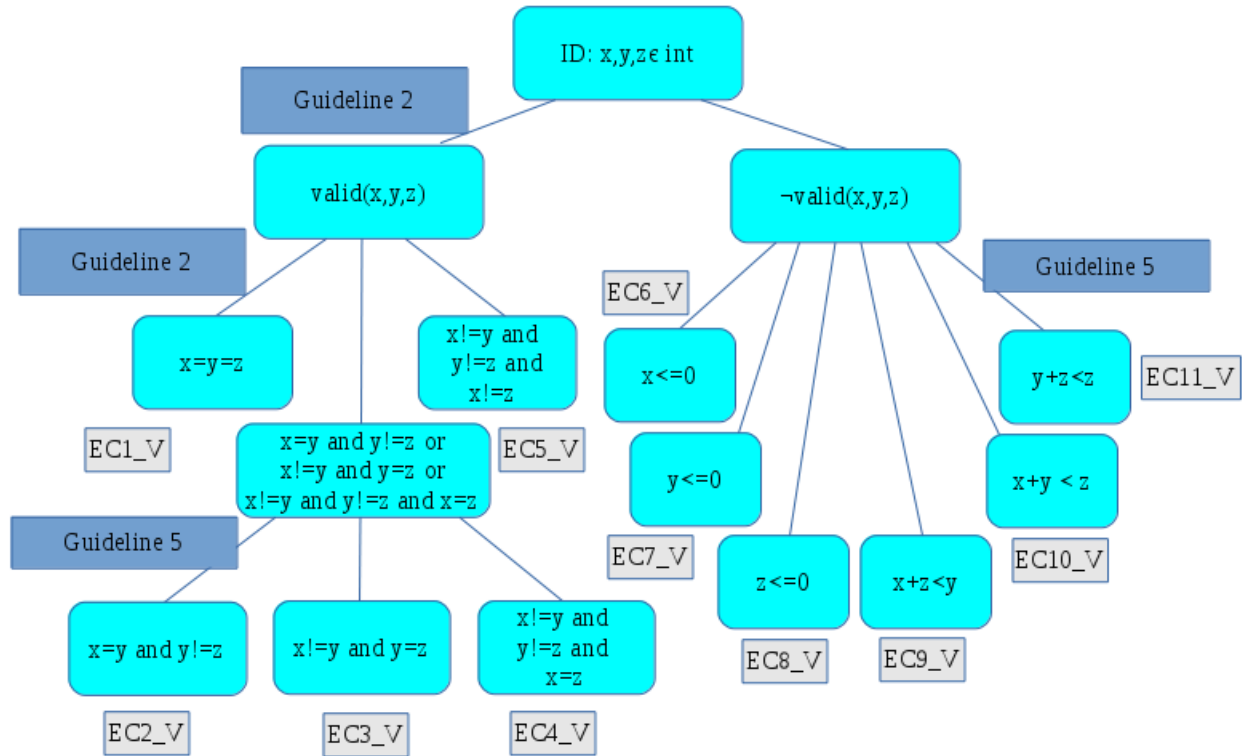
We apply guideline 2 to break this into valid and invalid triangles, giving us two new leaf nodes:



Then, we apply guideline 2 again to the three different types of triangle: equilateral, isosceles, and scalene, breaking the valid triangle node into three new leaf nodes:



Continue the process several more times, we end up with the following test template tree:



Note that the leaf nodes: (a) do not overlap, because we were careful to always break leaf nodes into non-overlapping cases; and (b) cover the entire input space, because we were careful to always ensure that each breakdown covered its parent node.

From here, the 11 equivalence classes fall out directly from the test template tree:

$$\begin{aligned}
 EC1\_V &= \{x, y, z \in \text{int} \mid \text{valid}(x, y, z) \wedge x = y = z\} \\
 EC2\_V &= \{x, y, z \in \text{int} \mid \text{valid}(x, y, z) \wedge x = y \wedge y \neq z\} \\
 EC3\_V &= \{x, y, z \in \text{int} \mid \text{valid}(x, y, z) \wedge x \neq y \wedge y = z\} \\
 EC4\_V &= \{x, y, z \in \text{int} \mid \text{valid}(x, y, z) \wedge x \neq y \wedge y \neq z \wedge x = z\} \\
 EC5\_V &= \{x, y, z \in \text{int} \mid \text{valid}(x, y, z) \wedge x \neq y \wedge y \neq z \wedge x \neq z\} \\
 EC6\_V &= \{x, y, z \in \text{int} \mid \neg \text{valid}(x, y, z) \wedge x \leq 0\} \\
 &\dots \\
 EC11\_V &= \{x, y, z \in \text{int} \mid \neg \text{valid}(x, y, z) \wedge y + z < z\}
 \end{aligned}$$

## 2.7.5 Mitigating Tree Explosion

Generating partitions like this can result in trees that are prohibitively large. Even the tree for the triangle example above starts to become tedious, and this is a tiny program. The problem is caused by the nature of the partitioning: if we apply a guideline to every leaf node of a tree, then the number of nodes grows exponentially with the depth of the tree.

To mitigate the problem of explosion, we can apply a couple of rules of thumb:

1. *Test invalid, exceptional, and error cases only once:* Many test inputs involve invalid or exceptional cases, such as throwing an exception or returning an error codes. For these cases, aim to have only one node in the tree that corresponds to each trigger for these exceptional cases. Applying guidelines to exceptional cases will *probably* not (but not certainly not) provide good test inputs. This is because most people use defensive programming, and therefore errors/exceptions are detected at the start of programs and the error/exception is thrown immediately. As such, the remainder of the program is not executed. So, if a node in a test template tree is designed to throw

an exception due to an ill-formed input, creating child partitions that test several different inputs for ‘normal case’ functionality are unlikely to find any additional faults because the corresponding code will not be executed.

2. *Look at variable interactions:* When applying a guideline, use your experience and intuition and apply only to the current leaf nodes in which this partition is likely to involve some interaction between variables. For example, consider a system for recording loans from a library, with inputs such as date of loan, length of loan, name of book/DVD, author, etc. There is a clear interaction between the date of the loan and the length of the loan, because these two variables determine the due date. However, the name of the book and the author (probably) do not affect the due date. As such, when testing different values of loan length, it would be better to prioritise the nodes that focus on the loan date, rather than on the book name and author. This reduces the number of partitions applied, and therefore mitigates the problem of explosion.

### 2.7.6 Practical use

In reality, software engineers/testers do not tend to capture these test template trees in diagrams. The idea of the test template tree is merely as a way to structure one’s thinking in a divide-and-conquer approach to testing. The trees can be constructed loosely in one’s head to provide a top-down view of testing, or can be sketched when the task is too large to contain mentally.

That said, I have been in projects where these types of models are sketched out on whiteboards for system-level testing to provide an overview of a high-level testing strategy. So, this idea can be used as a communication tool about testing in an industry project in the same way that we are using it in these notes.

## 2.8 Combining Partitions

The partitioning techniques in this section advocate looking at the variables of a program, and using test case selection techniques to derive equivalence classes. However, we quickly run into problems in which we use multiple criteria, or one criteria over multiple variables.

For example, recall in Example 14, in which we derived test cases for the triangle program using equivalence partitioning, that we generated six overlapping equivalence classes for invalid triangles. This was done by using two different criteria over three different variables. Removing the overlapping cases would generate a large number of test cases, many of which would be of little use. For example, three of these cases would be the following:

$$\begin{aligned} \mathbf{x} \leq 0 \wedge y > 0 \wedge z > 0 \wedge x + y > z \\ x > 0 \wedge y > 0 \wedge \mathbf{z} \leq 0 \wedge x + y > z \\ \mathbf{x} \leq 0 \wedge y > 0 \wedge \mathbf{z} \leq 0 \wedge x + y > z \end{aligned}$$

The bold in these represent the values that make the triangle invalid. However, programs statements are executed in sequence, so an implementation of the triangle problem would check for non-positive  $x$  and  $z$ , but would check one before the other. This makes the third case redundant.

Redundancy is not the only issue. If we consider using a partitioning method on three different variables, one with the domain of non-negative integers (variable  $x$ ), one with the domain of non-positive integers (variable  $y$ , and one with the domain of characters  $z$ , we may end up with the following partitions:

- For  $x$ :  $x = 0$ ,  $0 < x < 10$ , and  $x \geq 10$  (three equivalence classes).
- For  $y$ :  $y = 0$  and  $y > 0$  (two equivalence classes).
- For  $z$ :  $z$  is a lower-case letter,  $z$  is an upper-case letter, and  $z$  is a non-letter character (three equivalence classes).

Ammann and Offutt call these *block* combinations, in that, while the equivalence classes overlap, there are distinct blocks that are different to each other in some way. In the above case, the blocks are based on the variables, and there are equivalence classes for each variables.

When we combine the variables, we encounter overlap. The question is how to resolve the overlap so that we do not derive too many test cases. We present three different criteria to solve the problem: the *all combinations* criterion, the *pair-wise combinations* criterion, and the *each choice combinations* criterion.

## 2.8.1 All Combinations

The *all combinations* criterion specifies that every combination of each equivalence class between blocks must be used. This is analogous to the *cross product* of sets. So, if we considering the program with the inputs  $x$ ,  $y$ , and  $z$ , then we would require 18 test cases ( $3 \times 2 \times 3$ ), satisfying the following 18 equivalence classes, consisting of the following 6 equivalence classes:

$$\begin{aligned} x = 0 \quad \wedge \quad y = 0 \quad \wedge \quad z \text{ is lowercase} \\ x = 0 \quad \wedge \quad y = 0 \quad \wedge \quad z \text{ is uppercase} \\ x = 0 \quad \wedge \quad y = 0 \quad \wedge \quad z \text{ is a non-letter} \\ x = 0 \quad \wedge \quad y > 0 \quad \wedge \quad z \text{ is lowercase} \\ x = 0 \quad \wedge \quad y > 0 \quad \wedge \quad z \text{ is uppercase} \\ x = 0 \quad \wedge \quad y > 0 \quad \wedge \quad z \text{ is a non-letter} \end{aligned}$$

and the remaining twelve, which are as above except with  $0 < x < 10$  and  $x \geq 10$ .

The number of test cases is a product of the number of blocks and the number of equivalence classes in each block. In this case, it relates to the number of variables, and the number of equivalence classes for each variable. Therefore, if there was two additional variables, each with three equivalence classes, the number of test cases would be  $3 \times 2 \times 3 \times 3 \times 3 = 162$  test cases. This is likely to be more test cases than necessary, as we saw with the triangle program.

## 2.8.2 Each Choice Combinations

The *each choice* criterion specifies that just one test case must be chosen from each equivalence class. This is the approach we took in the testing of triangle program in Example 14.

Taking the example of the three variable program, the each choice criterion could be satisfied by choosing inputs that satisfy the following three equivalence classes:

$$\begin{aligned} x = 0 \quad \wedge \quad y = 0 \quad \wedge \quad z \text{ is lowercase} \\ 0 < x < 10 \quad \wedge \quad y > 0 \quad \wedge \quad z \text{ is uppercase} \\ x \geq 10 \quad \wedge \quad y = 0 \quad \wedge \quad z \text{ is a non-letter} \end{aligned}$$

This is significantly weaker than the all combinations criterion, and does not consider combinations of values.

## 2.8.3 Pair-Wise Combinations

The *pair-wise combinations* criterion aims to combine values, but not an exhaustive enumeration of all possible combinations. As the name suggests, an equivalence class from each block must be paired with every other equivalence class from all other blocks.

In our above example, this implies that we must have test cases to satisfy the following 21 pair-wise combinations:

$$\begin{array}{llll} x = 0 \wedge y = 0 & 0 < x < 10 \wedge y = 0 & x \geq 10 \wedge y = 0 & y = 0 \wedge z \text{ is lowercase} \\ x = 0 \wedge y > 0 & 0 < x < 10 \wedge y > 0 & x \geq 10 \wedge y > 0 & y = 0 \wedge z \text{ is uppercase} \\ x = 0 \wedge z \text{ is lowercase} & 0 < x < 10 \wedge z \text{ is lowercase} & x \geq 10 \wedge z \text{ is lowercase} & y = 0 \wedge z \text{ is a non-letter} \\ x = 0 \wedge z \text{ is uppercase} & 0 < x < 10 \wedge z \text{ is uppercase} & x \geq 10 \wedge z \text{ is uppercase} & y > 0 \wedge z \text{ is lowercase} \\ x = 0 \wedge z \text{ is a non-letter} & 0 < x < 10 \wedge z \text{ is a non-letter} & x \geq 10 \wedge z \text{ is a non-letter} & y > 0 \wedge z \text{ is uppercase} \\ & & & y > 0 \wedge z \text{ is a non-letter} \end{array}$$

However, a single test input can cover more than one of these. For example, the test input:

$x = 0 \wedge y = 0 \wedge z \text{ is lowercase}$  covers both  $x = 0 \wedge y = 0$  and  $x = 0 \wedge z \text{ is lowercase}$ .

With this in mind, we can select test cases that satisfy only nine equivalence classes; the three classes below:

$$\begin{array}{llll} x = 0 & \wedge & y = 0 & \wedge & z \text{ is lowercase} \\ x = 0 & \wedge & y > 0 & \wedge & z \text{ is uppercase} \\ x = 0 & \wedge & y > 0 & \wedge & z \text{ is a non-letter} \end{array}$$

and the remaining six, which are as above except with  $0 < x < 10$  and  $x \geq 10$ .

One can generalise this to *T-Wise Combinations*, in which we require  $T$  number of combinations instead of pairs. If  $T$  is equal to the number of blocks, then this is equivalent to the all combinations criterion.

## 2.9 References

P. Ammann and J. Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008.

R. Hierons, Avoiding coincidental correctness in boundary value analysis, *ACM Transactions on Software Engineering and Methodology*, 15(3):227–241, 2006.





## BOUNDARY-VALUE ANALYSIS

### 3.1 Learning outcomes of this chapter

At the end of this chapter, you should be able to:

- Explain why boundary-value analysis is valuable compared to standard equivalence partitioning
- Given a set of equivalence classes for a program, identify the boundaries of those equivalence classes, including the on points and off points.
- From a set of equivalence classes, select test inputs that adequately cover the boundaries of a program.

### 3.2 Chapter Introduction

The input partitioning methods presented in Chapter 2 can be improved by trying to find better test cases within equivalence classes. *Boundary-value analysis* is both a refinement of input partitioning and an extension of it. Boundary-value analysis selects test cases to explore conditions on, and around, the edges of equivalence classes obtained by input partitioning.

---

#### Definition

*Boundary conditions* are predicates that apply directly on, above, and beneath the boundaries of input equivalence classes and output equivalence classes.

---

Intuitively, boundary-value analysis aims to select test cases to explore the boundary conditions of a program. Boundary-value analysis and input partitioning are closely related. Both of them exploit the idea that each element in an equivalence class should execute the same paths in a program. However, boundary-value analysis works on the theory that, if a programmer makes a mistake in the logic of the program, such that some inputs in an equivalence class execute the incorrect paths, then those mistakes will often occur at the *boundary* between equivalence classes, because these boundaries are related to flow control constructs such as if statements and while loops in programs.

---

#### Definition

A *computational fault* is a fault that occurs during a computation in a program; for example, an arithmetic computation or a string processing error.

---

Consider the following statement to calculate the average of two numbers:

```
average = (left + right) / 2
```

A computational fault would be:

```
average = left + right / 2
```

In the above, the division operator binds tighter than the addition operator, so the computation is incorrect as it calculates  $\text{average} = \text{left} + (\text{right} / 2)$ .

---

**Definition**

A *boundary shift* is when a predicate in a branch statement is incorrect, effectively ‘shifting’ the boundary away from its intended place.

---

For example, if our program should do something if a list has more than 10 elements, we would have:

```
if (length(list) > 10) then ...  
else ...
```

However, a programmer could make a simple mistake of using  $\geq$  instead of  $>$ :

```
if (length(list)  $\geq$  10) then ...  
else ...
```

Here, the boundary has ‘shifted’ across one value from where it should be *according to the specification*.

Boundary-value analysis attempts to find these faults near the boundary by selecting test cases on and around the boundary. In the example above, after identifying the equivalence classes, we would select *at least* test cases that satisfy the condition cases:  $\text{length}(\text{list}) = 10$  (the top end of the first EC); and  $\text{length}(\text{list}) > 10$  (the bottom end of the second EC).

If we arbitrarily choose values of the equivalence classes, such as a list of length 5 and one of length 15, then our tests would fail to find the above fault.

Collected defect data supports the theory behind boundary-value analysis. Many faults are boundary shifts introduced at boundary conditions because programmers either: (1) are unsure of the correct boundary for an input condition; or (2) have incorrectly tested the boundary.

Many computational faults are also found in defect data, however, it is important to note the tests on the boundaries can detect both computational faults and boundary shifts, while those away from the boundary defect only computational faults. Therefore, test cases that explore the boundary conditions have a more important role than test cases that do not.

Boundary-value analysis requires one *or more* test cases be selected from the edge of the equivalence class or close to the edge of the equivalence class, whereas equivalence partitioning simply requires that any element in the equivalence class will do. Boundary-value analysis also requires that test cases be derived from the output conditions. This is different to equivalence partitioning where only the input domain is usually considered.

## 3.3 Values and Boundaries

Before we discuss how to apply boundary-value analysis, we define some related terms.

---

**Definition**

- A *path condition* is the condition that must be satisfied by the input data for that path to be executed.
- A *domain* is the set of input data satisfying a path condition.
- A *domain boundary* is the boundary of a domain. It typically corresponds to a simple predicate.

Test inputs are chosen to be *on* or around the boundary. However, there are two distinctly different types of boundary: *closed boundaries* and *open boundaries*.

---

**Definition**

- A *closed boundary* is a domain boundary where the points on the boundary belongs to the domain, and is defined using an operator that contains an equality. For example,  $x \leq 10$  and  $y == 0$  are both closed boundaries.
- An *open boundary* is a domain boundary which is not closed, and is defined using a strict inequality. For example,  $y < 10$  is an open boundary, because 10 does not fall within the boundary.
- An *on point* is a point on the boundary of an equivalence. For example, the value 10 is an on point for the boundary  $x \leq 10$ . Counter-intuitively, 10 is also the on point for the boundary  $x < 10$ . Therefore, for a closed boundary, an on point will be a member of the equivalence class, and for an open boundary, it will not.
- An *off point* is a point just off the boundary of an equivalence class. For example, if  $x$  is an integer, the value 11 is the off point for the equivalence class  $x \leq 10$ . If  $x$  is a floating point number, the off point would be 10.001 (assuming that 0.001 is the smallest floating point on the specific architecture).

For the open boundary  $y < 10$ , the off point is 9 if  $y$  is an integer, and 9.999 if  $y$  is a floating point number.

The reverse of the on point then holds: for a closed boundary, the off point will fall outside of the equivalence class, and for an open boundary, it will fall inside the equivalence class.

A special case of off points is strict equalities. That is, boundaries of the form  $x == 10$ . In this case, there are two off points: 9 and 11.

---

## 3.4 Guidelines for Boundary-Value Analysis

Given a set of domain boundaries to test, we can apply the following guidelines to select test cases.

1. If a boundary is a strict equality, for example  $x == 10$  select the on point, 10, and both off points, 9 and 11.
2. If a boundary is an inequality, for example  $x < 10$ , select the on point 10, and the off point 9.
3. For boundaries containing unordered types, that is, types such as Booleans or strings, choose one on point and one off point. An on point is any value that is in the equivalence class, while an off point is any value that is not. For example, if the equivalence class for a string variable is “contains no spaces”, then test one string containing no spaces, and one string containing at least one space.
4. Analyse boundaries of equivalence classes; not individual equivalence classes. If we have the equivalence classes  $x \leq 0$  and  $x > 0$ , then 0 is the on point for both, so analyse that boundary not each class in isolation.

---

**Remark**

It is not always easy to see the boundaries for an equivalence class; boundaries may not exist in such simple forms or make sense. When they do, they give better test cases than equivalence partitioning.

---

---

**Example 23: Boundary-Value Analysis for the Triangle Program**

Consider again the triangle classification program from Example 14. To show the boundary-value analysis method more clearly we will change the specification slightly to input floating point numbers instead of integers.

The program reads floating point values from the standard input. The three values are interpreted as representing the lengths of the sides of a triangle. The program then prints a message to the standard output that states whether the triangle, if it can be formed, is equilateral, isosceles, scalene, or invalid.

Given the equivalence classes identified above using equivalence partitioning we now have the following possible boundary conditions. Let  $x$ ,  $y$  and  $z$  be the three floating point numbers input to the program.

- For an *equilateral* triangle the sides must all be of equal length and we have only one boundary where  $x = y \wedge x = z$  and we can explore this boundary using the following test cases:

(3, 3, 3)	(on point)
(2.999, 3, 3)	(off point below for x)
(3.001, 3, 3)	(off point above for x)

Plus the five additional cases that correspond to  $x = y \neq z$  (one above, one below),  $x \neq z = y$  (one above, one below) and  $x \neq y \neq z$ .

- For an *isosceles* triangle two sides must be equal. This gives the boundary conditions  $x = y \wedge y \neq z$ ,  $y = z \wedge z \neq x$  or  $x = z \wedge z \neq y$ . The boundary  $x = y \wedge y \neq z$  can be explored using the following test cases:

(3, 3, 4)	(on point)
(2.999, 3, 4)	(off point below)
(3.001, 3, 4)	(off point above)

Plus four additional cases for the off points of  $y$  and  $z$ .

- For a *scalene* triangle, all sides must be of a different length. This equivalence class is  $x \neq y \wedge y \neq z \wedge z \neq x$ . This can be explored using the following test cases:

(3, 3, 3)	(on point)
(2.999, 3, 3)	(off point below)
(3.001, 3, 3)	(off point above)

Plus four additional cases for the off points of  $y$  and  $z$ . However, note that all nine of these values have already been tested. The on and off points for this are the same as for the equilateral triangle.

- For invalid triangles, we have the cases in which the sides are of size 0 or below; that is  $x \leq 0$ . The following test cases explore this for  $x$ :

(0, 3, 3)	(on point)
(0.001, 3, 3)	(off point)

and similarly for  $y$  and  $z$ .

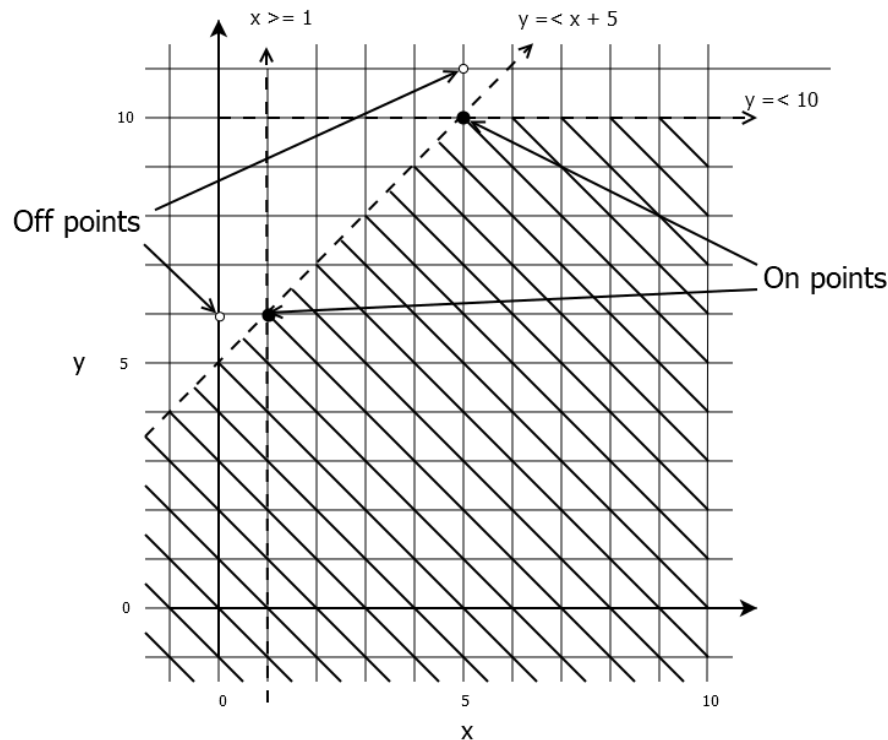
For invalid triangles, we also have the cases in which one of the sides is at least as long as the sum of the other two. The following test cases explore the equivalence class  $x + y \leq z$ :

(1, 2, 3)	(on point)
(1, 2, 2.999)	(off point)

### 3.4.1 Domains with Multiple Variables

So far, we have discussed strategies for choosing on and off points for domains containing only single boundaries. However, it is common for domains to have more than one variable. In fact, the triangle program in Example 23 does have equivalence classes with multiple variables, (for example, the invalid case of  $x + y > z$ ), however, we conveniently did not discuss this.

First, let us consider test cases for two-dimensional linear boundaries. Consider an equivalence class made up of the following linear boundaries:  $x \geq 1$ ,  $y \leq 10$ , and  $y \leq x + 5$ , in which  $x$  and  $y$  are integers. This can be visualised using the two-dimensional graph in Figure 3.1, with the shaded section identifying the equivalence class.



In this case, we have to choose on point values that satisfy all three of the linear boundaries. However, the intersections of the lines lead to fewer test cases, because they test more than one on point in a single test case. This is advantageous because it reduces the number of test cases, and gives us test cases that are more sensitive to boundary-related faults. However, it can also make it a bit more difficult to debug, since it is not clear which boundary a test refers to. In the above, we choose two on points:  $x = 1 \wedge y = 6$ , and  $x = 5 \wedge y = 10$ .

Similarly, the off points near intersections can reduce the numbers of test cases outside the equivalence class. In Figure 3.1, we choose the test inputs  $x = 0 \wedge y = 6$ , and  $x = 5 \wedge y = 11$ . The former is an off point for both  $x \geq 1$  as well as  $y \leq x + 5$ . The latter is an off point for  $y \leq 10$  as well as  $y \leq x + 5$ .

#### Example 24: The Triangle Program — Again!

If we return to the triangle example again, we see that there are equivalence classes based on three variables. Therefore, we need to consider an three-dimensional boundary. This is harder to visualise (and anything above three dimensions even more so!), but selecting the on and off points is still straightforward for this particular program.

If we consider the invalid case in which  $x + y \leq z$  (that is, the length of the side  $z$  is greater than the sum of the other two sides), then we need to select an on and an off point for this. Similar cases exist for when  $x$  and  $y$  are too long. The

inequality,  $x + y < z$ , describes a *plane* in a three-dimensional graph, rather than a line in a two-dimensional graph. To choose test inputs, we must choose three on points on the plane, and one off point.

If we return to the test cases that we derived for these in Example 23, one can see that the test cases selected were on and off points for this plane:

(1, 2, 3)	(on point 1)
(100, 200, 300)	(on point 2)
(400, 2, 402)	(on point 3)
(1, 2, 2.999)	(off point)

The point (1, 2, 3) is on the plane  $x + y = z$ , while the point (1, 2, 2.999) is just below it. However, is this enough to detected a boundary shift?

We also have the following changes compared to the earlier example:

- Equilateral:

(3, 3, 3)	(on point 1)
(100, 100, 100)	(on point 2 – far away from on point 1)
(2.999, 3, 3)	(off point below for x)
(3.001, 3, 3)	(off point above for x)

- Isosceles:

(3, 3, 4)	(on point 1)
(100, 100, 4)	(on point 2 – far away from on point 1)
(2.999, 3, 4)	(off point below)
(3.001, 3, 4)	(off point above)

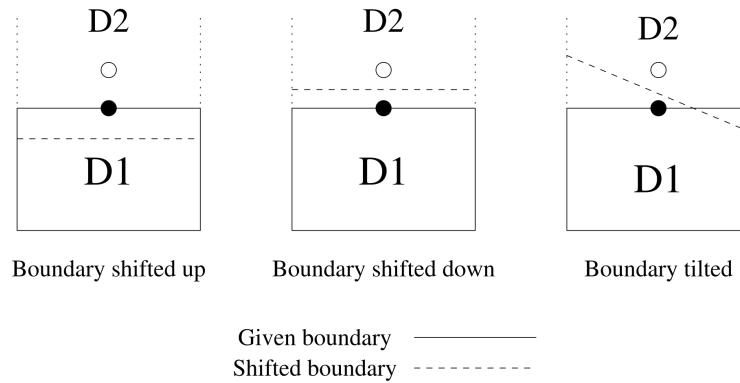
- Scalene:

(3, 3, 3)	(on point 1)
(100, 100, 100)	(on point 2 – far away from on point 1)
(2.999, 3, 3)	(off point below)
(3.001, 3, 3)	(off point above)

---

### 3.4.2 Detecting Boundary Shifts in Inequalities

Figure 3.2 shows a boundary with an on and an off point. The solid line represents the boundary that has been identified (for equivalence partitioning, this is the expected boundary derived from the functional requirements, while for domain testing, this is the actual boundary in the program), while the dashed line represents a shifted boundary (for equivalence partitioning, this is the boundary in the program, while for domain testing, this is the boundary derived from the functional requirements). The black dot represents the on point of the boundary, and the white dot the off point.



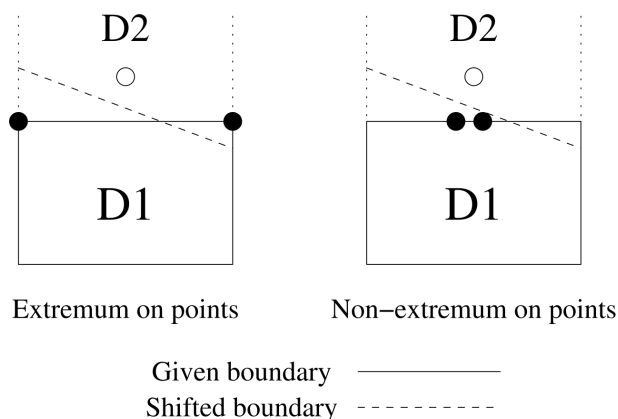
From this figure, one can see that a boundary shift down — in other words, the equivalence class is smaller than expected — will be detected because the on point is identified as being in the wrong domain. However, for the other two, it is possible to have a boundary shift that is not identified.

In the case of a shift up (the equivalence class is larger than expected), the shift has to be great enough that the off point can detect it. For example, if the boundary  $x \leq 10.0$  is identified from the functional requirements, then we would select the on point 10.0, and the off point 10.001. However, if this boundary is incorrectly implemented as  $x \leq 10.0000000001$  (highly unlikely I'm sure, but this is for illustrative purposes only), then the on point 10.0 still falls inside the boundary, and the off point 10.001 still falls outside the boundary. Therefore, these values will not detect the shift. This means that if the off point is a certain distance away from the boundary, then only shifts at least as big as this distance can be detected by that off point. Fortunately, as Beizer points out in his book *Black-Box Testing — Techniques for Functional Testing of Software and Systems*, faults in programs generally occur naturally, and are not the result of sabotage, so choosing an off point that is close to the boundary is almost always likely to detect boundary shifts.

In the case of the tilted boundary, the given boundary and the shifted boundary intersect at some point. In the diagram above, the on point falls within the boundary, and the off point falls outside of the boundary, therefore, these cases will not detect the boundary shift.

Looking at Figure 3.2, it is clear that select one on point and one off point is not sufficient to detect shifts. For the middle case, the best that can be done is to select the off point as close to the boundary as possible, which may be impossible in some domains.

However, we can improve on the case of the tilted boundary in some cases. If the values that can be selected have a minimum and a maximum (the *extremum points*), then we choose both of those as on points. This guarantees that, for any tilted boundary, at least one of the points must lie outside the shift boundary. Figure 3.3 demonstrates this. The position of the tilted boundary does not have an impact on this — as long as some of the boundary falls below and some above, then one of the on points will fall outside the domain. If the two on points are close together, then this reduces the chance of the boundary shift being detected.



In the case that the boundary has no extremums (or only a minimum or maximum but not both) due to it being infinite, then the further away the two on points are from each other, the greater chance there is of producing a failure.

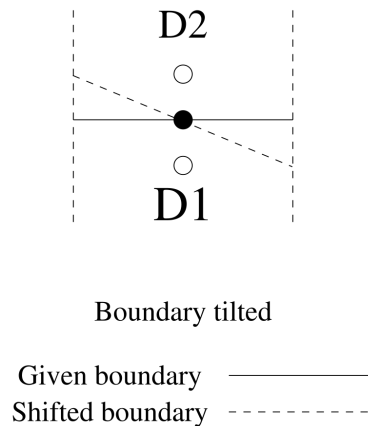
Once the two on points are selected, the best off point to select is one that is as close to the middle as possible between the two on points. This is known as the centroid. Alternatively, one could select more than one off point, but this would likely have little value.

If we return to Figure 3.1, this implies that we are required to add both point on the line  $y = x + 5$ , but it must be at a point where  $y < 10$  to remain inside the boundary.

### 3.4.3 Detecting Boundary Shifts in Equalities

For boundaries defined by equalities, the technique is slightly different. This is because the domain is defined by a line (in two dimensional cases) or a plane (in three dimensional cases), rather than a block. Therefore, if we consider the tilted boundary from Figure 3.2, the on point, represented by the black dot, would not fall inside the boundary, because the entire equivalence class is given by the line, rather than the area below the line.

Recall from Section *Values and Boundaries*, that for equalities involving one variable, we select the on point, and both off points (above and below). For a boundary shift up or down, these are enough to detect the shifts larger than the distance between the on point and off points. However, if the on point chosen happens to intersect with the given and shifted boundary, as in Figure 3.4, then the tilted shift will not be detected.



This is similar to the problem with shifted boundaries for inequalities, and as with that problem, choosing two on points removes this. However, unlike inequality boundaries, equality boundaries are strictly equal, therefore choosing extremum on points is unnecessary. If we consider the boundary tilt in Figure 3.4, then *any* on point other than the point at the intersection will detect the boundary shift. Therefore, if we have *any* two on points, then if the given and shifted boundaries intersect, by definition, at most one of them can be at this intersection. From this, we conclude that we need to choose two arbitrary points.

The same argument goes for non-equalities — that is, defined by predicates such as  $x \neq y + 10$ .

From the above discussion, we conclude the following about selecting boundary values for two-dimensional boundaries:

- For boundaries defined by an inequality, select two on points that are as far apart as possible (or reasonable for infinite domains), and one off point as close to the middle of the two on points as possible;
- For boundaries defined by an equality, select two on points, and two off points (one below and one above); similarly for non-equalities; and
- If the off point is of distance  $d$ , then only boundary shifts of magnitude greater than  $d$  can be detected.

We can generalise this idea to  $N$  dimensional input spaces.



- $N$  on points and 1 off point are needed for boundaries based on inequalities, with the on points as far apart as possible;
- $N$  on points and 2 off points if are needed boundaries based on equalities and non-equalities.
- If the off point test case is a distance  $d$  from the boundary, then only boundary shifts of magnitude greater  $d$  can be detected.

### 3.4.4 Boolean Boundaries

Some equivalence classes just specify a Boolean condition. For example, in input could be a Boolean or the equivalent class could just represent a “must be” situation, such as: the first character of an input must be a numeric character.

In these cases, the boundary is just the true/false value of the class.

---

#### Example – Boolean boundaries

If we recall [Example 13](#) from the *Input Partitioning* chapter, we have two equivalence classes:

1.  $\{s \mid \text{the first character of } s \text{ is a numeric}\}$
2.  $\{s \mid \text{the first character of } s \text{ is not a numeric}\}$

The on point for class 1 is just *true* while the off point is *false*

For class 2, the off point is *true* while the on point is *false*.

In this case, clearly the on point for each class is the off point for the other.

---



## COVERAGE-BASED TESTING

### 4.1 Learning outcomes of this chapter

At the end of this chapter, you should be able to:

- Select test inputs for a program based on its control-flow graph, using the various control-flow coverage criteria.
- Identify when a variable in a program is defined, referenced, and undefined.
- Select test inputs from a control-flow graph annotated with variable information, using the various data-flow coverage criteria.
- Identify static data-flow anomalies from the variable information in a control-flow graph.
- Compare and contrast the different control-flow and data-flow coverage criteria.
- Describe the process of mutation analysis and apply it to a small-scale program.
- Argue why you think mutation analysis is useful or not.
- Define equivalent mutants, and their impact on mutation analysis.
- Describe the importance of and motivate the use of coverage metrics in testing.

### 4.2 Chapter introduction

In this chapter, we discuss three software testing techniques that are based on achieving *coverage* of the software being tested. The notion behind of these techniques is to achieve some form of coverage of the , based on well-defined criteria, rather than partitioning the input domain of the program. While these techniques were originally designed as techniques to *generate* test inputs, in contemporary software engineering, they are more useful as *measures* of test suite quality. That is, we generate a test suite using some technique, such as input partitioning, and we use tools to automatically measure the coverage of the test suite; making improvements when we find parts of the program that should be covered, but are not.

The usual way of looking at the structure of the code is to look at the possible sequences of statements that can be executed. If the program contains an if statement then there are two possible sequence of statements: one in which the if condition evaluates to true; and one in which it evaluates to false. If the program contains a while loop then the actual sequences of statements that get executed will depend on the loop. In some cases it may execute exactly  $n$  times for any execution, in others it may vary, while in some it may execute indefinitely. Thus, the test inputs are selected to execute each statement, branch, or path in the program itself, and are therefore *white-box* testing techniques.

There are three techniques that we discuss in this chapter:

1. **Control-flow testing** – control-flow strategies select test inputs to *exercise* paths in the control-flow graph. They select paths based on the control information in the graph, typically given by predicates in if statements and while loops.

Test inputs are selected to meet criteria for *covering* the graph (and therefore the code) with test cases in various ways. Examples of coverage criteria include:

- Path coverage;
- Branch coverage;
- Condition coverage; and
- Statement coverage.

2. **Data-flow testing** — data-flow strategies select inputs to exercise paths based on the flow of data between variables; for example, between the *definition* of a variable, such as assigning a value to a variable, and the *use* of that variable in the program

As with control-flow strategies, test inputs are selected to cover the graph (and therefore the code) with test cases. Examples of coverage criteria include:

- Execute every definition;
- Execute every use;
- Execute every path between every definition and every use.

3. **Mutation analysis** — mutation analysis is a technique for measuring the effectiveness of test suites, with the side effect that test cases are created and added to that test suite. The technique is based on seeding faults in a program, and then assessing whether or not that fault is detected by the test suite. If not, then that test suite is *inadequate* because the fault was not detected, and a new test case must be added to find that fault.

Using specially designed operators, many copies of the are created, each one with a fault, with the aim of the test suite killing all of these.

---

**Remark**

Note that these techniques are useful only for selecting test *inputs*; not test *cases*. That is, they are not useful for selecting test outputs. If both the test inputs and expected outputs are derived from the program itself, then we will never produce failures because the expected outputs and actual outputs will always be the same. As such, a white-box testing technique still requires a [test\_oracles](test oracle), and therefore, a specification of the program behaviour.

---

## 4.3 Control-Flow Testing

In this section, we discuss *control-flow testing* techniques.

### 4.3.1 Control-Flow Graphs

The most common form of white-box testing is control-flow testing, which aims to understand the *flow of control* within a program. There are a number of ways of capturing the flow of control in a program, but by far the most common is by using a *control-flow graph* (CFG).

Informally, a CFG is a graphical representation of the control structure of a program and the possible *paths* along which the program may execute. More formally we have the following definition.

---

**Definition**

A *control-flow graph* is a graph  $G = (V, E)$  where:

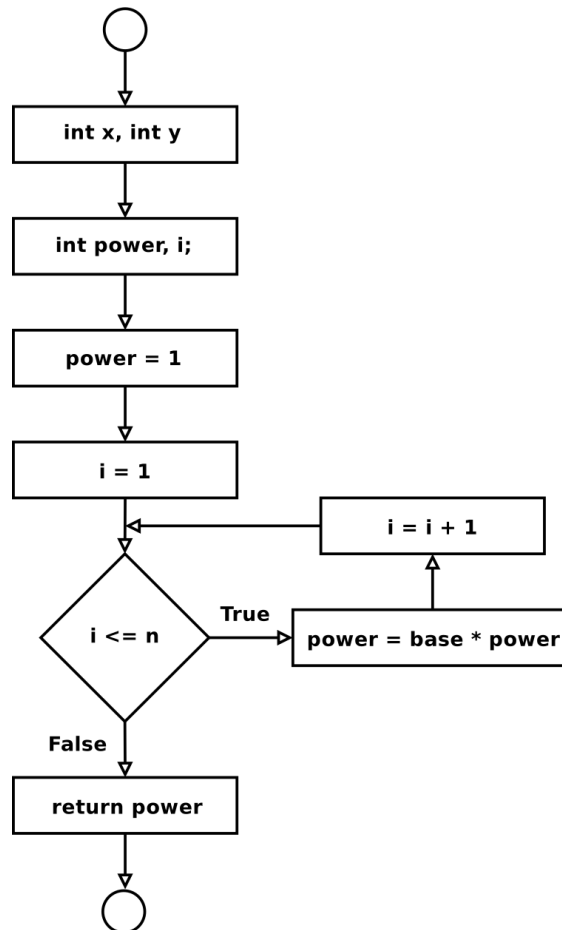
- A vertex in the CFG represents a program statement;

- An edge in the CFG represents the ability of a program to flow from its current statement to the statement at the other end of the edge;
- If an edge is associated with a conditional statement, label the edge with the conditionals value, either true or false.

In diagrams, statements are usually represented by square boxes and branches by diamond boxes.

```
int power(int base, int n)
{
    int power, i;
    power = 1;
    for (i = 1; i <= n; i++) {
        power = base * power;
    }
    return power;
}
```

As a first example consider the power function in Figure 4.1, which for integer inputs base and n, calculates  $base^n$ . All looping constructs decompose into *test-and-branch* control flow structures. The for loop in Figure 4.1 needs to be decomposed into a test  $i \leq n$  and a branch. The control graph for the power function appears in Figure 4.2.



We make two observations about the control-flow graph before continuing.

#### Remark

First, note that we have divided the CFG into a number of *basic blocks*. Each basic block is a sequence of statements with no branches and only a single entry point and a single exit point and so, only a single path through the block. The use of basic blocks makes large CFGs easier to visualise and analyse.

Second, we have labelled all of the nodes in the graph with letters. In general, since paths are sequences of nodes in the control-flow graph we will label nodes and not edges in the graph.

---

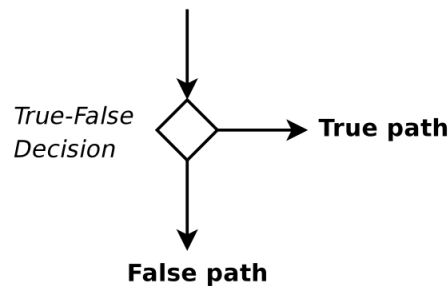
We introduce following definitions for our analysis for control flow graphs.

---

**Definition**

- An *execution path*, or just a *path*, is a sequence of nodes in the control flow graph that starts at the entry node and ends at the exit node.
  - A *branch*, or *decision*, is a point in the program where the flow of control can diverge. For example, if-then-else statements and switch statements cause branches in the control flow graph.
  - A *condition* is a simple *atomic* predicate or simple relational expression occurring within a branch. Conditions *do not* contain *and* (in C &&), *or* (in C ||) and *not* (in C !) operators.
  - A *feasible path* is a path where there is at least one input in the input domain that can force the program to execute the path. Otherwise the path is an *infeasible path* and no test case can force the program to execute that path.
- 

In terms of the control-flow graph, a branch is a node in the graph with two or more edges that leave that node:



In turn branches are made up from conditions. For example, the branch given by `if (a > 1 and b == 0)` consists of the conjunction of two conditions: (1) `a > 1`; and (2) `b == 0`. Analysing the branches and conditions in a program gives us a great deal of insight into how to choose test cases to follow specific paths. For example, we need to select test inputs to make a branch true and false and in turn this means choosing values for `a` and `b` to make the branch take on the values true and false.

In this example, there is one way to make the branch true:

- `a > 1 and b == 0`

and three ways to make the branch false:

- `a > 1 and b != 0`
- `a <= 1 and b == 0`
- `a <= 1 and b != 0`.

### 4.3.2 Coverage-Based Criteria

The aim of coverage-based testing methods is to *cover* the program with test cases that satisfy some fixed coverage criteria. Put another way, we choose test cases to exercise as much of the program as possible according to some criteria. If some part of the program is not exercised by any test case then there may well be undiscovered faults lurking there.

Coverage-based testing for control-flow graphs works by choosing test cases according to well defined *coverage* criteria. The more common coverage criteria are the following.

- **Statement coverage** (or **node coverage**): Every statement of the program should be exercised at least once.
- **Branch coverage** (or **decision coverage**): Every possible alternative in a branch (or decision) of the program should be exercised at least once. For if statements this means that the branch must be made to take on the values true and false, *even if the result of this branch does nothing*, for example, for an if statement with no else, the false case must still be executed.
- **Condition coverage**: Each condition in a branch is made to evaluate to true and false at least once. For example, in the branch `if (a and b)`, where `a` and `b` are both conditions, then we must execute a set of tests such as:
  - `a` evaluates to true at least once and false at least once; and
  - `b` evaluates to true at least once and false at least once.
- **Decision/Condition coverage**: Each condition in a branch is made to evaluate to both true and false and each branch is made to evaluate to both true and false. That is, a combination of branch coverage and condition coverage. For example, in the branch `if a and b`, where `a` and `b` are both conditions, we must execute a set of tests such that:
  - `a` evaluates to true at least once and false at least once;
  - `b` evaluates to true at least once and false at least once; and
  - `a and b` evaluates to true at least once and false at least once.

If `a` and `b` are both Boolean variables, this can be achieved using the following test inputs:

a	b	a and b
true	true	true
false	false	false

- **Multiple-condition coverage**: All possible combinations of condition outcomes within each branch should be exercised at least once. For example, in the branch `if (a and b)`, where `a` and `b` are both conditions, we must execute a set of tests such that all four combinations of `a` and `b` are true and false respectively. That is, we must execute the following four conditions:

a	b	a and b
true	true	true
false	true	false
true	false	false
false	false	false

For this, we have  $2^n$  number of objectives to satisfy, where  $n$  is the number of conditions.

- **Path coverage**: Every execution *path* of the program should be exercised at least once.

**Note** that path coverage is impossible for graphs containing loops, because there are an infinite number of paths. A typical work around is to apply the *zero-to-many* rule, discussed in the [Domain Testing](#) section, which states that the minimum number of times a loop can execute is zero, so this is a boundary condition. Similarly, some loops have an upper bound, so treat this as a boundary.

A general guideline for loops is to select test inputs that execute the loop:

- zero times – so that we can test paths that do not execute the loop;
- once – to test that the loop can be entered and that the results for a single iteration are “correct”;
- twice – to test that the results remain “correct” between different iterations of the loop;
- N times (greater than 2) – to test that an arbitrary number of iterations returns the “correct” results; and
- N+1 times to test that after an arbitrary number of iterations the results remain “correct” between iterations.

In the case that we know the upper bound of the loop, then set N+1 to be this upper bound.

---

**Example 28:**

To motivate the selection of test cases consider the following simple program in Figure 4.3.

```
void main(void)
{
    int a, b, c;
    scanf("%d %d %d", a, b, c);

    if ((a > 1) && (b == 0)) {
        c = c / a;
    }

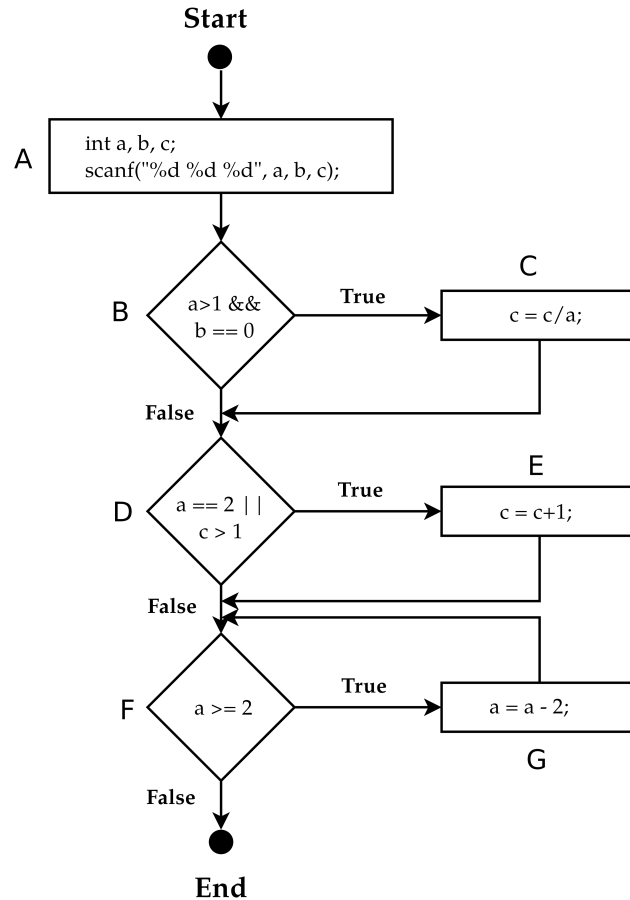
    if ((a == 2) || (c > 1)) {
        c = c + 1;
    }

    while (a >= 2) {
        a = a - 2;
    }

    printf("%d %d %d", a, b, c);
}
```

The first step in the analysis is to generate the control-flow graph which we show in Figure 4.4.





Now, what is needed for statement coverage? If all of the branches are true at least once then we will have executed every statement in the graph. Put another way to execute every statement at least once we must execute the path ABCDEFGF.

Now, looking at the *conditions* inside each of the three *branches* we can derive a set of constraints on the values of *a*, *b* and *c* such that  $a > 1$  and  $b == 0$  in order to make the first branch true,  $a \leq 2$  in order to make the third branch true, and  $c > 1$  to make the second branch true. A test case of the form  $(a, b, c) = (2, 0, 3)$  will execute all of the statements in the program.

Note that we have not needed to make every branch take on both values; nor have we made every condition evaluate to true and false; and nor have we traversed every path in the program.

We already have a test input  $(2, 0, 3)$  that will make all of the branches true. To meet the branch coverage condition all we need are test inputs that will make each branch false. Again looking at the conditions inside the branches, the test input  $(1, 1, 1)$  will make all of the branches false. So our two test inputs  $(2, 0, 3)$  and  $(1, 1, 1)$  is a test suite that will meet the branch coverage criteria.

For any of the criteria involving condition coverage we need to look at each of the five conditions in the program:

- $C_1$ :  $a > 1$
- $C_2$ :  $b == 0$
- $C_3$ :  $a == 2$
- $C_4$ :  $c > 1$
- $C_5$ :  $a \geq 2$ .

The test input  $(1, 0, 3)$  will make  $C_1$  false,  $C_2$  true,  $C_3$  false,  $C_4$  true and  $C_5$  false. Examples of sets of test inputs and the criteria that they meet are given in Table 4.1.

Coverage Criteria	Test Inputs	Path Execution
Statement	(2, 0, 3)	ABCDEFGFGF
Branch	(2, 0, 3)	ABCDEFGFGF
	(1, 1, 1)	ABDF
Condition	(1, 0, 3)	ABDEF
	(2, 1, 1)	ABDFGF
Decision/Condition	(2, 0, 4)	ABCDEFGFGF
	(1, 1, 1)	ABDF
Multiple condition	(2, 0, 4)	ABCDEFGFGF
	(2, 1, 1)	ABDFGF
	(1, 0, 2)	ABDEF
	(1, 1, 1)	ABDF
Path	(2, 0, 4)	ABCDEFGFGF
	(2, 1, 1)	ABDFGF
	(1, 0, 2)	ABDEF
	(4, 0, 0)	ABCDGFGFGF
	...	...

The set of test cases meeting the multiple condition criteria is given in Table 4.2. In the table, we define the branches using the following labels:

- $B_1$ :  $C_1 \ \&\& \ C_2$
- $B_2$ :  $C_3 \ || \ C_4$

Test inputs	$C_1$	$C_2$	$B_1$	$C_3$	$C_4$	$B_2$	$C_5$
	$a > 1$	$b == 0$		$a == 2$	$c > 1$		$a >= 2$
(1, 0, 3)	F	T	F	F	T	T	F
(2, 1, 1)	T	F	F	T	F	T	T
(2, 0, 4)	T	T	T	T	T	T	T
(1, 1, 1)	F	F	F	F	F	F	F

### Exercise 1

- Can you find an *infeasible path* in this example?

**Reminder:** A path is said to be *infeasible* if there are no test cases in the input domain that can execute the path.

- How many execution paths are there in the program?

### Example 29: A Second Example

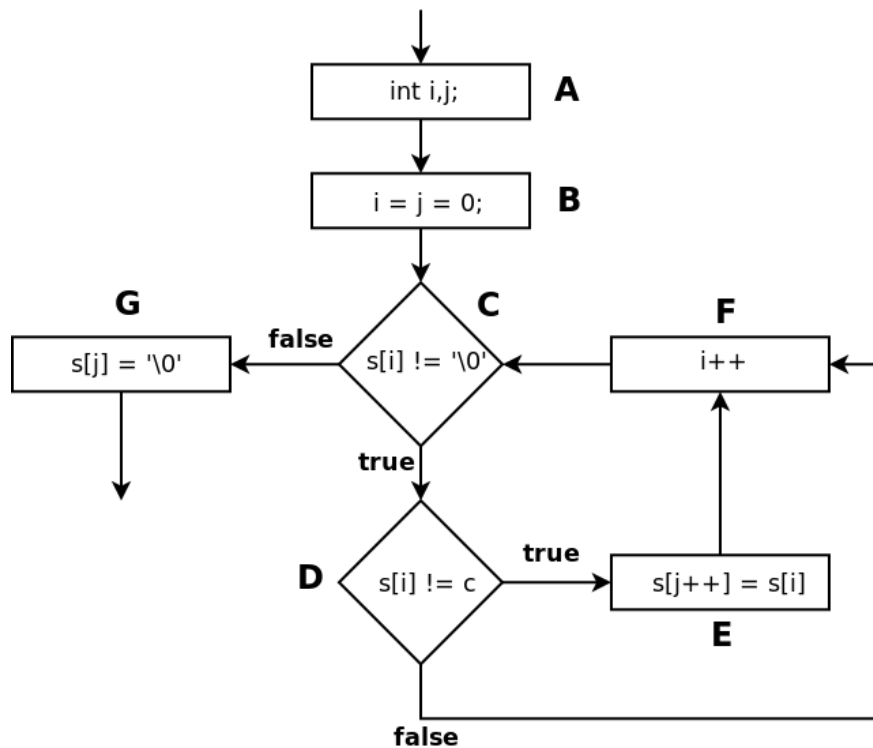
For the second example, consider the function *squeeze* from Section *Programs* (shown again in Figure 4.5).

```

void squeeze(char s[], int c)
{
    int i,j;
    for (i = j = 0; s[i] != '\0'; i++) {
        if (s[i] != c) {
            s[j++] = s[i];
        }
    }
    s[j] = '\0';
}

```

Our analysis begins with the standard analysis of input/output domains and then constructs a control-flow graph for the function. The input domain for the function `squeeze` is  $\text{char}[] \times \text{int}$  and the output domain is  $\text{char}[]$ . The control-flow graph is shown in Figure 4.6.



The next step is to decide which of the coverage criteria will be used to select test cases. Typically, the exact criteria (statement, branch, condition, multiple condition, path) will depend on the project and what testing is meant to achieve.

For this example we will begin by determining the branches and conditions. Fortunately there are only two branches each of which contains only a single condition. Consequently, if we choose test inputs so that each condition evaluates to both true and false then we will also have achieved branch coverage.

Notice that the paths divide the input domain into subsets where each element of a subsets selects a specific path. For the `squeeze` functions the subsets can be characterised as follows.

Path	Equivalence classes to select the path
<i>ABCG</i>	Selected by the test input (" <i>c</i> ", <i>c</i> ) for any character <i>c</i> .
<i>ABCDFCG</i>	Selected by any test input ( <i>S</i> , <i>c</i> ) where <i>S</i> is a string of length 1, and <i>c</i> occurs in <i>S</i> .
<i>ABCDEF CG</i>	Selected by any test input ( <i>S</i> , <i>c</i> ) where <i>S</i> is a string of length 1 and <i>c</i> does not occur in <i>S</i> .
⋮	
<i>ABCDFCDF ... CDFCG</i>	Selected by any test input ( <i>S</i> , <i>c</i> ) where <i>S</i> is a string of length > 1 and <i>c</i> does not occur in <i>S</i> .
<i>ABCDEF CDEF ... CDEF CG</i>	Selected by any test input ( <i>S</i> , <i>c</i> ) where <i>S</i> is a string of length > 1 and <i>c</i> occurs in <i>S</i> .

Next we need to derive the test cases. To do this there are two further questions that we need to answer.

- How many times do we need to execute the for loop?
- How do we determine the expected results for a test case?

If path coverage were to be demanded, then the coverage criteria could not be satisfied because there are an infinite number of paths. However, we can apply the guideline regarding loops and select test inputs that execute the loop 0, 1, 2, *N*, and *N*+1 times. In the squeeze example, the control-flow graph has two loops. Both start and end at node C, however, only one of them contains the node E. Therefore, the zero-to-many rule is applied to each of these. This gives us the paths *ABC(DFC)<sup>n</sup>G* and *ABC(DFEC)<sup>n</sup>G*, for  $n \in \{0, 1, 2, 4, 5\}$ .

This example shows the weakness with the zero-to-many rule (and in path coverage with loops). The paths generated for this only execute the case in which the character *c* is in every position of the array, and the case in which it is in none. Instead of following this strictly, we use our intuition to combine the two cases, and interleave the loop paths; therefore, in some iterations the test case executes the statement at node E, and in others, it does not.

Now that the test inputs have been derived, the expected outputs must be determined. In our case we will use the design-level specification for the squeeze function to manually determine the expected outputs.

Table 4.3 outlines the test cases, and the paths that they exercise, for the squeeze program.

Coverage Criteria	Test Input	Expected Output	Path
Statement Coverage	("c", 'c')	" "	ABCDEF CG
Branch Coverage	("", 'c')	" "	ABCG
	("ab", 'b')	"a"	ABCDFCDEF CG
Path Coverage	("", 'c')	" "	ABCG
	("c", 'c')	" "	ABCDEF CG
	("ab", 'b')	"a"	ABCDFCDEF CG
	("abcd", 'd')	"abc"	ABC(DFC) <sup>3</sup> DEF CG
	("abcde", 'e')	"abcd"	ABC(DFC) <sup>4</sup> DEF CG

### 4.3.3 Measuring Coverage

Clearly, applying control-flow testing manually is a time-consuming process that is unlikely to give many benefits. However, there are aspects of the process that are easily automatable due to their mechanical nature. Deriving the control-flow graph can be automated by simply looking at the source code. Deriving test inputs from is considerably more difficult, especially for programs with non-linear domains, and for non-primitive data types.

However, one straightforward and valuable task is to measure the level of coverage of a given test suite, which can be automated easily with tools. This can be measured by executing the test suite over the program, seeing which coverage objectives are met, and providing both high-level and fine-grained feedback on the coverage.

A *coverage score* is defined as the number of test objects met divided by the number of total test objectives. The test objectives measured are relative to the particular criterion. For example, for statement coverage, the number of objectives is the number of statements, and the number of objectives met is the number of statements executed by *at least one test*. For branch coverage, the number of objects is the number of branches  $\times$  two (each branch must be executed for both the true and false case).

As an example, consider the squeeze function from Figure 4.6. If we want to achieve branch coverage, there are four test objectives: two branches, each with two possible outputs. Now, consider the following test suite, consisting of two tests, which could be generated using any arbitrary technique:  $(s = \text{"abc"}, c = \text{"d"})$ ,  $(s = \text{""}, c = \text{"d"})$ .

We can measure the coverage of this as follows, in which e.g. the FFT in the first row indicates that branch  $C$  was executed *true* on the first three iterations of the loop, and *false* on the final iteration:

Inputs	$s[i] \neq '\backslash 0'$	$s[i] \neq c$
$(s = \text{"abc"}, c = \text{"d"})$	TTTF	FFF
$(s = \text{""}, c = \text{"d"})$	TTTF	FFF

Scanning down the columns, we can see that the branch  $s[i] \neq '\backslash 0'$  is executed for both true and false at least once, while the branch  $s[i] \neq c$  is executed only for the false case. The coverage score is calculated as:

$$\begin{aligned}
 & \frac{\text{objectives met}}{\text{total objectives}} \\
 &= \frac{3}{4} \\
 &= 75\%
 \end{aligned}$$

This means that we have achieved 75% branch coverage. Further, looking at the table, we can scan down the columns to see *which* coverage objectives were not met (the true case for  $s[i] \neq c$ ), and can add a new test input to cover this case.

Thus, from a practical perspective, the idea is to generate a test suite using some method (generally a black-box method), and to then measure the coverage of that test suite. Test objectives that are not covered can then be added using the technique described in this section. Such an approach is cheaper than deriving tests from the control-flow graph directly, due to the fact that a reasonable set of black-box tests will generally achieve a high coverage initially.

### Tool Support

There are many control-flow coverage tools available for most languages, although many of these are restricted only the statement coverage, and sometimes branch or decision/condition coverage. Some open-source tools for Java measuring Java code coverage can be found at <http://java-source.net/open-source/code-coverage>.

One excellent piece of software support is Atlassian's Clover tool (<https://www.atlassian.com/software/clover/overview>). Clover provides feedback on statement (node), branch, and method coverage, and reports fine-grained details such as which tests cover which parts of the code.

## 4.4 Data-Flow Testing

It is hard to do better to motivate data-flow testing than that given by Rapps and Wuyeker:

*It is our belief that, just as one would not feel confident about a > program without executing every statement in it as part of some test, > one should not feel confident about a program without having seen the > effect of using the value produced by each and every computation.*

Despite the analyses of input and output domains used in equivalence partitioning, boundary-value analysis, and control-flow graphs, testing still relies on personal experience to choose test cases that will uncover program faults. The problem is made harder if a program has a large number of input variables. For example, if a program with 5 input variables and each variable's input domain is partitioned into 5 equivalence classes then there are  $5^5 = 3125$  possible combinations to test.

*How can we select better test cases in order to increase the power of each test case?*

One way of improving our test cases is to find better criteria for test case selection. The other way is to find a testing scheme that produces additional information (information other than the output of the program under test) to use for program analysis.

Data-flow analysis provides information about the creation and use of data definitions in a program. The information generated by data-flow analysis can be used to:

- detect many simple programming or logical faults in the program;
- provide testers with dependencies between the definition and use of variables;
- provide a set of criteria to complement coverage based testing.

Static analysis of programs is a large field. We will go into enough depth in these notes to give you the general idea, some techniques that you can use in practice and hopefully to extend later for specific applications.

### 4.4.1 Static Data-Flow Analysis

In the simplest form of data-flow analysis, a programming language statement may act on a variable in 3 different ways. It may *define* a variable, *reference* a variable and *undefine* a variable:

- **Define (d)** – A statement *defines* a variable by assigning a value to the variable. For example if the program variable  $x$  has been declared then the statements  $x = 5$  and `scan(x)` in C both defined the variable  $x$ , but the statement  $x = 3 * y$  only defines  $x$  if  $y$  is defined.

---

#### Remark

Note that this is not the same as a variable being *declared*. In many languages, we can declare a variable without assigning a value to it; e.g. `int x` in C declares  $x$  as an integer but does not give it a value. Similarly, in many languages we can declare a value simply by defining it; e.g.  $x = 5$  in Python will define the value of  $x$ .

---

- **Reference (r)** – A statement makes a *reference* to a variable by reading a value from the variable.

If the program variable  $x$  is defined then an example of  $x$  used as a reference is the statement  $x = 5$  or `scan(x)` in C, where  $x$  is used to store a value and must be referenced to obtain its current value.

- **Undefine (u)** – A statement *undefines* a variable whenever the value of the variable becomes unknown. For example, the scope of a local variable ends.

The aim of data-flow analysis is to trace through the program's control-flow graph and detect *data-flow anomalies*. data-flow anomalies indicate the possibility of program faults.

**Definition – u-r anomaly**

A *u-r anomaly* occurs when an undefined variable is referenced. Most commonly *u-r* anomalies occur when a variable is referenced without it having been assigned a value first — that is, it is *uninitialised*. A common source of *u-r* anomalies arises when the wrong variable is referenced.

Formally, a *u-r* anomaly is when a variable becomes undefined (perhaps has not been defined at all) and it is referenced on *some* future definition-free path. Note that this only requires it to be referenced on at least one path. Even if the variable is defined before it is referenced, or never referenced at all, on other paths, *any* reference on a definition-free path is a potential fault, so this is flagged as an anomaly.

**Definition – d-u anomaly**

A *d-u anomaly* occurs when a defined variable has not been referenced before it becomes undefined. This anomaly usually indicates that the wrong variable has been defined or undefined.

Formally, a *d-u* anomaly is when a defined variable is not referenced on any future path before it is undefined. Note if there is at least one path that references the variable, there is **no** *d-u* anomaly.

**Definition – d-d anomaly**

A *d-d anomaly* indicates that the same variable is defined twice causing a hole in the scope of the first definition of the variable. This anomaly usually occurs because of misspelling or because variables have been imported from another module.

Formally, a *d-d* anomaly is when a defined variable is not referenced on any future path before it is re-defined. Note that if there is at least one path that references the variable, there is **no** *d-d* anomaly.

**Example 30: Static Data-Flow Anomalies**

```
/* Defining p1 and p2 */
TestClass1 p1 = new TestClass1();
TestClass2 p2 = new TestClass2();

/* References to p1 and p2 */
p1->process();
p2->process();

/* Reference to p1 */
p1->process2();

/* p2 is now undefined */
delete p2;

/* Reference to p2 which is now undefined */
p2->process2();
```

As a first example consider the program fragment in Figure 4.7, which is written in C++. Perhaps the most obvious indication of a fault in the program is a *u-r* anomaly where we reference the variable `p2` which is undefined at the point where its value is needed. However, in large systems this kind of anomaly can be difficult to detect manually because of branching in the control-flow graph.

The block of program between the definition of `p2` and the deletion of `p2` is not harmful. However, the anomaly can indicate the premature deletion of `p2` which may effect the execution of the program in the region after the deletion of `p2`.

```
int Faulty_Fibonacci(int n)
{
    int i, j, sum;
    int *c;

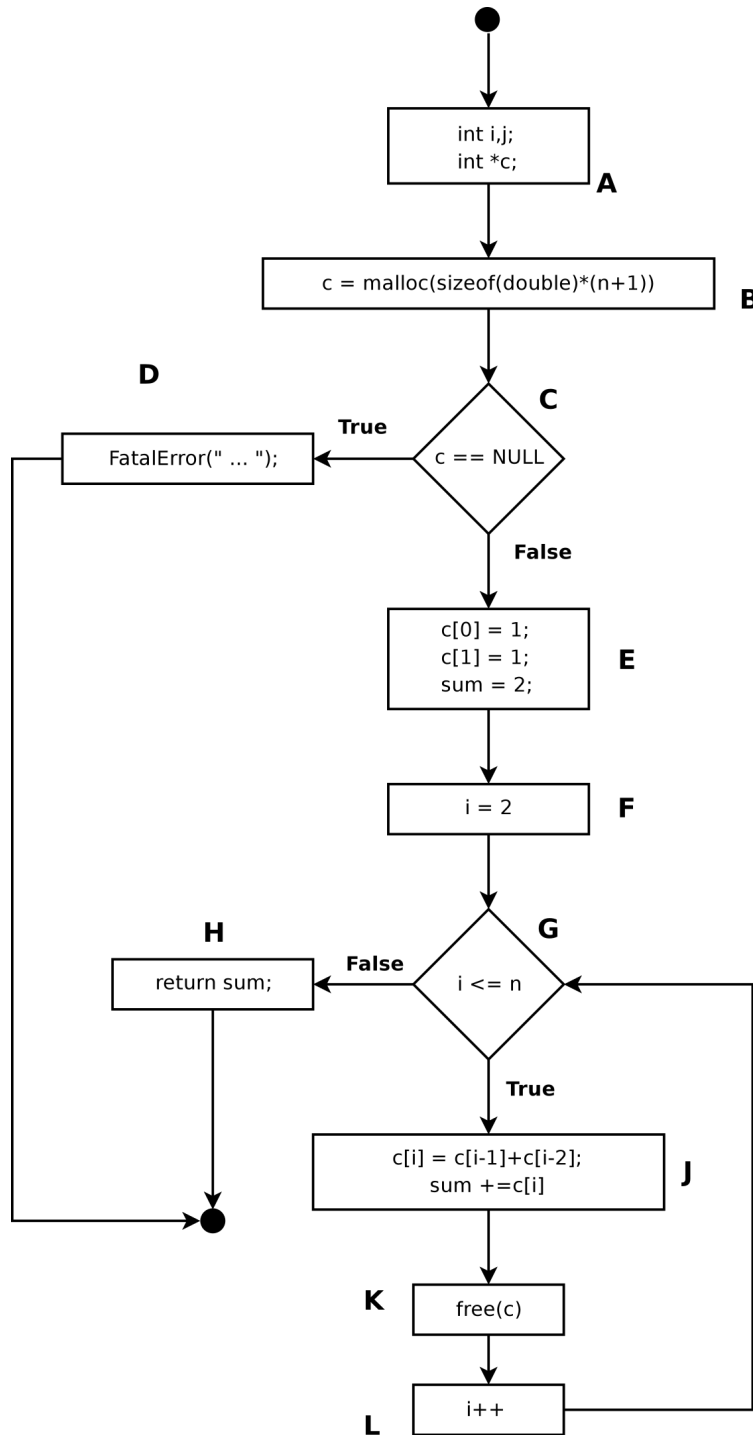
    c = malloc(sizeof(double) * (n + 1));
    if(c == NULL) {
        FatalError(OUT_OF_SPACE);
    }

    c[0] = 1;
    c[1] = 1;
    sum = 2;

    for(i = 2; i <= n; i++) {
        c[i] = c[i - 1] + c[i - 2];
        sum += c[i];
        free(c);
    }
    return sum;
}
```

The second example of a *u-r* anomaly is the function `Faulty_Fibonacci` given in Figure 4.8. The control-flow graph for `Faulty_Fibonacci` is given in Figure 4.9.





The fault in the Faulty\_Fibonacci program is that we have freed up our memory, the array `C` in this case, too early. *Our data-flow analysis will only pick this up if the loop is executed two or more times.*

If we look firstly at the path `ABCEFGJKL` then the variable `c` undergoes the following sequence of transitions at each of the nodes on the path above.

Node	Action on the Variable c
A	no action
B	define (d)
C	reference (r)
E	define (d)
	define (d)
F	no action
G	no action
J	reference (r)
	reference (r)
	define (d)
K	undefine (u)
L	no action

If we execute the loop again we must now revisit nodes G and J. The next action to take on node J is a reference (r) action and so we have a u-r anomaly from the first iteration of loop. This kind of scenario is quite typical of u-r anomalies in C.

As our third example consider the following program fragment:

```
read (m);  
read (n);  
m = n + n;
```

In this example the program variable m is given a value by the read statement (a definition) and then the assignment (another definition), but is not referenced in between.

The *d-d* anomaly can indicate either: (1) that we have mistakenly used m to store the results of the computation n+n, or (2) we have read the wrong variable m as input. The data-flow analysis does not tell us what the fault actually is; it just tells us that there is potentially something wrong.

More generally what types of faults can data-flow analysis detect? Typically we can detect common types of programming mistakes, such as:

- typing errors
  - uninitialised variables
  - misspelling of names
  - misplacing of statements
  - incorrect parameters
  - incorrect pointer references
- 

## 4.4.2 Dynamic Data-Flow Analysis with Testing

We will update our terminology to be consistent with that of Rapps and Wuyeker. The first change to note is that Rapps and Wuyeker distinguish between two different uses of a variable:

---

### Definition

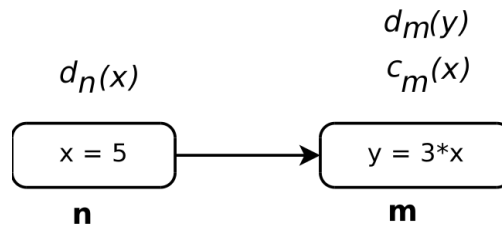
- A **C-use** of a variable is a *computation use* of a variable, for example,  $y = x * 2$ ;
- A **P-use** of a variable is a *predicate use*, for example,  $if (x < 2) \dots$

Secondly, we adopt the notation employed by Rapps and Wuyeker, as outlined in Definition 33 below.

### Definition

- Let  $d_n(x)$  denote a variable  $x$  that is assigned or initialised to a value at node (statement)  $n$  (**Definition**).
- Let  $u_n(x)$  denote a variable  $x$  that is used, or referenced, at node (statement)  $n$  (**Use**).
- Let  $c_n(x)$  denote a computational usage of the variable  $x$  at the node  $n$  (**Computational Use**).
- Let  $p_n(x)$  denote a predicate usage of the variable  $x$  at the node  $n$  (**Predicate Use**).
- Let  $k_n(x)$  denote a variable  $x$  that is killed, or undefined, at a node (statement)  $n$  (**Kill**).

The data-flow annotations of Definition 33 are attached to nodes in the control-flow graph. The node **n** in the figure below



is annotated with a definition for  $x$ . This means that the variable  $x$  at node  $n$  is defined from the node  $n$  onwards along any path containing **n**. The node **m** is annotated with a *computational* use of  $x$ , and a definition of  $y$ .

All of the defines, uses, and undefines of a single variable are collated, and are represented using a *data-flow graph*. A data-flow graph is simply control-flow graph, except that its nodes are annotated with information regarding the definition, use, and undefinition of all of the variables used in that node.

### Exercise

Annotate the graph in Figure 4.8 with its definition, use, and undefinition information.

We define the following for data-flow graphs.

### Definition

- A *definition clear path*  $p$  with respect to a variable  $x$  is a sub-path of the control-flow graph where  $x$  is defined in the first node of the path  $p$ , and is not defined or killed in any of the remaining nodes in  $p$ .
- A *loop-free path segment* is a sub-path  $p$  of the control-flow graph in which each node is visited at most once.
- A definition  $d_m(x)$  reaches a use  $u_n(x)$  if and only if there is a sub-path  $p$  that is definition clear (with respect to  $x$ , and for which  $m$  is the head element, and  $n$  is the final element).

The aim is to define criteria with which to select and assess test suites. We will only look at some of the more popular of the full set of criteria discussed in the literature, and give some feel for their relative effectiveness.

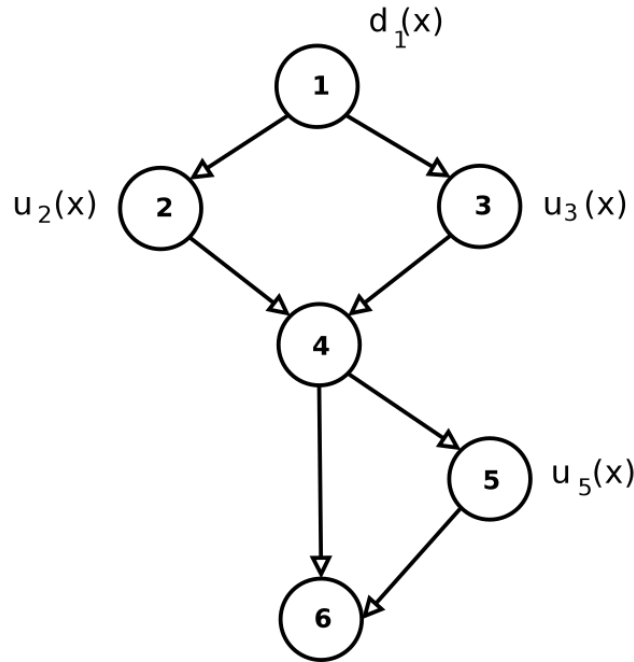
In particular we will be looking at the *data-flow path selection* criteria due to Rapps and Weyuker. The work is reported in [2].

The aim in data-flow based testing methods is to select test cases that traverse paths from nodes that *define* variables to nodes that *use* those variables, and ultimately to nodes that *undefine* those variables.

### 4.4.3 Coverage-Based Criteria

As with control-flow testing, the selection of test cases for data-flow testing involves achieving certain types of coverage on the graph, as defined by criteria. The most common forms of data-flow graph coverage criteria are the following.

- **All-Defs** — For the All-Defs criterion we require that there is some definition-clear sub-path from *all definitions* of a variable to a single use of that variable. For example, consider the following data-flow graph:



For a test suite to satisfy that All-Defs criteria, we would need to test at least one path from the single definition of  $x$ , to at least one use. A single test case is sufficient for this. The paths 1, 2, 4, 6 or the path 1, 3, 4, 5 would be satisfactory.

- **All-Uses** — The All-Uses criteria requires some definition-clear sub-path from *all definitions* of a variable to *all uses* reached by that definition. For example, consider the following data-flow graph:

The All-Uses criteria requires that we test  $d_1(x)$  to each use and its successor nodes.

1.  $d_1(x)$  to  $u_2(x)$ ;
2.  $d_1(x)$  to  $u_3(x)$ ; and
3.  $d_1(x)$  to  $u_5(x)$ .

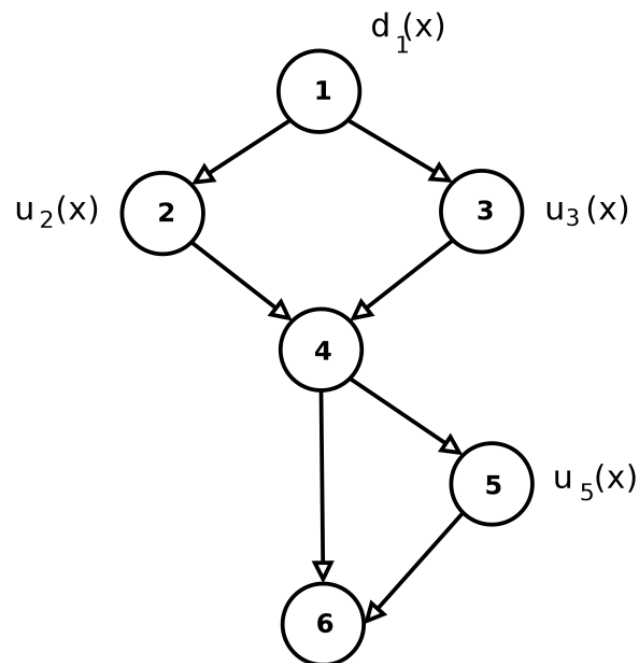
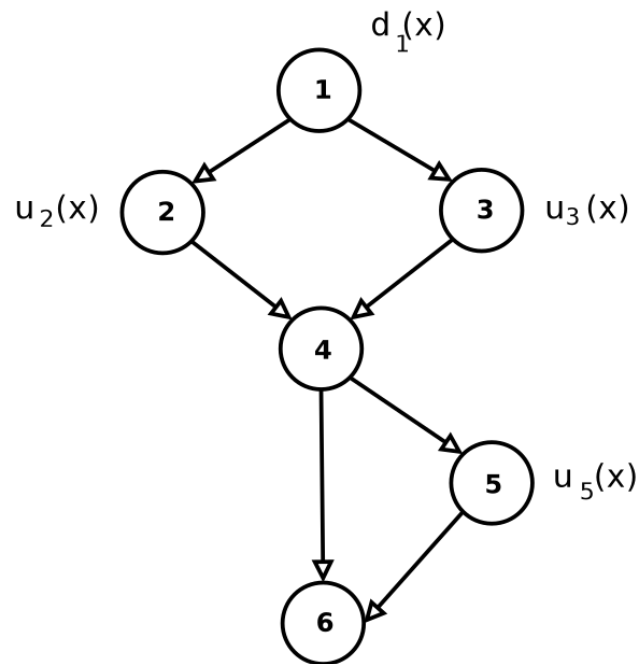
A test suite that traverses the paths 1, 2, 4, 5, 6 and 1, 3, 4, 6 are satisfactory under this criteria.

- **All-DU-Paths** — Here **DU** stands for *definition-use*. The All-DU-Paths criterion requires that a test set traverse *all definition-clear sub-paths* that are cycle-free or simple-cycles from *all definitions* to *all uses* reached by that definition, and every successor node of that use.

The All-DU-Paths criteria requires that we test  $d_1(x)$  to each use.

1.  $d_1(x)$  to a  $u_2(x)$ ;
2.  $d_1(x)$  to a  $u_3(x)$ ;
3. both paths from  $d_1(x)$  to  $u_5(x)$ .

Under this criteria, satisfactory paths are given by 1, 2, 4, 5, 6 and 1, 3, 4, 5, 6.



Recall from Definition 32, that a **P-use** of a variable is its use in a predicate, and a **C-use** of a variable is a use in a computation. Using these definitions, we can define the following additional data-flow test input selection criteria.

- **All-C-Uses, Some-P-Uses** — The All-C-Uses, Some-P-Uses criteria requires a test set to traverse some definition-clear sub-path from each definition to each C-Use reached by that definition.  
If no C-Uses are reached by a definition, then some definition-clear sub-path from that definition to at least one P-Use reached by that definition.
- **All-P-Uses, Some-C-Uses** — The All-P-Uses, Some-C-Uses requires that a test set to traverse some definition-clear sub-path from each definition to each P-Use reached by that definition and each successor node of the use.  
If no P-Uses are reached by a definition, then some definition-clear sub-path from that definition to at least one C-Use reached by that definition.
- **All-P-Uses** — Some definition-clear sub-path from each definition to each P-Use reached by that definition and each successor node of the use

#### 4.4.4 Tool Support

As with control-flow testing, manually applying data-flow testing is unlikely to yield outstanding results. However, there are parts of the process that are automatable. For example, the derivation of the data-flow graph is quite easily automated, due to its mechanical nature.

Derivation of test inputs from a data-flow graph would be considerably more difficult, especially for programs with non-linear domains, and for non-primitive data types. However, as with control-flow coverage, it is significantly easier to measure that a test suite has achieved some criterion by executing the suite over the program.

##### Footnotes

Coverlipse: See <http://coverlipse.sourceforge.net/>

**Coverlipse** is an open-source application that automatically determines whether a test suite achieves all-uses coverage, and provides feedback as to the paths that are missed by the test suite.

## 4.5 Mutation Analysis

##### Footnotes

[2]: In R. DeMillo, R. Lipton, and F. Sayward, Hints on Test Data Selection: Help for the Practicing Programmer, *IEEE Computer*, 11(4):34–41, 1978.

*“Programmers have one great advantage that is almost never exploited: > they create programs that are **close** to being correct!” — > Richard DeMillo [2]*

Recall the squeeze function from Section *Programs*, shown again below.

```

void squeeze(char s[], int c)
{
    int i, j;
    for (i = j = 0; s[i] != '\0'; i++) {
        if (s[i] != c) {
            s[j++] = s[i];
        }
    }
    s[j] = '\0';
}

```

Now consider an incorrect implementation of this function, in which the line

```
s[j++] = s[i];
```

is replaced with

```
s[j] = s[i];
```

That is, the variable *j* is never incremented, so the function places all references to *c* at position 0 in the array, and then terminates the string at position 0 (the last statement in the program). Clearly, this implementation contains a fault.

If we have a test suite for the *squeeze* function, and it is executed on the faulty version, one of two things can happen: 1) an error is encountered, and the programmer can debug the program to find the fault and repair it; or 2) an error is not encountered, and all test cases pass.

The latter occurs every day as part of every programmer's testing – whether they like to admit it or not is another story. Faults in programs go undetected because the test cases do not produce failures. When a fault is subsequently found, the test suite is (hopefully!) updated to include a test case that finds this fault, the fault is repaired, and the test suite is run again. The fact that the fault went undetected in the original test suite, and that the test suite is updated, means that the initial test suite was *inadequate*: there was a fault in the program that went undetected by it.

So far in these notes, we have discussed methods that aim to find faults by achieving coverage, but which give us little idea as to how good the resulting test suite is. In this section, we present *mutation analysis*: a method for measuring the effectiveness of test suites, with the side effect that we produce new test cases to be added to that test suite.

---

### Definition 35

Given a program, a *mutant* of that program is a copy of the program, but with *one* slight *syntactic* change. The term “mutant” is an analogy for a biological mutant, in which small parts of a nucleotide sequence are modified by access to ionising radiation etc.

---

The example of the fault in the *squeeze* function is a mutant. In this case, the slight syntactic change is the removal of the *++* operator in the array index.

---

### Definition 36

Given a program, a test input for that program, and a mutant, we say that the test case *kills* the mutant if and only if the output of the program and the mutant differ for the test inputs.

- A mutant that has been killed is said to be *dead*.
  - A mutant that has not been killed is said to be *alive*.
-

### 4.5.1 The Coupling Effect

The theory behind the mutation analysis is related to the quote from DeMillo at the start of this section. Programmers do not create programs at random, but rather write programs that are close to being correct, and continue to improve them so that they are closer to being correct over time. During this process, they learn the types of faults that programmers commonly make, and this experience is valuable in software testing. Most of these faults are either incorrect control flow of a program, or an incorrect computation, but the programs are close to their expected behaviour.

---

**Definition 37**

The *coupling effect* states that a test case that distinguishes a small fault in a program by identifying unexpected behaviour is so sensitive that it will distinguish more complex faults; that is, complex faults in programs are *coupled* with simple faults.

---

This is perhaps nothing Earth-shattering to anyone with experience in programming. If one is to look back over their repository logs, they would likely agree that many of the failures produced by their test suites are a result of one minor fault, or are so incorrect that almost any simple test case would uncover it. However, the coupling effect does have an impact on how we select test cases, whether via mutation analysis, equivalence partitioning, or any method. It implies that we should select test cases to find simple faults, and this will result in the same test cases finding larger ones.

The coupling effect has not been proven, and in fact, can not be proven. However, empirical studies of the types of faults that are made in programs gives significant support to the theory.

### 4.5.2 The Coupling Effect, Mutants, and Testing

Recall the mutant of the squeeze program from the start of this section. We labelled any test suite that did not uncover this mutant as *inadequate*. But what relationship does this have to the test suites of programs that we are testing? In the squeeze example, we know where the fault is, so by not finding it, it is clear that the test suite is inadequate.

However, consider the following scenario. We are testing the original squeeze function with a test suite, and all of our test cases pass. To evaluate the quality of the test suite, we *deliberately* insert the fault into the program by removing the ++ operator in the array index, creating a *mutant* of the program, and then run the test suite again. If this produces a failure, then the fault has been uncovered, and the mutant is killed. However, if a failure is *not* produced, then we know that the test suite is inadequate, because there is at least one possible fault that it fails to uncover. If it fails to uncover this slight fault, then it would likely fail to uncover other faults. This is the process of *mutation analysis*.

In the latter case, the tester would then aim to find a test case that does kill the mutant, and add this to the test suite. Therefore, mutation analysis can be used to guide test input generation — *we must find a test case that kills every mutant* — as well as way of assessing test suite quality — *a test suite that kills more mutants than another is of higher quality*. Specifically, if test suite *S* kills all of the mutants that *T* kills, *plus some additional mutants*, then *S* subsumes *T*.

### 4.5.3 Systematic Mutation Analysis via Mutant Operators

Like other approaches to testing, mutation analysis is only effective if applied systematically. This is done using *mutant operators*.

---

**Definition 38**

A *mutant operator* is a transformation rule that, given a program, generates a mutant for that program.

---

---

**Example 39: Relational Operator Replacement rule for Java**

---



The *relational operator replacement* rule takes an occurrence of a relational operator,  $<$ ,  $=<$ ,  $>$ ,  $>=$ ,  $=$ , or  $!=$ , replaces that occurrence with one of every other type of relational operator, and replaces the entire proposition in which that operator occurs with `and`.

Therefore, if the statement `if (x < y)` occurs in a program, the following seven mutants will be created:

- `if (x =< y)`
- `if (x > y)`
- `if (x >= y)`
- `if (x == y)`
- `if (x != y)`
- `if (true)`
- `if (false)`

This operator mimics some of the problems that programmers commonly make regarding the evaluation of Boolean expressions, which typically lead to incorrect paths.

### Example 39: Arithmetic Value Insertion rule for Java

The *Arithmetic Value Insertion* rule takes an arithmetic expression and replaces it with its application to the absolute value function, the negation of the absolute value function, and *fail-on-zero* function, in which the fail-on-zero throws an exception if the expression evaluates to zero.

Therefore, if the statement `x = 5` occurs in a program, the following three mutants will be created:

- `x = abs(5)`
- `x = -abs(5)`
- `x = failOnZero(5)`

This operator is designed to enforce the addition of test cases that consider the case in which every arithmetic expression to evaluate to zero, a negative value, and a positive value. This mimics some of the problems that programmers commonly make, such as dividing by zero.

Mutants are *syntactic* changes to a program, so mutant operators are dependent on the syntax of programming languages. Therefore, the mutant operators of one programming language may not necessarily apply to other languages.

### Remark

It is important to note that mutant operators must produce mutants that are *syntactically valid*. It is possible to create mutants of programs that are not syntactically valid, however, in most programming languages, a compiler will detect these, so they cannot be killed by a test case. One could consider that they are killed by the compiler, however, syntactically invalid mutants will always be caught by a correct implementation of a compiler, so they give us no insight into test suite quality or test input generation. Even if a language is dynamically interpreted, a good collection of mutant operators will ensure that any statically invalid programs are killed.

### Footnotes

[1]: P. Ammann and J. Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008.

In addition to the two mutant operators outlined above, Ammann and Offutt [1] define the following mutant operators for the Java programming language.

- *Arithmetic Operator Replacement*: Replace each occurrence of an arithmetic operator `+`, `-`, `*`, `/`, `**`, and `%` with each of the other operators, and also replace this with the left operand and right operand (for example, replace `x + y` with `x` and with `y`).
- *Conditional Operator Replacement*: Replace each occurrence of a logical operator `&&`, `||`, `&`, `|`, and `^` with each of the other operators, and also replace this entire expression with the left operand and the right operand.
- *Shift Operator Replacement*: Replace each occurrence of the shift operators `<<`, `>>`, and `>>>` with each of the other operators, and also replace the entire expression with the left operand.
- *Logical Operator Replacement*: Replace each occurrence of a bitwise logical operator `&`, `|`, and `^` with each of the other operators, and also replace the entire expression with the left and right operands.
- *Assignment Operator Replacement*: Replace each occurrence of the assignment operators `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, and `>>>=` with each of the other operators.
- *Unary Operator Insertion*: Insert each unary operator `+`, `-`, `!`, and `~` before each expression of the correct type. For example, replace `if (x = 5)` with `if (!(x = 5))`.
- *Unary Operator Deletion*: Delete each occurrence of a unary operator.
- *Scalar Variable Replacement*: Replace each reference to a variable in a program by every other variable of the same type that is in the same scope.
- *Bomb Statement Replacement*: Replace every statement with a call to a special `Bomb()` method, which throws an exception. This is to enforce statement coverage, and in fact, only one call to `Bomb()` is required in every program block.

To systematically generate mutants, each of these operators is applied to every statement of a program to which it is applicable, generating a number of mutants for the program. Each of these mutants is then tested using the test suite, and the percentage of mutants killed by the test suite is noted.

---

**Definition 42**

The *mutation score* of a test suite for a program is the percentage of the mutants killed by that test suite, calculated by dividing the number of killed mutants by the number of total mutants:

$$\text{mutation score} = \frac{\text{mutants killed}}{\text{total mutants}}$$

---

## 4.5.4 Equivalent Mutants

A major problem with mutation analysis is the *equivalent mutant* problem.

---

**Definition 43**

Given a program and a mutation of that program, the mutant is said to be an *equivalent mutant* if, for every input, the program and the mutant produce the same output.

---

An equivalent mutant cannot be killed by any test case, because it is equivalent with the original program. As an example of an equivalent mutant, consider the `squeeze` function. Using the *Arithmetic Value Insertion* rule, the line

```
s[j++] = s[i];
```

can be mutated to

```
s[j++] = s[abs(i)];
```

in which `abs` is the absolute value function. However, the variable `i` only ever takes on values between 0 and the size of the array, so `abs(i)` will be equivalent to `i` irrelevant of the test inputs that we choose. Therefore, this mutant is equivalent to the original program, and cannot be killed.

Equivalent mutants pose problems because they are impossible to kill, and difficult to detect. That is, once a test suite has been run over a collection of mutants, it is difficult to determine which mutants have not been killed due to the test suite being inadequate, or due to them being equivalent. Computing whether two programs are equivalent is undecidable, so a manual analysis needs to be undertaken for many instances, although some automated techniques exist that apply heuristics to eliminate equivalent mutants.

The equivalent mutant problem implies that, for many programs, an *adequate* test suite — that is, one that kills every mutant — is unachievable.

In many practical applications of mutation analysis, a *threshold* score is targeted — for example, the tester aims to kill 95% percent of the mutants, and continues trying to kill mutants until that threshold is reached.

---

### Exercise

Using the mutant operators from above, define all of the mutants for the `squeeze` function. Identify which of these mutants are equivalent.

---

## 4.5.5 Tool Support

Mutation analysis is an expensive process. Firstly, it can generate quite a large number of mutants. In fact, the number of mutants grow exponentially with the size of the program. Generating these mutants manually is not feasible for anything other than small programs. Secondly, it requires the tester to execute the test suite over every mutant, and most likely to iterate this execution a number of times.

For these reasons, a lot of work has been put into tools for automatic mutant generation, such as **MuJava**, a mutant generator for Java that uses the above rules, **Jumble**, a Java mutant generator that mutates bytecode instead of source code, and **PITest**, which also works on Java bytecode. These tools automatically generate the mutants for a program given its sourcecode or bytecode, and, given a test suite, automatically execute the mutants, and produce a report outlining the mutation score, and which mutants are still alive.

---

### Footnotes

MuJava: See <http://cs.gmu.edu/~offutt/mujava/>.

Jumble: See <http://jumble.sourceforge.net/>.

PITest: See <http://pitest.org/>

---

Unfortunately, the equivalent mutant problem is not solved by the above tools. However, research into mutation analysis is leading to cheaper ways of automatically eliminating equivalent mutants using constraint solvers and compiler optimisation techniques. Applying these tools in practice is a long way off yet.

At this cost comes benefit: mutation testing is considered to be the most successful test coverage criterion for finding faults. In the next section, we compare the three coverage criteria presented in this chapter.

## 4.6 Comparing Coverage Criteria

A question that often arises in practice is: *which criteria gives the best coverage?*

The way to compare these criteria is to defined a specific relation between the criteria. The relation in question is called *subsumption* and is defined as follows.

### Definition

Criterion *A* *subsumes* criterion *B* if and only if for any control-flow graph *P*:

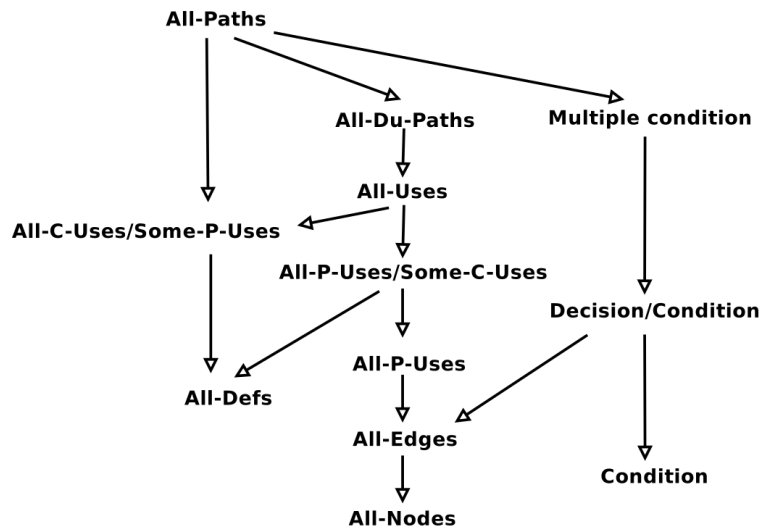
$$P \text{ satisfies } A \implies P \text{ satisfies } B$$

Criteria *A* is equivalent to criteria *B* if and only if *A* subsumes *B*, and *B* subsumes *A*.

The *subsumes* relation is transitive, therefore, if *A* subsumes *B*, and *B* subsumes *C*, then *A* subsumes *C*.

*How can we compare these criteria?*

Both data-flow criteria and coverage criteria select a set of paths that must be traversed by the test cases. To compare across different criteria Frankl and Weyuker have compared the paths that each criteria selects. Note, however, that the set of paths that satisfy a criterion are not necessarily unique. The results are shown in Figure 4.10.



It is straightforward to see the relationships between the different data-flow techniques, and between the different control-flow techniques. For example, branch coverage (**All-Edges**) subsumes statement coverage (**All-Nodes**) because every statement is located within a branch.

The relationships between the different control- and data-flow criteria are less clear to see. Condition coverage does not subsume statement coverage. This can be illustrated by the example `if (a && b)`. Condition coverage mandates that a test suite must exercise *a* as both `true` and `false`, and *b* as both `true` and `false`. This can be achieved with two test cases: `(a == true, b == false)`, and `(a == false, b == true)`. However, these test cases never execute the case that *a* and *b* are *both* true, therefore, the statements in that branch are not executed.

Multiple condition coverage and the criteria that it subsumes do not subsume any of the data-flow criteria. This can be illustrated with the following simple program:

```

if (y == 1) {
    x = 2;
}

if (z == 1) {
    a = f(x)
}

```

in which  $f$  is some function. In this example, multiple condition coverage enforces that  $y == 1$  and  $z == 1$  are both executed for the and cases. To achieve any of the data-flow criterion, at least one test must execute a definition-clear path from the statement  $x = 2$  to  $a = f(x)$ . However, multiple condition coverage does not enforce this, therefore, it does not subsume any of the data-flow criteria.

Similarly, none of the data-flow coverage criteria subsume any of the coverage metrics related to conditions or decisions. If we consider the weakest of these, condition coverage, the none of the data-flow criteria will evaluate the following statement to :

```

if (x == 1) {
    a := f(x)
}

```

because there is no block related to the case, and therefore no variable uses in it.

Decision/condition coverage clearly subsumes both branch and condition coverage, as it is a combination of the two. The *subsumes* relation is transitive, therefore, decision/condition coverage also subsumes statement coverage, because branch coverage does. Multiple-condition coverage subsumes decision/condition coverage, and path coverage (if possible) subsumes branch coverage.

Mutation analysis has not been discussed so far. It is difficult to evaluate because mutant operators are defined specific to programming languages. However, one can easily take the operators defined by Ammann and Offutt for Java, described earlier, and apply them to many programming languages.

These operators subsume many of the criteria in Figure 4.10, including multiple-condition coverage, and All-Defs coverage. This can be proved in these cases. For example, any test suite that kills all non-equivalent mutants generated by the *Bomb Statement Replacement* rule is guaranteed to achieve statement coverage (or node coverage in a control-flow graph), because every statement is replaced by a call to `Bomb()`. Similarly, any test suite that kills all non-equivalent mutants generated using the *Conditional Operator Replacement* rule is guaranteed to achieve multiple-condition coverage.

### Footnotes

[1]: P. Ammann and J. Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008.

In Section 5.2.2. of [1], Ammann and Offutt prove that mutation analysis is stronger than many of the coverage criteria in Figure 4.10. They hypothesise that even though some of them are not subsumed by mutation analysis, it is likely that specific mutant operators could be derived to achieve this, but that there is little benefit in doing so.

### 4.6.1 Effectiveness of Coverage Criterion

The effectiveness of these criteria is important to consider. If we go to the effort of measuring coverage and adding tests to ensure that we achieve coverage, we'd like to know that we are adding value to our test suite.

---

**Footnotes**

[3]: L. Inozemtseva and R. Holmes, Coverage is not strongly correlated with test suite effectiveness, *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014.

[4]: R. Just, et al., Are mutants a valid substitute for real faults in software testing?, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014.

---

Several studies have looked into the effectiveness of test coverage criteria. For example, a 2014 study [3] looked into the correlation between coverage and fault-finding ability, considering statement coverage, branch coverage, and modified-condition coverage using 31,000 test suites, and found that there is a low correlation between test suite coverage and fault-finding effectiveness. Further, they found that stronger forms of coverage provide very little value. Another 2014 study [4] showed that mutation score provides is much better correlated with fault-finding ability, which is unsurprising because the mutation score is high if a test suite finds seeded faults.

These studies confirmed a long-held view in software engineering that structural coverage criteria, such as control- and data-flow criteria, should be used to identify areas of the program that remain untested, and not as measure of the quality of the test suite.

In short, structural test coverage is a *necessary but not sufficient* method for software testing.

## TESTING MODULES

### 5.1 Learning outcomes of this chapter

At the end of this chapter, you should be able to:

- Discuss the impact that observability and controllability have when testing modules.
- Construct a finite state automaton from the specification of a module.
- Derive a set of test sequences that achieve the different coverage criterion of a finite state automaton.
- Derive a set of test sequences from a finite state automaton that intrusively and non-intrusively test the module.
- Describe the pros and cons of polymorphism and inheritance in testing object-oriented programs.
- Derive a test suite that takes advantage of inheritance and counteracts the problems of polymorphism.

### 5.2 Chapter Introduction

In the previous chapters, we have discussed how to select test inputs to maximise the chances of uncovering faults in a program. Examples in these chapters have been single functions, with only parameters as input, and return values as output. In this chapter, we discuss the testing of program with state; that is, testing *modules*. Modules present some different problems with regards to testing when compared to single functions.

### 5.3 State and Programs

During the years proceeding the wide-spread adoption of third-generation programming languages, computer programs started becoming more and more complex. Practitioners realised that, instead of allowing any part of a program to manipulate any part of data, if access to data was provided via a clear and unambiguous interface made up of functions on that data, then complex program could be understood more readily.

---

#### Footnotes

Strictly, a module does not require a state, but can be simply a collection of related operations.

---

Typically, the data, called the *state*, is manipulated and accessed via a collection of *operations* on that state. Collectively, these are referred to as a *module* [1]. Using modules enforces a separation of interface and implementation, allowing programmers to think of these collections of operations as a black box. Specific details about the data is hidden from the user of the module, and a change in the underlying data structure has no impact on the program that uses it as long as the interface remains the same.

Operations are meaningless when separated from their state and treated in isolation. For example, an operation may require the module to be in a specific state before it can be executed, and that state can only be set by another operation (or combination of operations) in the module.

Object-oriented programs are special classes of state-based programs, in that multiple instances of the data can be created.

## 5.4 Testability of State-Based Programs

Recall from Section *Testability* that the *testability* of a program is defined by the *observability* of the program, and the *controllability* of the program. State-based programs present some issues with regards to these two measures, even for APIs.

For example, consider a module that allows the manipulation of and access to a *stack*. The operations of the class allow us to push an element on to the stack, view the top of the stack, pop the top element from the stack, and check whether the stack is empty. The specification of the class is allows:

Initially, the stack is empty — it contains no elements. When created, the calling program determines a maximum size for the stack, passed as a parameter to the constructor.

An element can be *pushed* onto the stack if the size of the stack is not already equal to the maximum size, as determined when initialised. If full, pushing another element will result in the exception `StackFullException`.

The top element of the stack can be *popped* from the stack provided that the stack is not empty. Pushing an element and then popping the stack will result in the stack before pushing. If the stack is empty, popping will result in the exception `StackEmptyException`.

The calling program can see the value of the top of the stack, but no other parts of the stack. If the stack is empty, peeking at the top of the stack will result in the exception `StackEmptyException`.

Finally, the stack can be checked to see if it is empty, and can be checked to see if it is full. These operations allowing the calling program to avoid exceptions.

Figure 5.1 shows an implementation of such a module in Java, in which the stack has a maximum size, which is determined by the calling program when creating the stack data type. The variables prefixed with an underscore (`_`) as those that are encapsulated by the class, and are not directly controllable or observable.

```
public class Stack
{
    private Object _stack[];
    private int    _maxsize = 0;
    private int    _top = 0;

    public void Stack(int maxsize)
    {
        _maxsize = maxsize;
        _top = 0;
        _stack = new Object[maxsize];
    }

    public void push(Object n)
```

(continues on next page)



(continued from previous page)

```

        throws StackFullException
    {
        if (_top == _maxsize)
            throw new StackFullException();
        else {
            _stack[_top] = n;
            _top++;
        }
    }

    public void pop()
        throws StackEmptyException
    {
        if (_top == 0)
            throw new StackEmptyException();
        else
            _top--;
    }

    public Object top()
        throws StackEmptyException
    {
        return _stack[_top];
    }

    public boolean isEmpty()
    {
        return (_top == 0);
    }

    public boolean isFull()
    {
        return (_top == _maxsize);
    }
}

```

Using the techniques discussed in the previous chapters, we can derive some test cases to test the operation that pushes an element from the stack. Any good test suite will include at least equivalence classes for an empty stack, and a stack with at least one element. After selecting values from these equivalence classes, we can derive the executable test cases.

However, access to the stack data is hidden behind the interface. The `push` operation does not take a parameter for the stack, meaning that the `push` operation cannot be *controlled* via its interface. Furthermore, once an element is pushed onto the stack, we cannot directly *observe* the value of the stack. Therefore, the operations as part of the stack module are not easily controllable or observable.

This reduction in controllability and observability means that unit testing in isolation is not possible for the operations of the stack module. In cases such as this, the other operations that make up the module may be required to test an operation, so we have a case in which unit testing and module testing can be considered as the same activity.

One way around these testability issues is to *intrusively* test the module. That is, the tester modifies the program code to give access to the data type that is hidden, therefore breaking the information hiding aspect of the module. The program is tested, and the data type is hidden again. This has serious drawbacks:

- it does not test the implementation that is to be used, but an altered version of it;
- if the underlying data type in the module changes, then the tests must be changed as well; and
- it ignores the fact that operations acting on the same data as inherently linked, and must be tested together.

One could argue that the inter-dependencies could be tested as well, but with modules such as the stack class, the high-level of dependency between operations implies that testing an operation in isolation would give us little benefit.

Rather than intrusively testing the module, we use a *non-intrusive* method: we use the operations defined by the module to set the state of the module to test value that we want, perform the necessary test, and then use the operations again to query the result of the test. In the example of the stack class, to test the behaviour of pushing an element onto a stack containing one element, we first push an element onto an empty stack using the `push` (setting up the state for our test), then push another element for our test, then use the `top` and `pop` operations to see the state of the module.

Note that many object-oriented languages provide us with constructs that are somewhat in between intrusive and non-intrusive testings. For example, in the Java programming language, variables inside classes can be declared as *protected*, which means that if we can inherit the class, then we can access the variables in the class. This allows us to passively observe the values of variables without changing the module itself. Other tools provide non-intrusive ways to observe data, such as inserting breakpoints in programs, and allowing the tester to observe the state of the program at that breakpoint.

## 5.5 Unit Testing with Finite State Automata

The way that many modules, especially those that implement abstract data types, are abstracted is to envisage them as *finite state automata*.

---

### Definition 45

A *finite state automaton* (FSA) or *finite state machine* is a model of behaviour consisting of a finite set of states with actions that move the automata from one state to another.

---

With regards to a module, each state in a FSA corresponds to a *set* of states in the module it is modelling. The transitions between states correspond to the operations in the module. Transitions can be prefixed with predicates specifying the preconditions that must hold for them to take place.

FSAs are useful for modelling behaviour of programs, and can be used to derive test cases for such programs. The states of the automaton derived from subsets of the states of the data encapsulated in the module, and the transition of the automaton are the operations of the module. Test cases are selected to test sequences of transitions in this automaton.

### 5.5.1 Constructing a FSA

To construct a finite state machine, we perform the following steps.

- **Step 1:** identify the states of the FSA. Each state in the FSA corresponds to a set of states in the module. Intuitively, we want to consider states with same effect on operations as being *equivalent* and collect them together into a single FSA state. The situation is analogous to the situation in equivalence partitioning where every element of an equivalence class has the same chance of finding an error. Therefore, we use the techniques described in the previous chapters to derive the states that we want to test.
- **Step 2:** identify which operations are enabled in a state and which operations are not enabled in a state — an operation is enabled if it can be called safely without error in a given state.

For example, the `pop` and `top` operations are not enabled if the stack is empty, and the `push` operation is not enabled if the stack is full.

From this information, we identify the *start* state(s), the *exit* state(s) (those that have no enabled operations).

- **Step 3:** is to identify the source and target states of every operation in the model. To do this, we take every state and every operation that is enabled in that state, and calculate the state that results in applying that operation.

For example, if we derive a state specifying the stack as empty (state  $s_0$ ), and a state specifying the stack having one element (state  $s_1$ ), then the `push` operation links  $s_0$  to  $s_1$ , the `pop` operation links  $s_1$  to  $s_0$ , and the `isEmpty` operation links both states to themselves. If modelling exceptions, then applying the `pop` operation in the empty state will result in the state `StackEmptyException`.

#### Example 46: Deriving the stack FSA

As a first example, consider a simple `Stack` class that we wish to test. An informal specification for this is given in Section *Testability of State-Based Programs*, and an implementation of that specification is given in Figure 5.2.

Firstly, we want to derive the states of the FSA. Intuitively, we want to consider states with same effect on operations as being *equivalent* and collect them together into a single FSA state. The equivalence partitioning techniques can be used to derive this. Although equivalence partitioning is an *input* partitioning technique, the state can be considered as input to any operation that uses it, but it is input that is not passed as a parameter.

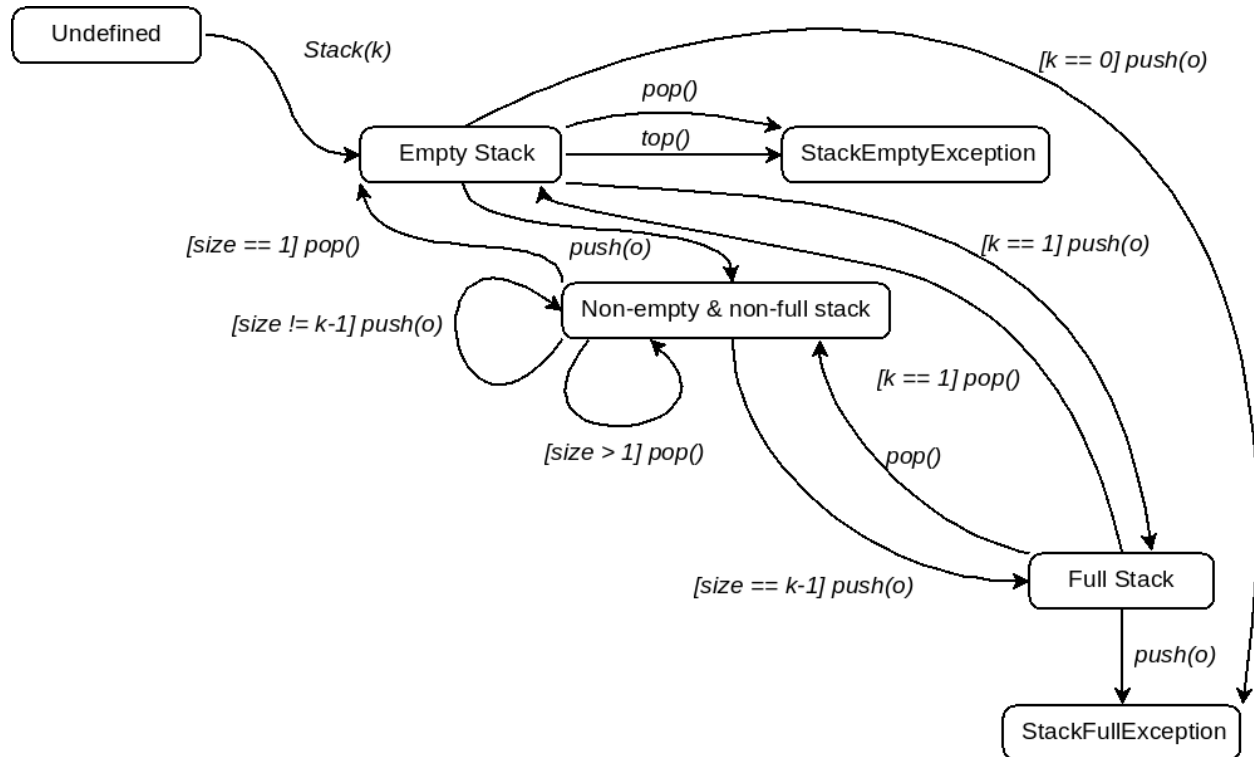
Instead of looking at the input domain we look at the states of a module and ask what states can be considered equivalent if we want to find faults in the module operations. We do this for the `Stack` class in Table 5.1.

State	Description	Enabled Operations
State 1	An undefined stack	Stack constructor
State 2	A defined stack that is empty; that is, a stack with no values	push, isEmpty, isFull
State 3	A stack that is full; that is, it has the maximum number of elements	pop, top, isEmpty, isFull
State 4	A stack that is defined but is neither empty nor full	push, pop, top, isEmpty, isFull
State 5	A stack that is in the <code>StackEmptyException</code> ; that is, the caller has attempted to pop or see the top of an empty stack	none
State 6	A stack that is in the <code>StackFullException</code> , that is, the caller has tried to push an element onto a full stack	none

The *initial* state of this module is the undefined stack. Alternatively, one could remove the undefined state, and instead specify that the initial state is the empty state. The exit states are the exception states.

The exit states depend on the the specification for the module. For example, popping an empty stack throws an exception, however, an alternative specification of the stack may *assume* that the stack is not empty when elements are popped, therefore, the behaviour of popping an empty stack is undefined, and cannot be meaningfully tested. In this case, there are no exit states (there is always one operation that can be executed in any state).

Finally, we derive the transitions between the states. The final FSA for the stack module is shown in Figure 5.2. The transitions of `isEmpty` and `isFull` are omitted because they are enabled in all states (except the undefined state) and always loop back to the same state from which they started.



In this figure, the predicate in the square brackets on the transition labels represent the conditions that must hold for that transition to take place. Note that these are not the preconditions of the operations themselves, but merely the conditions that must hold for that operation to change to the destination state.

Note the `push` transition directly from the empty state to the full state, with the condition  $k == 1$ , is for when the maximum size is 1, in which case there is no state in which the stack can be non-empty and non-full. Similarly for the reverse action: popping the stack from a full state, and for pushing onto a stack of maximum size 0, which leads to an exception state.

## 5.5.2 Deriving Test Cases from a FSA

Recall from Section *Testability of State-Based Programs* that the observability and controllability of modules is low due to their hidden information. We discussed that fact that to set up a state for a test input of an operation may require us to execute other operations in the module.

The FSA of a module gives us the information that is required to do so. If the states in the FSA represent the input states that we wish to test, then the transitions that move the state from the initial state to each state represent the operations that need to be executed.

### Definition

A *path* from a state  $s_0$  to a state  $s_n$  in the automaton is a sequence of transitions  $t_1 t_2 \dots t_n$  such that the source of  $t_1$  is  $s_0$  and the target of  $t_n$  is  $s_n$ .

The aim is to generate *test sequences* such that the paths in the automaton are exercised. A test case is no longer a single function call but may require a sequence of operation calls to force the module along a certain [path](#). To derive these sequences, we define *traversal criteria* for a FSA. Broadly, there are three criteria that we can use:

1. *\*State coverage*: Each state in the FSA must be reached by the traversal.
2. **Transition coverage**: Each transition in the FSA must be traversed. This subsumes state coverage.
3. **Path coverage**: Each path in the FSA must be traversed. This subsumes state coverage, but is impossible to achieve for a FSA with cyclic paths.

State coverage is clearly inadequate, because there are transitions in the FSA that will not be traversed, and therefore, operations that will not be tested for the test states that were derived. Path coverage is impossible for FSAs with cycles, so this is infeasible in many cases. Transition coverage is feasible, straightforward to achieve, and it tests every test state that was derived, so this is generally the most practiced traversal technique.

Using transition coverage, we repeatedly derive test sequences starting at the initial node until all transitions have been covered at least once. In some cases, this can be done using one long sequence, but in others, this is not possible; for example, the stack FSA as two exit states, and one of these has two input transitions, which together imply that there must be at least three sequences. Often, it is more efficient to derive many short sequences than a few long sequences.

### 5.5.3 Intrusively Testing the State Transition Diagram

Now we can look at intrusive and non-intrusive methods for testing an automaton.

As discussed earlier, *intrusive testing*, in these notes at least, will mean adding testing code to the module to measure and monitor the internal states of the module as it is tested. The problem with this kind of testing is that it breaks encapsulation.

The points at which to insert testing code into a module are to observe the values of private state variables, observe the values of private operation calls and even observe intermediate states in module operations.

In an object-oriented programming language, this is not as obvious as it sounds because of the problem of inheritance. Consider the class Node for implementing a doubly linked list in the following example.

```
public class Node
{
    private Object _data;
    private Node _previous, _next;
}
```

#### Footnotes

[2]: In the case of the type Object in Java this can mean a large number of classes in theory, but in practice it is not often that large.

To observe the state of Node we need to get access to the values of the `_data` instance variable. However, because `_data` is an instance of the `Object` class, this means that we need to ensure that there are operations to access the private state elements of all the objects in the system that inherit from `Object`[2]; that is, we would need to extend all elements of the hierarchy with testing code. In this case there is a great deal of additional code to write which may well further impact the properties of the entire system.

In languages like C and C++ we can use the pre-processor to compile test code into the executable when required, or to omit the testing code when the program is to be released.

As a result of these problems, systematic intrusive testing is difficult to achieve, in fact, it is often impossible for some languages if we do not have access to the source. Many of the solutions are hacks that give the testers some benefit that outweighs the breaking of the encapsulation.

The best cases are languages that provide semi-private access, such as the `protected` keyword in Java, or the `friend` keyword in C++. These give testers the ability to read and write to the variables without changing the program-under-test.

### 5.5.4 Non-intrusive Testing the State Transition Diagram

There will be occasions when we simply do not have access to the internal structure of a module or its hierarchy. In this case, or in the case where you simply need to test a module without adding code to a module, then we have *non-intrusive* testing.

The idea is to test each of the transitions in the testing automaton implicitly by using other operations in the module to examine the results of a transition. Lets start by dividing up the operations in a module as follows:

Transition type	Description
Initialisers	The operations that initialise the module, such as the <code>Stack</code> constructor
Transformers	Operations to change the state of the module, such as <code>pop</code> operation in the stack example.
Observers	Operations to observe the module state, such as the <code>isEmpty</code> and <code>top</code> operations in the stack example

Recall that in the FSA for the stack module, shown in Figure 5.1, the `IsEmpty`, `isFull`, and `top` operations were omitted because they do not move the automaton to a new state. This is because these operations are *observers* — with the exception of `top` moving to the `StackEmptyException` state.

The technique for testing involves deriving test sequences, using initialisers and transformers, to move the automaton to the required state. At each state, observe the value of the module using the observer operations. The only case where the observer operations are not used is in the exception states, because the execution of the module is considered to have terminated. Alternatively, and depending on the programming language, these exception states could be remove from the FSA, and considered as observer behaviour. Many programming languages support the catching of exceptions, thereby allowing the execution to continue.

#### Example 47: Traversing the stack FSA

For the stack automaton in Figure 5.2, we can obtain transition coverage using the following test sequences:

Sequence	End State
<code>Stack(0); push(o)</code>	<code>StackFullException</code>
<code>Stack(1); push(o); pop(); pop()</code>	<code>StackEmptyException</code>
<code>Stack(k); top()</code>	<code>StackEmptyException</code>
<code>Stack(k); push(o)<sup>k</sup>; pop()<sup>k</sup>; push(o)<sup>k</sup>; push()</code>	<code>StackFullException</code>

in which the notation  $op()^k$  is shorthand for executing  $op()$  in sequence  $k$  times, and  $k > 1$ .

Regarding observer operations, there are several places that these can be tested:

1. all observer operations could be used to observe the state after a call to any initialiser or transformer operation;
2. all observer operations could be used to observe the state any time a transition takes us to another state in the FSA;  
or
3. all observer operations could be used to observe the state when an FSA state is being visited for the first time.

The first of these seems like overkill, and would lead to a high number of test cases. The second is more reasonable, but considering that each FSA state represents many module states, and each of these is representative of the equivalence class to which it belongs, then it seems reasonable to only test the observer operations when an FSA state is being visited

for the first time. For the stack example, this gives us the following test sequences, which we instantiate into actual test cases:

Sequence	End State
Stack(0); (isEmpty() == true); (isFull() == true); push(o)	StackFullException
Stack(1); push(1); (isEmpty() == false); (isFull() == true); (top() == 1); pop(); pop()	StackEmptyException
Stack(2); top()	StackEmptyException
Stack(2); push(1); push(2); (isEmpty() == false); (isFull() == true); (top() == 2); pop(); pop(); push(3); push(2); push(1); push(0)	StackFullException

The additional calls to the observer operations, and the checking of their values, is equivalent to adding them into the FSA as transitions from a node to itself.

#### Remark

1. Each of the test cases above consists of a sequence of operation calls rather than just a single call.
2. It is possible that a test case may consist of an equality between two sequences of operation calls, for example,

$$(Stack(); push(o); pop()) == Stack()$$

if such a test for equality can be made.

1. If operations exist for forcing a module into one of the testing states then this can make testing long sequences of operation calls easier.

## 5.5.5 Discussion

The design of state-based systems is an important part of testing. This section has outlined that it is important to define the interface of a module to make as testable (controllable and observable) as possible. Designing for testability is as important as designing to meet requirements, even though there may be conflicts between the two.

Fortunately, many modular design notations that are part of design methodologies resembled FSAs. For example, UML uses state charts precisely for the purpose of specifying and understanding the legal sequences of operations on a class instance, and the states that result from these sequences.

## Problems with State Based Testing

Of course there are some problems with FSA-based testing:

- It may take a lengthy sequence of operations to get an object into some desired state.
- FSA-based testing may not be useful if the module is designed to accept any possible sequence of operation calls. This would result in a FSA with little structure, and require a prohibitively large number of test sequences to cover.
- State control may be distributed over an entire application with operations from other modules referencing the state of the module under test.

System-wide control makes it difficult to verify a module in isolation and requires that we identify module *hierarchies* that collaborate to achieve a particular functionality. Here the collaboration and behaviour diagrams are the most useful.

## 5.6 Testing Object-Oriented Programs

Object-oriented programs pose some new and interesting challenges to unit, integration and systems testing. One can view an object-oriented system can be viewed as a modular system, and therefore can test an object-oriented system by applying the techniques already discussed in this chapter. In this case, the modules are *classes*, and the operations are *methods*.

However, there are certain properties of object-oriented languages that make testing more difficult than for other imperative languages such as C. In this section, we discuss these differences and present techniques to minimise their impact.

### 5.6.1 Object-Oriented Programming Languages

Object-oriented programming languages support a number of features that are aimed at making the design, maintenance and reuse of code much easier. We will not go into detail but will briefly review the major features and start to think about them from a *testing* viewpoint.

The idea in object-oriented programming languages is that the language provides constructs to support encapsulation and structuring. The key elements that languages such as Java and C++ provide are:

1. *Classes and objects* which provide the main units for structuring.
2. *Inheritance*, which provides the one of the key structuring mechanisms for object-oriented design and implementation.
3. *Dynamic Binding or Polymorphism*, which provides a way of choosing which object to use at run-time.

#### Classes and Objects

Classes are used as templates for creating objects. The objects do the work while the class just shows you the what sets of objects are active in the program. The common view is that a class declaration introduces a new *type* while the objects are the elements of that type. Whenever a new object is created it is an *instance*, or element, of that type. Every object in the class has:

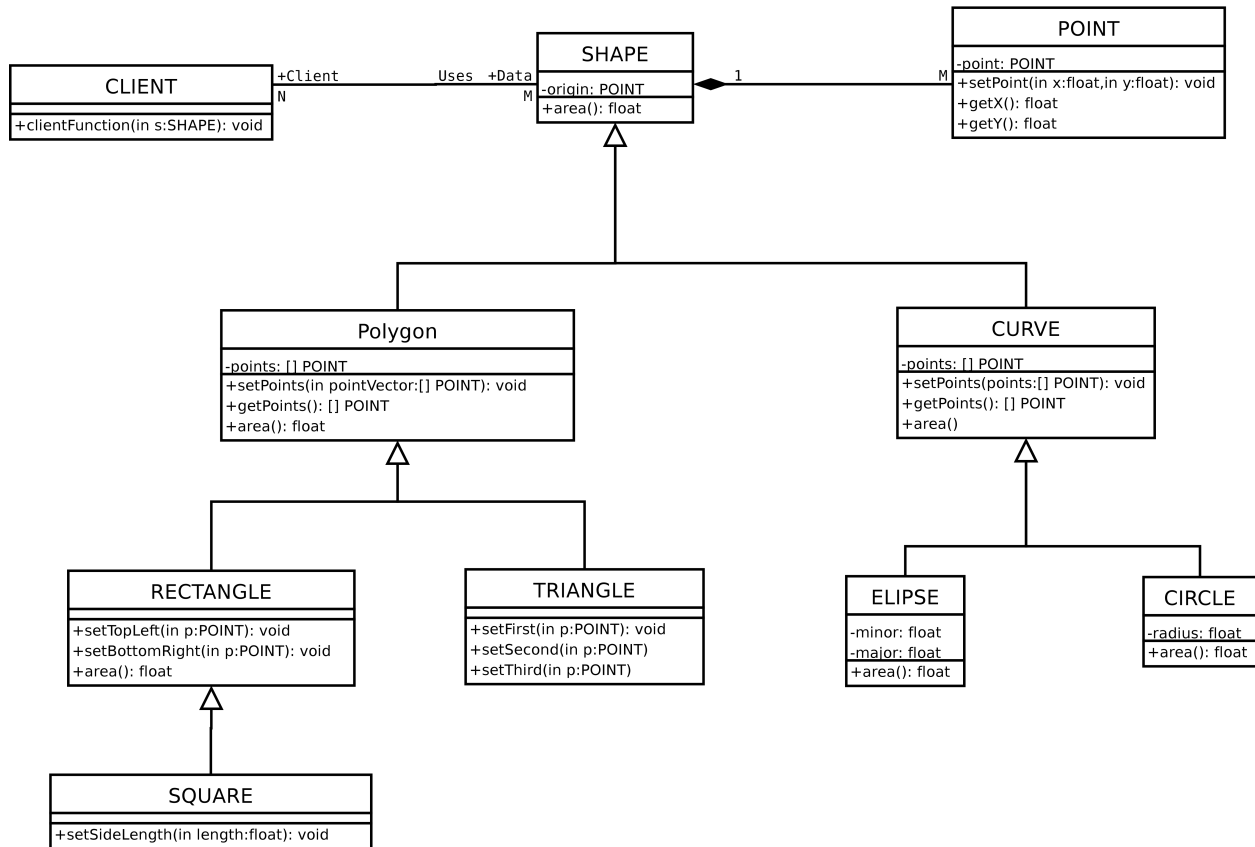
- The *methods* defined by the class; and
- The *instance variables* (or *attributes*) defined by the class;

There can be any number of objects that are instances of a class, each with different values for their instance variables but no matter what the specific values are the object will still have the attributes and methods specified by the class. This is one thing that we can rely on in testing.

For example, the class Shape in Figure 5.3 defines a type where each object of that type contains a private hidden instance variable origin and a publicly visible method area().

Classes and objects do not present any major obstacles in testing on top of the encapsulation problems in testing modules. In fact, for the purpose of testing a class in isolation, one can consider an instance of a class as simply a record type, in which we have variables bound to values, and a method call to an object as simply an operation call, but with the object as an input/output value.





## Inheritance

*Inheritance*, or *generalisation*, relationships exist between classes. The parent is the more general class providing fewer methods or more general methods while the child *specialises* the parent. A child class inherits all of the properties of its parent, but the child class may *override* operations in the parent and *extend* the parent by adding more attributes and operations.

In particular, generalisation means that the objects of the child class can be used anywhere that the objects of the parent class can be used — but not the converse. Some languages, such as Java, support only *single inheritance*, in which a class may only inherit from at most one parent class. Others, such as C++, support *multiple inheritance*, in which a class may inherit from one or more parent classes. The parent class is called the *superclass* and the child is called the *subclass*.

This idea translates to instances as well — every instance of the child class is also an instance of the parent class. Every instance the child class has the same attributes and methods as the parent — however, the child may use a different implementation of a method to that of the parent.

Inheritance is found only in object-oriented programming languages, but does not pose any immediate problems in testing. The techniques presented so far in this chapter are sufficient, however, the most immediate consequence of inheritance is that it effects the structuring of test suites.

## Polymorphism

Figure 5.3 shows an inheritance hierarchy with the class Shape as the base class in the hierarchy. Any of the classes that have Shape as the superclass can be used anywhere that Shape can be used. For example, the class Client has a method called clientFunction that takes an object of type Shape as a parameter. However, the actual parameter can be an object of type Shape, an object of type Polygon, an object of type Circle or indeed an object of any type that has Shape as one of its ancestors.

This is the essence of *dynamic binding*, or as it is often called, *polymorphism* in object-oriented languages. The combination of inheritance and dynamic binding is extremely powerful for designing reusable and extendable programs. Dynamic binding and polymorphism are both found in some non-object-oriented languages, however, this is uncommon, whereas they are the essence of object-oriented languages.

The combination of inheritance and dynamic binding creates some real headaches for testers, or at least the combination of polymorphism and inheritance must be taken into account.

### 5.6.2 Testing with Inheritance and Polymorphism

In object-oriented testing the view is that classes are:

$$\text{class} = \text{object state} + \text{set of methods}$$

The internal states of the object become relevant to the testing. What this means in testing is that the correctness of an object depends on the internal state of the object as well as the output returned by a method call.

This is the same idea of modules being a state and a set of operations, so classes are thus the natural unit for testing object-oriented programs.

However, once this choice is made then we will need to explore the effects of object-oriented testing in the presence of inheritance/generalisation and polymorphism.

### Testing Inheritance Hierarchies

Inherited features require re-testing, because every time a class inherits from its parent the state and operations of the parent are placed into new context — the context of the child. Multiple inheritance complicates this situation by increasing the number of contexts to test.

Ideally, an inheritance relationship should correspond to a problem domain specialisation, for example, from Figure 5.3 a Polygon is a special kind of Shape, a Rectangle is a special kind of Polygon and so on. The re-usability of superclass test cases depends on this idea. Unfortunately, many inheritance relationships do not respect this rule and simply inherit from classes when they want to use library functions.

*Which functions must be tested in a subclass?*

Figure 5.4 shows a simple parent-child inheritance hierarchy.

```
class Parent
{
    int number()
    {
        return 1;
    }

    float divide(int x)
    {
        return x/number();
    }
}
```

(continues on next page)

(continued from previous page)

```

    }
}

class Child extends Parent
{
    int number()
    {
        return 0;
    }
}

```

Parent is at the top of an inheritance hierarchy, so we derive test cases for this class first, and test it. When testing the Child class, we need to retest the number() method because this has been modified directly. However, we also need to retest the divide() method because it directly references number(), which has changed.

*Can tests for a parent class be reused for a child class?*

First we need to observe that parent.number() and child.number() are two different functions with different implementations. They may or may not have the same specification (there is no specification for this example); for example, the specification may say that the method must return a number between 0 and 10, but with no constraints as to which number — thus making the specification *non-deterministic*.

Test cases that are generated for these two different functions, however, are likely to be similar because the functions are similar. Thus, if we minimise the overloading by using principles such as the *open/closed principle* in our design, then there is a much greater chance that the inherited methods will not need new test cases in the context of the child class.

Clearly, new tests that are necessary will be those for which the *requirements* of the overridden methods change.

## Building and Testing Inheritance Hierarchies

When dealing with inheritance hierarchies it is important to consider both testing and building at the same time. There are two key reasons for this:

1. the first is to keep control of the number of test cases and test harnesses that need to be written; and
2. the second is to make sure that we know where faults occur within the inheritance hierarchy as much as possible.

A first approach to inheritance testing involves *flattening the inheritance hierarchy*. In effect each subclass is tested as if all inherited features were newly defined in the class under test — so we make absolutely no assumptions about the methods and attributes inherited from parent classes. Test cases for parent classes may be re-used after analysis but many test cases will be redundant and many test cases will need to be newly defined.

A second approach to testing and building is *incremental inheritance-based testing*. Incremental building and testing proceeds as follows:

**Step 1:** First test each base class by:

- Testing each method using a suitable test case selection technique; and
- Testing interactions among methods by using the techniques discussed earlier in this chapter.

**Step 2:** Then, consider all sub-classes that inherit or use (via composition or association) only those classes that have already been tested.

A child inherits the parent's test suite which is used as a basis for test planning. We only develop new test cases for those entities that are directly, or indirectly, changed.

Incremental inheritance-based testing does reduce the size of test suites, but there is an overhead in the analysis of what tests need to be changed. It certainly reduces the number of test cases that need to be selected over a flattened hierarchy.

If the test suite is structured correctly, test cases can be reused via inheritance as well, which provides the same benefits for conceptualisation and maintenance for test suite implementations as it does for product implementations.

Inheritance-based testing can also be considered to be a form of regression testing where the aim is to minimise the number of test cases needed to test a class modified by inheritance.

### **Implications of Polymorphism**

Consider the inheritance hierarchy in Figure 5.3. The implementation of `area()` that actually gets called will depend on the state of the client object and the runtime environment.

In procedural programming, procedure calls are *statically* bound — we know exactly what function will be called at compile time — and further, the implementation of functions do not change (well, not unless there is some particularly perverse programming) at runtime.

In the case of object-oriented programming, each possible binding of a polymorphic class requires a separate set of test cases. The problem for testing is to find all such bindings — after-all the exact binding used in a particular object instance will only be known at run-time.

Dynamic binding also complicates integration planning. Many service and library classes may need to be built and tested before they can be integrated to test a client class.

There are a number of possible approaches to handling the explosion of possible bindings for a variable. One approach is to try and determine the number of possible bindings through a combination of static and dynamic program analysis.

Consider the Client class in Figure 5.3. The problem from a testing point of view is to know which actual Shape object gets called at run-time. If an instance of Square is bound to the variable `s` in `clientFunction` then we would expect a different result for the `area()` computation than if that instance Circle was bound to `s`.

If we instrumented the code for Shape and all of its descendants to reveal the types of the actual objects that are bound to `S` then we could use that information to determine the subset of the class hierarchy that is actually used by the method `clientFunction`. The subset can then be covered more thoroughly with test cases. For example, it may be the case that Polygon hierarchy is used by the client function while Curve hierarchy is not. If that is the case the testing `clientFunction` and the classes in the Polygon hierarchy makes sense but testing the classes in the curve hierarchy makes less sense.

Beware however, this approach is not foolproof and is biased heavily towards the data used to generate the bindings.

## TEST ORACLES

### 6.1 Learning outcomes of this chapter

At the end of this chapter, you should be able to:

- Define the term “test oracle”.
- Contrast the different types of test oracle.
- Choose the correct type of test oracle for a given problem.
- Design an oracle for a program from a specification.

### 6.2 Chapter Introduction

In this chapter, we will discuss *test oracles*. Recall from Section *Black-Box and White-Box Testing* that a test case consist of three elements:

- A test input or sequence of test inputs;
- An expected output or sequence of expected outputs; and
- The testing environment.

---

#### Footnotes

[1]: This is a failure in the sense of Section *The Language of Failures, Faults, and Errors*

---

The normal procedure for executing a test case is to execute the program using the inputs in the test case, record the results, and then to determine if the outputs obtained are failures[1] or not.

Who or what determines if the results produced by a program are failures? One way is for a human tester to look at the result of executing the test input and the expected results and decide if the program has failed the test case. In this case the human tester is playing the role of a *test oracle*.

A test oracle is someone or something that determines whether or not the program has passed or failed the test case. Of course, it can be another program that returns a “yes” if the actual results are not failures and “no” if they are.

---

#### Definition

A *test oracle* is:

- a program;
  - a process;
  - a body of data;
- 

that determines if actual output from a program has failed or not. ...

---

**Footnotes**

[2]: Automated means that we can execute the oracle with no human intervention.

---

Ideally an oracle should be automated [2] because then we can execute a larger volume of test cases and gain greater coverage of the program, **but** this is often extremely hard in practice.

## 6.3 Active and Passive Test Oracles

An automated oracle can be placed into one of two categories:

**Active oracle:** A program that, given an input for a program-under-test, can generate the expected output of that input.

**Passive oracle:** A program that, given an input for a program-under-test, and the actual output produced by that program-under-test, verifies whether the actual output is correct.

Passive oracles are generally preferred. This is for two main reasons.

Firstly, passive oracles are typically easier to implement than active oracles. For example, consider testing a program that sorts a list of numbers. It is considerably easier to check that an output produced by the is a sorted list, than it is to sort this list. This not only saves the tester some time, but also means that there is less chance of introducing a fault into the oracle itself. If the active oracle is required to simulate the entire program-under-test, it may be as difficult to implement, and therefore, is just as likely to contain faults of its own.

The second reason that passive oracles are preferred is because they can handle *non-determinism*. Recall from Section [Programs](#), that a program is non-deterministic if it can return more than one output for a single input. If an active oracle is used, there is a good chance that the output produced by the will be different to the output produced by the active oracle. However, if we use a passive oracle, which simply checks whether the output is correct, then non-determinism is not an issue.

## 6.4 Types of Test Oracle

In general, to design a test oracle, there are several types of oracle that can be produce, which are categorised by the way their are derived, or the way they run.

### 6.4.1 Formal, executable specifications

Formal specifications written in a tight mathematical notation are better for selecting and creating testing oracles than informal specification written in a natural language. They can be used as active oracles by generating expected output using a simulation tool, or as passive oracle, by proving that the specification is satisfied by the input and the actual output. These are unlikely in practice, and are mostly reserved by high integrity applications such as safety-, security-, or mission-critical systems.

### 6.4.2 Solved examples

Solved examples are developed by hand, or the results from a test input can be obtained from texts and other reference works. This is especially useful for complex domains, in which deriving the expected output automatically requires a process as complicated as the program itself, and deriving it manually requires expertise in a specific area that a test engineer is unlikely to have.

Data in the form of tables, documents, graphs or other recorded results are a good form of testing oracle. The test input and actual results can be looked up in a table or data-base to see if they are correct or not.

These types of oracles have the disadvantage that the inputs chosen are restricted to the examples that we have access to. Despite this, they are common due to their abundance in many fields.

### 6.4.3 Metamorphic oracles

In some cases, we can use certain metamorphic properties between tests to check each other. For example, if we want to test a function that sorts a list of numbers, then we can make use of the fact that, given a list, any permutation of that list will result in the same output of the sort function (assuming a non-stable sort).

To do metamorphic testing, we generate an input for a program, execute this input, and then generate another input whose output will be related to the first. In the sorting example, we can generate a test input as a list of numbers, and then randomly permute the elements of the list to get a new list. Then, we execute *both* tests on the sort function, and compare their output. If their output is different, then we have produced a failure.

Another example is a program for finding the shortest path between two nodes on a graph. We can select any two nodes on the graph and run the program, returning a path. To check if this path is correct, we can select any two nodes on that path and run the shortest path program on those two nodes. The resulting path should be a sub-path of the first path.

The existence of metamorphic properties for programs is surprisingly common, and metamorphic oracles have been used to test many numerical programs, but also many applications in non-numerical domains as well, including bioinformatics, search engines, machine learning, medical imaging, and web services.

### 6.4.4 Alternate implementations

An alternate implementation of a program, which can be executed to get the expected output. This is not ideal because experience has shown that faults in different implementations of the same specification tend to be located on the same inputs. Therefore, an alternate implementation is likely to have the same faults as the program-under-test, and some faults would not be detected via testing as a result.

One approach that has shown to be useful is to provide a *partial, simplified* version of the implementation, which does not implement the full behaviour of the program-under-test, and does not consider efficiency or memory.

For example, if we are an efficient sorting algorithm, then we can restrict the oracle (the alternative implementation) to only sorting lists of integers whose values, when sorted, form a complete integer sequence; e.g., 10, 11, 12, 13, 14, 15. To perform the sort, the oracle needs to only find the lowest and highest element in the list using a linear search, and then return a list with the lowest element at the first index, the highest element at the last, and the corresponding elements in

between. This is partial, and perhaps not efficient, but it results in a list that is a sorted version of the inputs, and is less likely to contain a fault than the original sorting algorithm due to its reduced complexity.

Such an approach restricts the test inputs that can be used, but is often sufficient to find many faults in a system.

### 6.4.5 Heuristic oracles

Perhaps the most widely-used types of automated oracles being used today are *heuristic oracles*. These are oracles that provide *approximate* results for inputs, and tests that “fail” must be scrutinised closely to check whether they are a true or false positive.

The trick with heuristic oracles is to find patterns in complex programs — that is, patterns between the inputs and outputs — and exploit them. For example, a very simple heuristic of databases is that, when a new record is inserted into a table, the number of records in that table should increase by 1. We can run thousands of tests inserting single records, and checking that the number of rows in table increases by 1. This is not complete though, because we are not checking that the contents of the row are accurate.

As a more realistic example, consider an oracle for checking a system that calculates driving directions for a GPS-enabled device. If the algorithm finds the shortest path between the start point and the destination, a complete oracle would need to check that this is indeed the shortest path. However, to check this fully, our oracle would have to re-calculate the shortest path as well, which would likely be as complicated as the original algorithm, and therefore just as prone to faults. Instead, we can use a heuristic that states that the shortest path should be within, e.g. 1.5 times the distance of a straight line between the two points. Anything outside of this could signal a fault. This is clearly not complete: the distance between two points may be small, while the shortest path via a road network may have to take a bridge that is far away from the destination.

In fact though, all oracles are heuristic, in that none of them really replicates the expected behaviour of the corresponding program. However, we use the term *heuristic oracle* to refer to oracles that are designed based on some heuristics about the software under test.

### 6.4.6 The Golden Program!

The ultimate source for a testing oracle but rare in practice. The golden program is an executable specification, a previous versions of the same program, or test in parallel with a trusted system that delivers the same functionality.

Still, the golden program is not a pipedream. In industry, it is not uncommon to use the previous release of a piece of software as a test oracle.

## 6.5 Oracle derivation

In these notes, we will not consider how to derive oracles. Current industry practice leaves much of the test case generation, including the oracle, up to human testers, who typically derive the oracles by looking at the specification and design of the artifact that are testing. In many unfortunate cases, the tester is left to guess what the behaviour of the software should be.

State-of-the-art testing includes *model-based testing*, which is used to automate both input and oracle generation. Using model-based testing, rather than the test engineer deriving test cases, he/she derives a *model* of the expected behaviour of the program-under-test, using some formal, executable language. From this model, test inputs are generated automatically (using the types of criteria that we discuss in these notes), and the expected outputs for those inputs are calculated by simulating the model. This can be seen as a cross between the first and last types of oracle. The model can be seen as an abstract alternative of the implementation, which is both formal and executable. However, unlike other alternate implementations, the higher level of abstract means that the likelihood of faults being located on the same inputs is



reduced. Empirical evidence demonstrates that model-based testing is, in general, no more expensive than manual test case generation, and in many cases, is significantly more efficient, and is as successful for locating faults.



## TESTING-AND-INTEGRATION

### 7.1 Learning outcomes of this chapter

At the end of this chapter, you should be able to:

- Map different integration strategies to test strategies.
- Describe the purpose of the different levels of testing: unit, integration, system, user acceptance, and regression.
- Compare and contrast exploratory testing to systematic forms of testing.

### 7.2 Chapter Introduction

---

#### Footnotes

Informally, latent faults are faults that remain undiscovered during development until after the system has been released and a specific state of the system or specific external circumstances trigger the fault.

---

In this chapter, we look at the process of building up systems from their component. The relationship between integration (or creating system *builds*) and testing is an important one. The order in which modules are integrated and tested can make the process of finding faults easier if a good strategy is chosen and may result in fewer latent faults [1].

The process of *developing* and *debugging* your programs, *building* your programs, *validating* and *verifying* your programs and *testing* your programs are inter-related and depend upon each other.

In general terms, a *software system* is a collection of programs. You may be developing a single (often referred to as a *monolithic*) program, or you may need to develop a *suite* of programs that must work together to achieve a common goal. We will use the word “system” to mean either a single program or a collection of programs.

The development of a system typically follows a process that involves requirements, design, implementation and testing phases. The process also typically uses various methods for validating and verifying the documents, plans, programs and other artifacts that are developed in each phase. This is where the various life-cycles and techniques from the Software Processes and Management subject are used.

## 7.3 Integrating the System

In the process of designing a system, you will have decomposed your system into modules, packages or classes. If the system is complicated then you may have decomposed each module into sub-modules and so on.

*Integrating* your system refers to putting the modules, packages or classes together to create subsystems, and eventually the system itself. Generally speaking, integration and testing go together. Good integration strategies can give help you to minimise testing effort required. Poor integration strategies generally cause more work, and lead to lower quality software.

Normally, a system integration is organised into a series of *builds*. Each build combines and tests one subset of the modules, packages or objects of the system.

The aim for most integration strategies is to divide the modules into subsets based on their *dependencies*. Typical build strategies then try and integrate all those modules with no dependencies first, the modules that depend on these second, and so on. Here are some examples of integration strategies.

### 7.3.1 The Big-bang Integration Strategy

The *Big-Bang* method involves coding all of the modules in a sub-system, or indeed the whole system, separately and then combining them all at once. Each module is typically unit tested first, before integrating it into the system. The modules can be implemented in any order so programmers can work in parallel.

Once integrated, the completely integrated system is tested. The problem is that if the integrated systems fails a test then it is often difficult to determine which module or interface caused the failure. There is also a huge load on resources when modules are combined (machine demand and personnel).

The other problem with a Big-Bang integration strategy is that the number of input combinations becomes extremely large — testing all of the possible input combinations is often impossible and consequently there may be latent faults in the integrated sub-system

### 7.3.2 A Top-Down Integration Strategy

The *top-down* method involves implementing and integrating the modules on which no other modules depend, first. The modules on which these *top-level* modules depend are implemented and integrated next and so on until the whole system is complete.

The advantages over a Big-Bang approach are that the machine demand is spread throughout integration phase. Also, if a module fails a test then it is easier to isolate the faults leading to those failures. Because testing is done incrementally, it is more straightforward to explore the input for program faults.

The top-down integration strategy requires *stubs* to be written. A stub is an implementation used to stand in for some other program. A stub may simulate the behaviour of an existing implementation, or be a temporary substitute for a yet-to-be-developed implementation.

### 7.3.3 The Bottom-Up Integration Strategy

The *bottom-up* method is essentially the opposite of the top-down. Lowest-level modules are implemented first, then modules at the next level up are implemented forming subsystems and so on until the whole system is complete and integrated.

This is a common method for integration of object-oriented programs, starting with the testing and integration of base classes, and then integrating the classes that depend on the base classes and so on.

The bottom-up integration strategy requires *drivers* to be written. A driver is a piece of code used to supply input data to another piece of code. Typically, the piece of modules being tested need to have input data supplied to them via their interfaces. The driver program typically calls the modules under test supplying the input data for the tests as it does so.

### 7.3.4 A Threads-Based or Iterative Integration Strategy

The *threads-based* integration method attempts to gain all of the advantages of the top-down and bottom-up methods while avoiding their weaknesses. The idea is to select a minimal set of modules that perform a program function or program capability, called a **thread**, and then integrate and test this set of modules using either the top-down or bottom-up strategy.

Ideally, a thread should include some I/O and some processing where the modules are selected from all levels of the module hierarchy. Once a thread is tested and built other modules can be added to start building up a complete the system.

This idea is common in agile methodologies. Threads are focussed on user requirements, or user stories. Given a coherent piece of functionality from a set of user requirements, we build that small piece of functionality to deliver some value to the user. However, this is NOT an integration strategy on its own: we will still have to bring together parts of this as we go, using top-down and bottom-up strategies.

## 7.4 Types and Levels of Testing

### Footnotes

[2] The diagram in Figure 7.1 is called the V Model. The V Model is not (necessarily) a process model. Instead, it is used to demonstrate that you design your system tests against requirements specifications, integration tests against high level design specifications and unit tests against detailed design specifications.

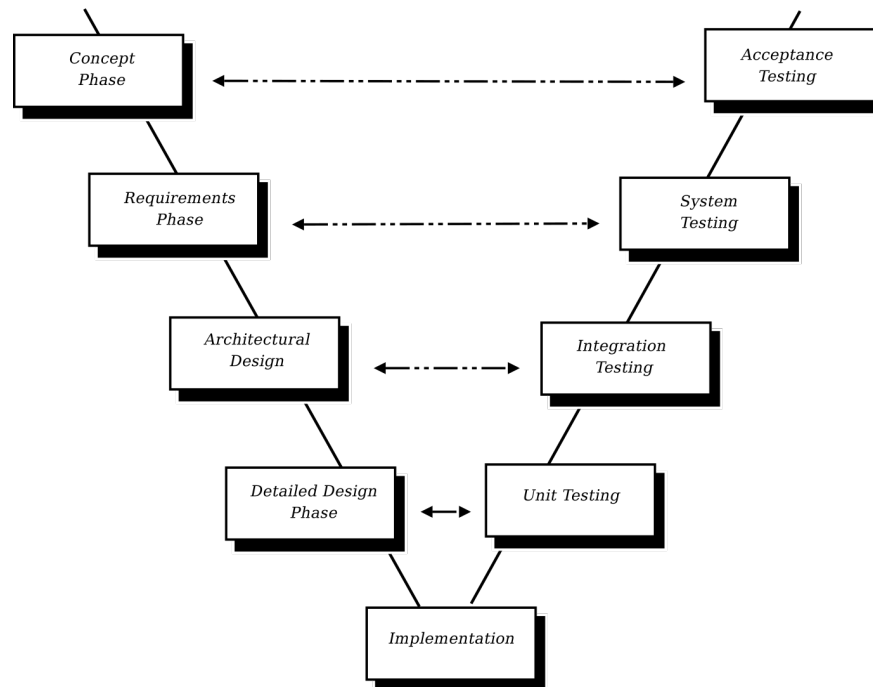
There are various types of testing that systems typically undergo. Each type of testing is aimed at detecting different kinds of failures and making different kinds of measurements. Figure 7.1 is a representative diagram that shows how each type of testing corresponds to a stage in the development<sup>[2]</sup>.

### 7.4.1 Unit Testing

The first task in unit testing is to work out exactly what a unit is for the purpose of testing. In most cases, a unit is a module: a collection of procedures or functions that manipulate a shared state. In these cases, the module is tested as a whole, because changes in one procedure can effect others. In other cases, a unit is a single procedure or function. In these cases, the only inputs and outputs are parameters and return values — there is no state.

Units, if they are functions or classes, are tested for correctness, completeness and consistency. Unit testing measures the attributes of units. They can be tested for performance but almost never for reliability; units are far too small in size to have the statistical properties required for reliability modelling.

If each unit is thoroughly tested before integration there will be far fewer faults to track down later when they are harder to find.



### 7.4.2 Integration Testing

Integration testing tests collections of modules working together. Which collection of modules are integrated and tested depends on the integration strategy. The aim is to test the interfaces between modules to confirm that the modules interface together properly. Integration testing also aims to test that subsystems meet their requirements.

Integration tests are derived from high level component designs and requirements specifications.

### 7.4.3 System Testing

System testing test the entire system against the requirements, use cases or scenarios from requirements specification and design goals. Sometimes system testing is broken down into three groups of related testing activities:

---

#### Definition

- **System Functional Test:** tests the entire system against the functional requirements and other external sources that determine requirements such as the user manual.
  - **System Performance Test:** test the non-functional requirements of the system for example, the load that the system can handle, response times and can test usability (although this latter testing is more like a survey than what we would call an actual *test*).
  - **User Acceptance Test:** is a set of tests that the software must pass before it is accepted by the clients. This is typically a form of *validation*, whereas testing against the specification is a form of *verification*.
-

### 7.4.4 Regression testing

Programs undergo changes but the changes do not always effect the entire program. Rather certain functions or modules are changed to reflect new requirements, system evolution or even just bug fixes. History shows that, after making a change to a system, testing only the part of the system that has changed is not enough. A change in one part of a system can effect other parts of the system in ways that are difficult to predict. Therefore, after any change, the entire test suite of a system must be run again. This is called *regression testing*.

Regression testing is one of the key reasons for wanting to be able to execute test suites automatically. To test an entire system by hand is costly even for the smallest of systems, and infeasible for many others.

### 7.4.5 Exploratory testing

*“Exploratory testing is simultaneous learning, test design, and test > execution.”* – James Bach, Exploratory Testing Explained.

Exploratory testing is a form of *unstructured* testing. It is done typically by experienced testers, preferably with some domain knowledge about the application. All they do is explore the software, using their experience from years of software engineering and what they have learnt about the piece of software during their exploratory testing to find faults.

It is typically conducted at a system level (but not always), simply by having people sit and explore the application, trying out different parts, observing what happens, and then repeating. It aims to exploit human insight to find faults. By exploring the application, the human tester gains insight that they could not gain by just writing tests.

Exploratory testing has several advantages over other types of structured testing. First, the knowledge gained by the tester during the testing process can lead to them understanding the requirements better than someone doing structure testing, so they may think of things that they wouldn't have otherwise. Second, structured tests require both a test input and an expected output. So, the tester will be focused on the test output, and may completely ignore other things that are going on. Consider a web application, in which a tester executes a test, and looks at a specific field to get the actual output to compare to the expected output. They are less likely to notice other anomalies on the page compared to an exploratory tester.

It also have disadvantages. First, it does not really offer any type of structure around the testing, such as how to maximise chances of finding faults. Second, it is difficult to repeat, and difficult to run regression tests, because we do not really record what is happening.

Exploratory testing is best used in combination with structured techniques. Although it is less scientific in the way test selection and execution is done, its advantage is the use of human insight to find faults.





## SECURITY TESTING

### 8.1 Learning outcomes of this chapter

At the end of this chapter, you should be able to:

- Discuss the purpose of security testing.
- Explain the potential security implications (in terms of integrity, confidentiality, availability, arbitrary code execution etc.) of security faults
- Explain the relative strengths and weaknesses of random fuzzing vs mutation based fuzzing vs generation based fuzzing
- Choose (and justify) which security testing technique (random fuzzing, mutation based fuzzing, generation based fuzzing, or some combination) is most appropriate for testing a particular software component
- Explain the security implications of undefined behaviour in C code
- Discuss the advantages and disadvantages of using a memory debugger to check for memory errors, and undefined behaviour sanitizers to check for undefined behaviour
- Explain in general terms how general-purpose greybox fuzzers like AFL or libFuzzer operate and their advantages and disadvantages as compared to random, mutation and generation-based fuzzing.

### 8.2 Chapter Introduction

In this chapter, we will look at a few systematic and automated approaches to testing security of systems. Specifically, we will look at three different ways of *penetration testing*.

Penetration testing (or *pentesting*) is the process of attacking a piece of software with the purpose of finding security vulnerabilities. In practice, pentesting is just hacking, with the difference being that in pentesting, you have the permission of the system owners to attack the system.

In other words, penetration testing is *hacking with permission*.

One way to perform penetration testing is to just try out a whole load of tricks that have worked before. For experienced pentesters, this is an excellent and fruitful approach; and in fact, many people make their career out of intelligently trying to break systems. However, this is exceedingly difficult to teach in a few week period off a university subject. Instead, in this chapter, we are going to focus on *automated* pentesting, which is generally called *fuzzing* or *fuzz testing*, which is a systematic and repeatable approach to pentesting, and one which is used by many great penetration testers.

## 8.3 Introduction to Penetration Testing

First, some terminology. The aim of penetration testing is to find *vulnerabilities*. A vulnerability is a security hole in the hardware or software (including operating system) of a system that provides the possibility to attack that system; e.g., gaining unauthorised access. Vulnerabilities can be weak passwords that are easy to guess, buffer overflows that permit access to memory outside of the running process, or unescaped SQL commands that provide unauthorised access to data in a database.

---

### Example 66

As an example of an SQL vulnerability, consider a request that allows a user to lookup their information based on their email address via a textbox, resulting in the following query to a database:

```
SELECT * FROM really_personal_details WHERE email = '$EMAIL_ADDRESS'
```

in which \$EMAIL\_ADDRESS is the text copied from the text box.

If the text myemail@somedomain.com is entered, then this works fine. However, if the user is aware of such a vulnerability, they may instead enter: myemail@somedomain.com' OR '1=1 (note the unopened and unclosed quotation marks here). If the developers have not been careful, this *may* result in the following query being issued:

```
SELECT * FROM really_personal_details WHERE email = 'myemail@somedomain.com' OR '1=1'
```

The WHERE clause evaluates to *true*, and therefore, the user will be able to gain unauthorised access to all data in the table, including that of other users.

---

---

#### Footnotes

[1]: Taken from [https://en.wikipedia.org/wiki/Stack\\_buffer\\_overflow](https://en.wikipedia.org/wiki/Stack_buffer_overflow)

---

---

#### Footnotes

[2]: Taken from [https://en.wikipedia.org/wiki/Stack\\_buffer\\_overflow](https://en.wikipedia.org/wiki/Stack_buffer_overflow)

---

Buffer overflows are another common security vulnerability. These occur when data is written into a buffer (in memory) that is too small to handle the size of the data. In some languages, such as C and C++, the additional data simply overwrites the memory that is located immediately after the buffer. If carefully planned, attacker-generated data and code can be written here.

---

### Example 67

Consider the following example of a stack buffer overflow[1].

The following C program takes a string as input, and allocates it to a buffer 12 characters long:

```

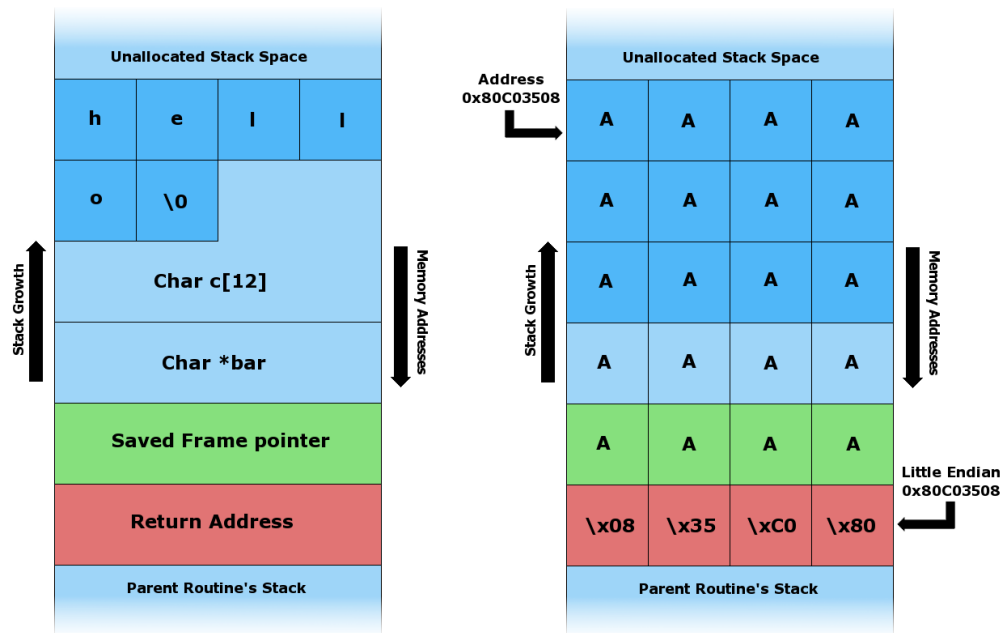
void foo (char *bar)
{
    char c[12];

    strcpy(c, bar); // no bounds checking
}

int main (int argc, char **argv)
{
    foo(argv[1]);
}

```

Now, consider the two cases in Figure 9.1[2]. On the left is the case where the input string, `hello\0`, is small enough to fit into the buffer. The frame pointer and return address for the `foo` function sit next to the buffer in memory. On the right side is an example where the input string, `AA... \x08\x35\xC0\x80`, is longer than the buffer. This writes over the frame pointer and the return address. Now, when `foo` finishes executing, the execution will jump to the address specified in the return address spot, which is at the start of `char c[12]`. In this case, the contents of `char c[12]` is meaningless, but in an attack, the string would contain payload that could be a malicious program, which would have privileges equivalent to the program being executed; e.g., it could have superuser/admin access.



Clearly, there are many ways to try to reduce the chance such attacks: defensive programming that checks for array bounds, using programming languages that do this automatically, or finding and eliminating these problems. We can find these problems using reviews and inspections, letting hackers find them for us (not recommended), or using verification techniques.

In this chapter, we look at verification techniques, specifically, fuzz testing.

### Definition

*Fuzz testing* (or *fuzzing*) is a (semi-)automated approach for penetration testing that involves the randomisation of input data to locate vulnerabilities.

Typically, a fuzz testing tool (or *fuzzer*) generates many test inputs and monitors the program behaviour on these inputs, looking for things such as exceptions, segmentation faults, and memory leaks; rather than testing for functional correctness. Typically, this is done live: that is, one input is generated, executed, and monitored, then the next input, and so on.

We will look at three techniques for fuzzing:

1. Random testing: tests generated randomly from a specified distribution.
2. Mutation-based fuzzing: starting with a well-formed input and randomly modifying (mutating) parts of that input
3. Generation-based fuzzing: using some specification of the input data, such as a grammar of the input.

## 8.4 Random Testing

Random testing is one approach to fuzzing. In random fuzzing, tests are chosen according to some probability distribution (possibly uniform) to permit a large amount of inputs to be generated in a fast and unbiased way.

### 8.4.1 Advantages

- It is often (but not always) cheap and easy to generate random tests, and cheap to run many such tests automatically.
- It is *unbiased*, unlike tests selected by humans. This is useful for pentesting because the cases that are missed during programming are often due to lack of human understanding, and random testing may search these out.

### 8.4.2 Disadvantages

- A prohibitively large number of test inputs may need to be generated in order to be confident that the input domain has been adequately covered.
- The distribution of random inputs simply misses the program faults (recall the Pareto-Zipf principle from Chapter 1).
- It is highly unlikely to achieve good coverage.

The final point deserves some discussion. If we take any non-trivial program and blast it with millions of random tests, it is unlikely that we will achieve good coverage, where coverage could be based on any reasonable criteria (control-flow, mutation, etc.).

Consider the following example, in which we have a “fault” at line 7 where the program unexpectedly aborts.

```
1. void good_bad(char s[4]) {  
2.     int count = 0;  
3.     if (s[0] == 'b') count++;  
4.     if (s[1] == 'a') count++;  
5.     if (s[2] == 'd') count++;  
6.     if (s[3] == '!') count++;  
7.     if (count >= 3) abort(); //fault  
8. }
```

This program will only fail on the input “bad!”. Using random testing, random arrays of four characters will be generated, but encountering the fault at line 7 is highly unlikely. Since each char data type (in C programming language) occupies 8 bit in the memory, the probability of randomly generating the string “bad!” requires us to randomly choose ‘b’ at index 0 (probability of 1 in  $2^8$ , ‘a’ at location 1 (probability of 1 in  $2^8$ ), etc., leaving us with the probability of 1 in  $2^{32}$  that the code will be executed. In fact, almost every randomly generated test will execute false for every branch.

While this example is a fabricated example, such cases are common place in real programs. For example, consider any branch of the form  $x == y$  (unlikely that  $x$  and  $y$  will have the same value). Or, more interestingly, consider the chance of randomly generating a correct username and password to get into a system, or generating the correct checksum for a packet of data that is received. This latter case is common in security-critical applications: only if we have the correct checksum, we can “do something interesting”:

```

1. boolean my_program(char [] data, int checksum)
2. {
3.     if (calc_checksum(data) == checksum) {
4.         //do something interesting
5.         return true;
6.     }
7.     else {
8.         return false;
9.     }
10. }
```

In cases such as this, a random testing tool will likely spend most of its time just testing the false case over and over, which is not particularly useful.

A standard way to address this is to (if possible) measure the code coverage achieved by your tests after a certain amount of time, and look for cases like these. Then, modify your random testing tool to first send some data with a correct checksum, and then use random testing for the remainder of inputs. Rinse and repeat.

## 8.5 Mutation-based Fuzzing

Mutation-based fuzzing is a simple process that takes valid test inputs, and mutates small parts of the input, generating (possibly invalid) test inputs. The mutation (not to be confused with mutation analysis discussed in Section *Mutation Analysis*) can be either random or based on some heuristics.

### Footnotes

[3]: Taken from <http://pages.cs.wisc.edu/~rist/642-fall-2012/toorcon.pdf>

As an example, consider a mutation-based fuzzer for testing web servers[3]. A standard HTTP GET request for the index page could be a valid input:

```
GET /index.html HTTP/1.1
```

From this, many anomalous test inputs, which should all be handled by the server, can be generated by randomly changing parts of that input:

```

AAAAAA...AAAA /index.html HTTP/1.1
GET //////////index.html HTTP/1.1
GET %n%n%n%n%n%n.html HTTP/1.1
GET /AAAAAAAAAAAAA.html HTTP/1.1
GET /index.html HTTTTTTTTTTTTTP/1.1
GET /index.html HTTP/1.1.1.1.1.1.1.1.1
```

One can imagine a simple program being set up to generate such inputs and deliver them to a web server.

As an example of a more heuristic-based mutation fuzzer, consider a text field in a web form in which a user enters their email address, as in Example 66. Attacks such as those presented are common, so an heuristic mutation fuzzer will add targets such as that to fields, instead of random data. So, a valid input such as:

```
'myemail@somedomain.com'
```

could be mutated to things such as:

```
'myemail@somedomain.com' OR '1=1'  
'myemail@somedomain.com' AND email IS NULL  
'myemail@somedomain.com' AND username IS NULL  
'myemail@somedomain.com' AND userID IS NULL
```

These last three attempt to guess the name of the field for the email address. If any of these give a valid response, we know that we guess the name of the field correctly; otherwise a server error will be thrown.

### 8.5.1 Advantages

The main advantages of fuzzing are compared to random testing are:

- It generally achieves higher code coverage than random testing. While issues such as the checksum issue discussed earlier still occur, they often occur less of the time if the valid inputs that are mutated have the correct values to get passed these tricky branches. Even though the mutated tests may change these, some will change different parts of the input, and the e.g., checksum will still be valid.

### 8.5.2 Disadvantages

- The success is highly dependent on the valid inputs that are mutated.
- It still suffers from low code coverage due to unlikely cases (but not to the extent of random testing).

### 8.5.3 Tool support

There are many tools to support mutation-based fuzzing.

- Radamsa (<https://gitlab.com/akihe/radamsa>) is typically used to test how well a program can withstand malformed and potentially malicious inputs. It works by reading sample files of valid data and generating interestingly different outputs from them.
- zzuf (<http://caca.zoy.org/wiki/zzuf>) is a commonly-used tool for “corrupting” valid input data to produce new anomaly tests. It uses randomisation, and has controllable properties such as how much of the input should be changed for each test.
- Peach (<http://www.peachfuzzer.com/>) is a well-known fuzzer that supports mutation fuzzing, and has reached a level of maturity that make it applicable to many projects.

**Part II**

**APPENDIX**





## A BRIEF REVIEW OF SOME PROBABILITY DEFINITIONS

Probability theory is concerned with *chance*. Whenever there is an event or an activity where the outcome is uncertain, then probabilities are involved. We normally think of any activity or event with an uncertain outcome as an *experiment* whose outcome we observe. Probabilities are then measures of the likelihood of any of the possible outcomes.

An *experiment* represents an activity whose output is subject to chance (or variation). The output of the experiment is referred to as the *outcome* of the experiment. The set of all possible outcomes is called the *sample space*.

---

### Definition

The *sample space* for an experiment is the set of all possible outcomes that might be observed.

---

---

### Some examples of experiments and their outcomes

- An experiment involving the flipping of a coin has two possible outcomes Heads or Tails. The sample space is thus  $S = \{\text{Heads, Tails}\}$ .
  - An experiment involving testing a software system and counting the number of failures experienced after  $T = 1$  hour has many possible outcomes: we may experience no failures, 1 failure, 2 failures, 3 failures, .... The sample space is thus  $N = \{0, 1, 2, 3, \dots\}$ .
  - An experiment involving the testing of a software system and recording the time at which the first failure occurs has a real valued sample space  $R$ , although in practice this is more likely to be a time interval.
- 

Probabilities are usually assigned to events.

---

### Definition

Let  $S$  be a sample space. An *event* is a subset  $A$  of the sample space  $S$ , that is,  $A \subset S$ . An event is said to have *occurred* if any one of its elements is the outcome observed in an experiment.

---

The probability of an event  $A$  is a non-negative real number that relates to the number of times we observe an outcome in  $A$ . Often this real number is just the fraction of times that we observe an outcome in  $A$  over the total number of possible outcomes. We write:

$$P\{A\} = \lim \frac{m}{n}$$

where  $m$  is the number of outcomes in  $A$  and  $n$  is the total number of possible outcomes, that is, the number of elements in  $S$ . For two events,  $A$  and  $B$ ,  $A \cup B$  to refer to event  $A$  occurring or event  $B$  occurring, or both, and,  $A \cap B$  to refer to event  $A$  and event  $B$  both occurring. Therefore  $P\{A \cap B\}$  is the probability of events  $A$  and  $B$  both occurring. The

*conditional probability* of an event  $A$  is the probability of  $A$  given that  $B$  has occurred. This is written  $P\{A|B\}$ . In conditional probabilities  $B$  is often called the *conditioning event*.

Probabilities must satisfy certain *probability laws* in order to be meaningful measures of likelihood. The following probability laws hold for any event  $A$  and any state space  $S$ :

- $0 \leq P\{A\} \leq 1$ ; and
- $P\{\neg A\} = 1 - P\{A\}$
- $P(S) = 1$ .

That is: (i) the probability of an event occurring is between 0 and 1 inclusive; (ii) the probability of  $A$  not occurring is 1 minus the probability that it will; and (iii) the sum of the probabilities of all events in the state space is 1, and no events outside the sample space can occur.

Two events are said to be *independent* if the occurrence of one event does not depend on the other and the converse. For example, if we have two dice, the probability of one falling on the outcome 6 is independent of the outcome of the other dice. However, if we throw both dice, but one falls on the floor out of sight, while the other shows a 3, then the probability of their total equaling 7 is dependent on the probability of the second dice.

If events  $A$  and  $B$  are *dependent*, then the following laws hold:

$P\{A \cap B\} = P\{A|B\}P\{B\}$  *Multiplicative Law*

$$P\{A|B\} = \frac{P\{A \cap B\}}{P\{B\}} \text{ Conditional Probability}$$

If events  $A$  and  $B$  are *independent* then the following laws hold:

$P\{A \cap B\} = P\{A\}P\{B\}$  *Multiplicative Law*

$P\{A|B\} = P\{A\}$  *Conditional Probability*

The independent and dependent cases are related. For example, if  $A$  and  $B$  are independent, then  $P\{A \cap B\} = P\{A\}P\{B\}$ , and therefore  $P\{A|B\}$  is equal to  $(P\{A\}P\{B\})/P\{B\}$  (the first conditional probability law), which is clearly  $P\{A\}$  (the second conditional probability law).

Two events are said to be *mutually exclusive* if they cannot both occur. For example, if we throw a single die, then it is not possible that both 1 and 2 will result. If events  $A$  and  $B$  are not mutually exclusive, then the following law holds:

$P\{A \cup B\} = P\{A\} + P\{B\} - P\{A \cap B\}$  *Additive Law*

If events  $A$  and  $B$  are mutually exclusive, then the following laws hold:

$P\{A \cup B\} = P\{A\} + P\{B\}$  *Additive Law*

$P\{A \cap B\} = 0$  *Mutual Exclusion*

These laws are also related. If  $A$  and  $B$  are mutually exclusive, then  $P\{A \cap B\}$ , and therefore  $P\{A\} + P\{B\} - P\{A \cap B\}$  (the first additive law) is  $P\{A\} + P\{B\} - 0$ , which is equivalent to the second additive law.

## 9.1 Random Variables

Now, suppose that we can represent each element of the sample space by a number. If we perform an experiment then for each element  $E$  of the sample space  $S$ , there is a certain probability that we will observe  $E$  as the outcome. A *random variable* assigns a number to each outcome in the sample space. Random variables that we will encounter in this subject are:

1. Number of failures at time  $T$ , which is *discrete* a integer number. The sample space is  $T$  and the number that we assign to it as the random variable is the number of failures.

2. Time to first failure, which is a *continuous* real-valued number. The sample space is again  $T$  and the random variable in this case is the time  $\tau$  to the first failure observed.

As an example, consider an experiment in which we execute random test cases on a piece of software for a period of 1 hour and observe the number of failures.

1. The sample space that we will consider (and its not the only one possible either) is  $N = \{0, 1, 2, 3, \dots\}$ .
2. Let  $X$  be a random variable that returns the number of failures experienced after 1 hour of testing (assuming we can suitably quantify failure).
3. Then we can ask questions such as what is the probability that there are fewer than  $N$  failures after 1 hour – written  $P\{X < N\}$  – or what is the probability that we will have no failures after 1 hour –  $P\{X = 0\}$ . The random variable which we have called  $X$  is the number of failures experienced in 1 hour.

## 9.2 Probability Density Functions

Normally associated with each random variable is a *probability density function*, or sometimes a *probability law*, that assigns a probability to every outcome in the random variable's sample space. A random variable can be discrete or continuous.

- The sample space of a discrete random variable takes on specific values at discrete points  $\{a_1, \dots, a_k\}$ .
- A continuous random variable takes on all values in the real line or an interval of the real line.

We think of the probability density function as a function  $f$  that maps each value of the sample space  $S$  to  $[0, 1]$  so that  $f : S \rightarrow [0, 1]$ . For example, the probability density function of throwing a die is  $f(o) = \frac{1}{6}$  for all outcomes  $o$ .

The two key properties that we require of probability density functions are as follows.

For Discrete Random Variables: we require that

- $(f(X) \geq 0) \text{ for all } (x \in S)$ ; and
- $(\sum_{x \in S} f(x) = 1)$ .

For Continuous Random Variables: we require that

- $(f(X) \geq 0) \text{ for all } (x \in S)$ ; and
- $(\int_{-\infty}^{+\infty} f(x) = 1)$ .

The cumulative density function for a discrete random variable  $X$  is defined as

$$F_X(T) = \sum_{X \leq T} f(X)$$

that is, a sum of all of the probabilities for outcomes less than or equal to  $T$ . If we wish to calculate the probability  $P\{A < X \leq B\}$  then this is simply

$$P\{A < X \leq B\} = F_X(B) - F_X(A).$$

For a continuous random variable  $X$  the cumulative density function is defined as

$$F_X(T) = P\{X \leq A\} = \int_{-\infty}^A f(X)$$

## 9.3 Probability in Reliability Measurement

Random variables and their distributions will be important to us because many of our reliability measures are expressed in terms of random variables and their distributions. Some examples are:

1. Let  $T$  be a random variable representing the time of failure of a particular system. Then the *failure probability* is expressed as a cumulative density function:

$$F(t) = P\{T \leq t\}.$$

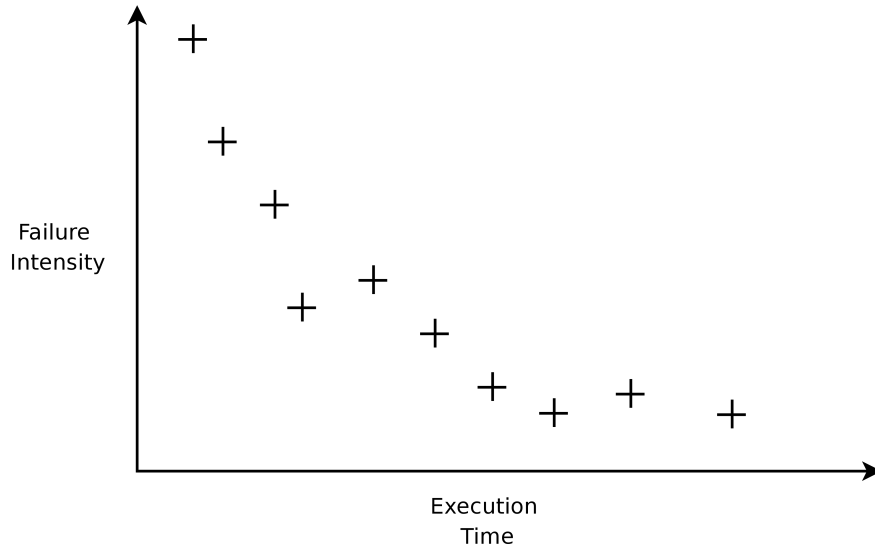
1. Reliability is simply the converse – it is the probability that a particular systems survives — that is, does not experience a failure — until after time  $t$ :

$$R(t) = 1 - F(t) = P\{T > t\}.$$

## MAXIMUM LIKELIHOOD ESTIMATION

Maximum likelihood estimation is by far the better technique for estimating model parameters. Unfortunately it also often requires numerical solutions to solve sets of equations for those very parameters.

Intuitively the maximum likelihood estimator tries to pick values for the parameters of the basic execution time model that maximise the probability that we get the observed data. For example suppose that we had the failure intensity data shown in Figure B.1.



Then we start exploring the parameter space consisting of pairs of values  $(\lambda_0, \nu_0)$  for values that will maximise the likelihood of obtaining the observed values. When we start exploring the values of  $\lambda_0$  and  $\nu_0$  we may find that the curves like something like those in Figure B.2

To estimate the parameters, maximum likelihood now works as follows. Suppose that we have only one parameter  $\theta$  instead of the two parameters in the Basic Execution time model. Now, if we make  $n$  observations  $x_1, x_2, \dots, x_n$  of the failure intensities for our program the probabilities are:

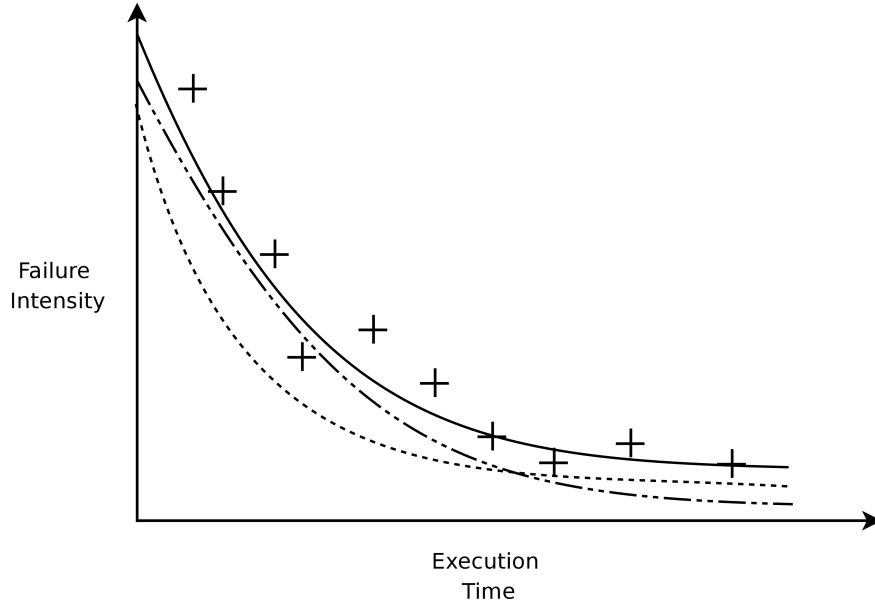
$$L(\theta) = P\{X(t_1) = x_1\}P\{X(t_2) = x_2\} \dots P\{X(t_n) = x_n\}$$

To function  $L(\theta)$  reaches its maximum when the derivative is 0, that is,

$$\frac{dL(\theta)}{d\theta} = 0$$

For example, if we have an exponential probability law  $\theta e^{-\theta T}$  with a parameter  $\theta$  and we make  $n$  observations  $x_1 \dots x_n$  at times  $t_1 \dots t_n$  then from the exponential distribution,  $L(\theta)$  becomes

$$\theta e^{-\theta t_1} \theta e^{-\theta t_2} \dots \theta e^{-\theta t_n} = (\theta)^n e^{-\theta \sum t_i}$$



An estimate for the parameter is then value of  $\theta$  making

$$\frac{d(\theta)^n e^{-\theta \sum t_i}}{d\theta} = 0$$

which gives a maximum. In general, when the exponential function  $e$  is involved take the natural log of  $L(\theta)$  and take the derivative. Doing this gives the same value for  $\theta$  as the derivative in. In the case of  $L(\theta)$  we get

$$\frac{d \ln(\theta)^n e^{-\theta \sum t_i}}{d\theta} = \frac{n}{\theta} - \sum t_i = 0$$

and we can easily solve for  $\theta$  to get  $\theta = \frac{n}{\sum t_i}$ .

The situation with the basic execution time model is not so easy because we have two parameters. To simplify the procedure let  $\beta = \frac{\lambda_0}{\nu_0}$  and let  $X$  be our random variable whose probability density function is  $\lambda_0 e^{-\beta_1 x}$ . Next, assume that we have been testing for an execution time period of  $T_F$  seconds and have experienced a total of  $M_F$  failures to this point. If we repeat the idea above and take *partial derivatives*:

$$\frac{\partial L(\lambda_0, \beta_1)}{\partial \lambda_0 x} = 0 \quad \frac{\partial L(\lambda_0, \beta_1)}{\partial \beta_1} = 0$$

we again arrive at our maximum. We again need to take the natural logarithms of both sides to get

$$\frac{\partial \ln L(\lambda_0, \beta_1)}{\partial \lambda_0 x} = 0 \quad \frac{\partial \ln L(\lambda_0, \beta_1)}{\partial \beta_1} = 0.$$

The resulting equations that need to be solved often require numerical methods that are outside of the scope of these notes. For completeness, the two estimators are included below.

$$\frac{M_F T_F}{\sum_{i=1}^{M_F} t_i + T_F(\nu_0 - i + 1)} + \sum_{i=1}^{M_F} \frac{1}{\nu_0 - i + 1} = 0$$

$$\frac{M_F}{\beta_1} - \frac{M_F T_F}{e^{\beta_1 T_F} - 1} - \sum_{i=1}^{M_F} t_i = 0$$

**Part III**

**TUTORIALS**





## SWEN90006 TUTORIAL 1

### 11.1 What is testing really?

*It works! Trust me, I've tested the code thoroughly!*

With those bold words many software projects have been made bankrupt.

The aim of this tutorial is to start to get an understanding of our major themes; that is, of the factors that make up quality and how testing effects them.

While we have not yet looked at testing formally yet you will be asked in this tutorial to test a small program fragment with the purpose of starting to see where the difficulties of testing lay in practice.

### 11.2 Important terminology

Before beginning the exercises, consider the following three definitions, the first three of which are defined in the chapter [Introduction to software testing](#) of the subject notes:

- *Fault*: An incorrect step, process, or data definition in a computer program. Faults are the source of failures – fault in the program triggers a failure under the right circumstances.
- *Failure*: A deviation between the observed behaviour of a program, or a system, from its specification. Faults cause failures; and in fact, one fault can cause many failures.
- *Error*: An incorrect *internal* state that is the result of some fault. An error may not result in a failure – it is possible that an internal state is incorrect but that it does not affect the output.

*Faults cause failures and errors.* A fault in a program can trigger a failure and/or an error under the right circumstances. In fact, a single fault could cause multiple failures and multiple errors.

In normal language, software faults are usually referred to as “bugs”, but the term “bug” is ambiguous and can mean to faults, failures, or errors; as such, as will avoid this term.

## 11.3 Your tasks

The small programs below are taken from the exercises in Chapter 1 of *Introduction to Software Testing* by Offutt and Ammann. For each program, the test case below the program results in a failure that is caused by a fault in the program.

For each of the programs below, perform the following tasks:

1. Specify the input domain of the function.
2. Identify the fault.
3. If possible, identify a test case that does not execute the faulty statement.
4. If possible, identify a test case that executes the faulty statement, but does not result in an failure.
5. If possible, identify a test case that forces the program into an error, but does not produce a failure.
6. Fix the fault and verify that the given test now produces the expected output.

## 11.4 The programs

```
'''
    x is a list of integers
    y is an integer
    Return the index of the last element in x that equals y.
    If no such element exists, return -1
'''
def find_last (x, y):
    i = len(x) - 1
    while i > 0:
        if x[i] == y:
            return i
        i -= 1
    return -1

# The expected output is 0 (the 0th element)
x = [2, 3, 5]
y = 2
assert find_last(x, y) == 0, "Test failed: find_last(%s, %d) == %d" % (x, y, find_
↳last(x, y))
```

```
-----
AssertionError                                Traceback (most recent call last)
<ipython-input-1-2414f9a61719> in <module>
      16 x = [2, 3, 5]
      17 y = 2
--> 18 assert find_last(x, y) == 0, "Test failed: find_last(%s, %d) == %d" % (x, y,
↳find_last(x, y))

AssertionError: Test failed: find_last([2, 3, 5], 2) == -1
```

```
'''
    x is a list of integers
    Return the index of the LAST 0 in x.
    Return -1 if 0 does not occur in x.
'''
```

(continues on next page)

(continued from previous page)

```
def last_zero(x):
    i = 0
    while i < len(x):
        if x[i] == 0:
            return i
        i += 1
    return -1

# The expected output is 2
x = [0, 1, 0]
assert last_zero(x) == 2, "Test failed: last_zero(%s) == %d" % (x, last_zero(x))
```

```
-----
AssertionError                                Traceback (most recent call last)
<ipython-input-2-afe149f20270> in <module>
     14 # The expected output is 2
     15 x = [0, 1, 0]
--> 16 assert last_zero(x) == 2, "Test failed: last_zero(%s) == %d" % (x, last_
    ↪ zero(x))

AssertionError: Test failed: last_zero([0, 1, 0]) == 0
```

```
'''
    x is a list of integers
    Return the number of _strictly_ positive elements in x.
'''
def count_positive(x):
    count = 0
    i = 0
    while i < len(x):
        if x[i] >= 0:
            count += 1
        i += 1
    return count

# The expected output is 2
x = [-4, 2, 0, 2]
assert count_positive(x) == 2, "Test failed: count_positive(%s) == %d" % (x, count_
    ↪ positive(x))
```

```
-----
AssertionError                                Traceback (most recent call last)
<ipython-input-3-1980abda4298> in <module>
     14 # The expected output is 2
     15 x = [-4, 2, 0, 2]
--> 16 assert count_positive(x) == 2, "Test failed: count_positive(%s) == %d" % (x,
    ↪ count_positive(x))

AssertionError: Test failed: count_positive([-4, 2, 0, 2]) == 3
```



## SWEN90006 TUTORIAL 2

### 12.1 Introduction

The aim of this tutorial is twofold. First, it aims to give you some practise at deriving test cases from specifications. Second, it aims for you to start exploring the limits of your test cases, and of the specifications.

### 12.2 The Program

**Input File:** The input file format is as follows. Each line contains the data for a single student, with contain several fields. Each field is separated by a colon.

Each line consists of the following fields, in order:

- A student number, which must be a 5 digit, 6 digit or 9 digit number.
- The student's month of birth, which must be a string of 3 alphabetic characters with the first character capitalised and the remaining in lower case. That is, it must be from the set:

$\{Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec\}$

- The day of birth, which must be a number from 1–31 and must be within the valid range of days in that month; that is, it cannot be 30 February, because February never has 30 days.
- The student's surname, which must be a string of alphabetic characters all in capitals.
- The first letter of the student's first name, which must be a single capitalised alphabetic character.
- The number of lectures that they slept through, which must be an integer between 0 and 24 (both inclusive).

If any input row is invalid, the program should print a warning message and continue with the next record. If the program encounters a more serious problem (e.g. unable to open input file), it will print an error message and exit gracefully.

**Example 1.** *As an example, suppose we had the following data.*

- *Student number: 12345*
- *Month of Birth: May*
- *Day of Birth: 26*
- *Surname: CHAN*
- *First letter of first name: K*
- *Number of lecture(s) slept through: 0*

*The input line for this data would be 12345:May:26:CHAN:K:0*

## 12.3 Tasks

1. What is the input domain for the LWIG program? What are the input conditions for the LWIG program?
2. Derive input test-cases for the program using equivalence partitioning and boundary-value analysis.
3. Implement your tests in the JUnit driver below. Do your tests find any faults?
4. Of course, the client has not completely specified the program (but, that is to be expected). There is an additional requirement that the records can be sorted by different output fields. What are the implications of sorting on the various fields and what test cases would you choose to ensure that sorting has been correctly implemented?

## 12.4 Java Implementation

The following code is a minimal implementation of the LWIG program.

For the purpose of this tutorial, let's assume that the input file has already been parsed into an array containing the important elements.

### 12.4.1 Prepare the Java Kernel

Since Java is not natively supported by Colab, we need to run the following code to enable Java kernel on Colab.

1. Run the cell bellow (click it and press Shift+Enter),
2. Refresh the Notebook (F5)
3. Change the kernel to Java (Runtime -> Change Runtime Type -> Java)

```
%%loadFromPOM

<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13.2</version>
</dependency>
```

## 12.5 LWIG Implementation

The following is a basic Java implementation of LWIG.

```
import java.time.*;
import java.time.format.*;
import java.util.Arrays;
import java.util.Collection;
import java.util.Locale;

public class LWIG {

    public static boolean isValidDate(String month, String date) {
        boolean isValid = false;
        try {
            DateTimeFormatter dtf = new DateTimeFormatterBuilder()
```

(continues on next page)

(continued from previous page)

```

        .parseCaseInsensitive()
        .appendPattern("d-MMM-uuuu")
        .toFormatter(Locale.ENGLISH);
    LocalDate.parse(String.format("%s-%s-2012", date, month), dtf); // Hard-coded a leap year or common year
    isValid = true;
} catch (Exception e) {
    //e.printStackTrace();
    isValid = false;
}
return isValid;
}

public static boolean isValidID(String id) {
    String regex = "^(\d{5}|\d{6}|\d{9}|\d{10})$"; //
    return id.matches(regex);
}

public static boolean isValidSurname(String name)
{
    String regex = "[A-Z]+$";
    return name.matches(regex);
}

public static boolean isValidFirstLetter(String name)
{
    String regex = "[A-Z]$";
    return name.matches(regex);
}

public static boolean isValidSleptCount(String number)
{
    boolean isValid = false;
    try {
        int count = Integer.parseInt(number);
        if (count >= 0 && count < 24) {
            isValid = true;
        }
    } catch (Exception e) {
        isValid = false;
    }
    return isValid;
}
}

```

## 12.6 JUnit test script

The following code block is a JUnit test script. JUnit is a unit-testing framework for Java that allows you to easily create tests that can be run automatically.

In the code block below, put your test cases where it says “Your test cases start here”. Add test cases by adding new elements to the data array. These will then be executed automatically by JUnit.

```

import junit.framework.TestCase;

import org.junit.Test;
import org.junit.runner.*;
import org.junit.runner.RunWith;
import org.junit.runner.notification.Failure;
import org.junit.runners.Parameterized;

@RunWith(Parameterized.class)
public class LWIGTestCase extends TestCase {

    @Parameterized.Parameter(0)
    public String id;
    @Parameterized.Parameter(1)
    public String month;
    @Parameterized.Parameter(2)
    public String date;
    @Parameterized.Parameter(3)
    public String surname;
    @Parameterized.Parameter(4)
    public String firstLetter;
    @Parameterized.Parameter(5)
    public String sleptCount;
    @Parameterized.Parameter(6)
    public boolean result;

    @Test
    public void testLWIG() {
        boolean expectedResult = LWIG.isValidID(id) &&
            LWIG.isValidDate(month, date) &&
            LWIG.isValidSurname(surname) &&
            LWIG.isValidFirstLetter(firstLetter) &&
            LWIG.isValidSleptCount(sleptCount);
        assertEquals(result, expectedResult);
    }

    @Parameterized.Parameters(name = "Test case {index} failed: LWIG with id = {0}, {1}, {2}, {3}, {4}, {5}, {6}",
        ↪ Mon = {1}, date = {2}, surname = {3}, first letter = {4}, slept count = {5}, ↪
        ↪ expectedResult = {6}")
    public static Collection<Object[]> data() {
        Object[][] data = new Object[][]{
            // Your Test cases start here
            {"13456", "Jan", "30", "CHAN", "K", "1", true},
            {"12345678", "Feb", "26", "TOM", "3", "0", true}
            // Your Test cases end here
        };
        return Arrays.asList(data);
    }
}

```

```

Result result = JUnitCore.runClasses(LWIGTestCase.class);
for (Failure failure : result.getFailures()) {
    System.out.println(failure.toString());
}
System.out.println(String.format("Total run count: %s, Failed run count: %s", result.
    ↪ getRunCount(), result.getFailureCount()));

```



```
testLWIG[Test case 1 failed: LWIG with id = 12345678, Mon = Feb, date = 26, surname =  
↪TOM, first letter = 3, slept count = 0, expected result = true](REPL.$JShell$27  
↪$LWIGTestCase): expected:<true> but was:<false>
```

```
Total run count: 2, Failed run count: 1
```



## SWEN90006 TUTORIAL 3

### 13.1 Introduction

The aim of this tutorial is for you to familiarise yourself with the various coverage criteria and analysis of the program for the various coverage criteria. When you get back to your revision you should try comparing the test cases that you derive for a program using different techniques.

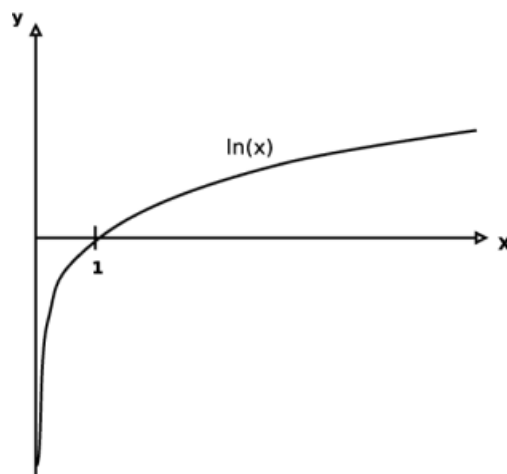
The different type of program that we encounter this week is a numerical program. One of the challenges of numerical programs is that we can never be certain that we will get an *exact* answer to our computation. Instead what we typically require is an answer to within some *error* value. (Recall from the lecture notes that an error is the difference between a computed value and the exact value). Numerical programs are tricky to debug, because they are often used to *find* the answer to some problem in the first place. For example, solving some integration or differentiation problems is too hard to do by hand and so we use a *numerical* method to approximate the answer.

### 13.2 Working With the Program

The program implements the standard bisection method for root finding. The root-finding problem is expressed as follows:

We are given a function  $f(x)$  taking a real number and returning a real number; that is a function  $f : \mathbb{R} \rightarrow \mathbb{R}$ . The function is negative at some point  $x_0$  and positive at some point  $x_1$ . Find the value  $x$  for which  $f(x) = 0$  on an interval  $[Lower, Upper]$ . The point  $x$  is a root of  $f$  on the interval  $[Lower, Upper]$ .

As an example, consider the natural logarithm function,  $\ln$ . The graph in Figure 1 shows the values of  $\ln(x)$  for various values of  $x$ . We can see that the value of  $\ln(x)$  is equal to 0 when  $x = 1$ . The bisection algorithm finds this value of  $x$ .



The idea behind the algorithm for finding roots is to look at the interval  $[Lower, Upper]$  and bisect it (hence the name of the algorithm) and find the midpoint of the interval  $x_r$ . If we know that  $f(Lower)$  is negative, and  $f(Upper)$  is positive then there must be root in the interval, provided that the function is continuous. If the value of  $f$  at  $x_r$  is positive then the root must be in the interval  $[Lower, x_r]$ . If the value of  $f$  at  $x_r$  is negative then the root must be in the interval  $[x_r, Upper]$ . The algorithm should converge to the root because the length of the interval is getting smaller every time (in fact the length of the interval is halved every time). Does this sound familiar?

## 13.3 Your Tasks

1. What is the input domain for the `Bisection` program below?
2. Draw the control-flow graph for the `Bisection` function. You may break the function up into basic blocks to simplify your CFG.

**Recall** that a *basic block* is a continuous sequence of statements where control flows from one statement to the next, a single point of entry, a single point of exit and no branches or loops.

3. Suppose that we concentrated on the (nice and linear) function  $f(x) = x - 2$ . Derive a set of test cases that achieve:
  - Statement coverage; and
  - Condition coverage.

Note that you will have to determine what it means for the `Bisection` function to return the *correct* or *expected* output first.

4. **Extended task:** After the tutorial, try implementing the tests in the JUnit driver below.

## 13.4 The Program

### 13.4.1 Prepare the Java Kernel

Since Java is not natively supported by Colab, we need to run the following code to enable Java kernel on Colab.

1. Run the cell bellow (click it and press Shift+Enter),
2. Refresh the Notebook (F5)
3. Change the kernel to Java (Runtime -> Change Runtime Type -> Java)

```
!wget https://github.com/SpencerPark/IJava/releases/download/v1.3.0/ijava-1.3.0.zip
!unzip ijava-1.3.0.zip
!python install.py --sys-prefix
```

```
%%loadFromPOM

<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13.2</version>
</dependency>
```

The following is a basic Java implementation of `Bisection`.

```

public class Bisection{

    static final int MAX_INT = 65535;

    static double bisection(double lower, double upper, double error, int max) {

        double sign = 0.0; /* Test for the sign of the midpoint xr. */
        double ea = MAX_INT; /* Calculated error value. */
        double xrold = 0.0; /* Previous estimate. */
        double xr = 0.0; /* Current x estimate for the root. */
        double fr = 0.0; /* Current value of f. */
        double fl = 0.0; /* Value of f at the lower end of the interval. */
        int iteration = 0; /* For keeping track of the number of iterations. */

        fl = func(lower);
        while ((ea > error && iteration < max)) {

            /* Start by memorising the old estimate in xrold and then calculate
            the new estimate and store in fr */
            xrold = xr;
            xr = (lower + upper) / 2;
            fr = func(xr);
            iteration++;
            /* Estimate the percentage error and store in ea. */
            if (xr != 0) {
                ea = Math.abs((xr - xrold)/xr) * 100;
            }

            /* To know whether fr has the same sign as f(Lower) or f(Upper) is easy:
            we know that f(Lower) is negative and we know that f(Upper) is positive.
            Multiple fr by f(Lower) and if the result is positive then fr must be
            negative. If the result is negative then fr must be positive. */

            sign = func(lower) * fr;
            if (sign < 0)
                upper = xr;
            else if (sign > 0)
                lower = xr;
            else
                ea = 0;
            System.out.println(String.format("iteration %d = (%f, %f, %f, %f, %f)\n",
iteration, lower, upper, xr, ea, sign));
        }
        return xr;
    }

    static double func(double x) {
        return x - 2;
    }
}

```

## 13.5 JUnit test script

The following code block is a JUnit test script. JUnit is a unit-testing framework for Java that allows you to easily create tests that can be run automatically.

In the code block below, put your test cases where it says “Your test cases start here”. Add test cases by adding new elements to the data array. These will then be executed automatically by JUnit.

```
import java.util.Arrays;
import java.util.Collection;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runner.RunWith;
import org.junit.runner.notification.Failure;
import org.junit.runners.Parameterized;

import junit.framework.TestCase;

@RunWith(Parameterized.class)
public class TestBisection extends TestCase {

    @Parameterized.Parameter(0)
    public double lower;
    @Parameterized.Parameter(1)
    public double upper;
    @Parameterized.Parameter(2)
    public double error;
    @Parameterized.Parameter(3)
    public int max;
    @Parameterized.Parameter(4)
    public double results;

    @Parameterized.Parameters(name = "{index}: lower: {0} upper:{1} error:{2} iterations:{3} results:{5}")
    public static Collection<Object[]> data() {
        Object[][] data = new Object[][]{
            // Your Test cases start here
            // Please follow the pattern: lower, upper, error, iterations, expected results
            {-1.0, 7.0, 1, 10, 2.0},
            {-1.0, 7.0, 1, 10, 2.0}
            // Your Test cases end here
        };
        return Arrays.asList(data);
    }

    @Test
    public void testBisection() {
        assertEquals(results, Bisection.bisection(lower, upper, error, max));
    }
}
```

```
Result result = JUnitCore.runClasses(TestBisection.class);
for (Failure failure : result.getFailures()) {
    System.out.println(failure.toString());
}
```

(continues on next page)

(continued from previous page)

```
}  
System.out.println(String.format("Total run count: %s, Failed run count: %s", result.  
    ↪getRunCount(), result.getFailureCount()));
```

```
iteration 1 = (-1.000000, 3.000000, 3.000000, 100.000000, -3.000000)
```

```
iteration 2 = (1.000000, 3.000000, 1.000000, 200.000000, 3.000000)
```

```
iteration 3 = (1.000000, 3.000000, 2.000000, 0.000000, -0.000000)
```

```
iteration 1 = (-1.000000, 3.000000, 3.000000, 100.000000, -3.000000)
```

```
iteration 2 = (1.000000, 3.000000, 1.000000, 200.000000, 3.000000)
```

```
iteration 3 = (1.000000, 3.000000, 2.000000, 0.000000, -0.000000)
```

```
Total run count: 2, Failed run count: 0
```





## SWEN90006 TUTORIAL 4

### 14.1 Introduction

The purpose of this tutorial is for you to gain a deeper understanding of control- and data-flow techniques, and to compare these to input partitioning techniques.

### 14.2 Working With the Program

In this tutorial we will focus on the procedure `bubble`, found in Figure 1. The program below serves to show you how we intend to use `bubble`. Of course, `bubble` implements a bubble sort.

Make special note of the third parameter to `bubble`, called `order`. This parameter allows us to create a flexible sorting algorithm, which can be used to sort in either ascending or descending order. Both functions `up` and `down` take two integers and return an integer (which is intended to be a boolean value). For this tutorial consider just

```
public static boolean up(int A, int B) {  
    return A < B;  
}
```

and

```
public static boolean down(int A, int B) {  
    return B < A;  
}
```

#### 14.2.1 Prepare the Java Kernel

Since Java is not natively supported by Colab, we need to run the following code to enable Java kernel on Colab.

1. Run the cell below (click it and press Shift+Enter),
2. Refresh the Notebook (F5)
3. Change the kernel to Java (Runtime -> Change Runtime Type -> Java)

```
// !wget https://github.com/SpencerPark/IJava/releases/download/v1.3.0/ijava-1.3.0.zip  
// !unzip ijava-1.3.0.zip  
// !python install.py --sys-prefix
```

```
%%loadFromPOM
```

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13.2</version>
</dependency>
```

The following is a basic Java implementation of Bubble Sort.

```
public class Bubble {

    static final int SIZE = 10;
    static final int data[] = {11, 4, 8, 22, 15, 7, 8, 19, 20, 1};
    static int counter;
    static int order;

    public static boolean up(int A, int B) {
        return A < B;
    }

    public static boolean down(int A, int B) {
        return B < A;
    }

    public static void printArray() {
        for (counter = 0; counter < SIZE; counter++)
            System.out.print(data[counter] + " ");
        System.out.println();
    }

    public static int[] bubble(int data[], int size, int order) {
        int pass, count;
        for (pass = 0; pass < SIZE - 1; pass++) {
            for (count = 0; count < SIZE - 1; count++) {
                if (order == 0) {
                    if (up(data[count], data[count + 1])) {
                        swap(data, count);
                    }
                } else {
                    if (down(data[count], data[count + 1])) {
                        swap(data, count);
                    }
                }
            }
        }
        return data;
    }

    public static void swap(int data[], int count) {
        int temp = data[count];
        data[count] = data[count+1];
        data[count+1] = temp;
    }
}
```

```
// bubble (int[] data, int size, int order), 0 for descending, and 1 for ascending
```

(continues on next page)

(continued from previous page)

```
Bubble.bubble(Bubble.data, 10, 1);
Bubble.printArray();
```

1

4

7

8

8

11

15

19

20

22

```
import org.junit.Assert;
import java.util.Arrays;
import java.util.Collection;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runner.RunWith;
import org.junit.runner.notification.Failure;
import org.junit.runners.Parameterized;

import junit.framework.TestCase;

@RunWith(Parameterized.class)
public class TestBubble extends TestCase {

    @Parameterized.Parameter(0)
    public int[] data;
    @Parameterized.Parameter(1)
    public int size;
    @Parameterized.Parameter(2)
    public int order;
    @Parameterized.Parameter(3)
    public int[] result;

    @Parameterized.Parameters(name = "{index}: data: {0} size:{1} order:{2} results:
↪{3}")
```

(continues on next page)

(continued from previous page)

```

public static Collection<Object[]> data() {
    Object[][] data = new Object[][]{
        // Your Test cases start here
        {new int[]{1, 3, 2, 4, 5, 6, 7, 8, 10, 9}, 10, 1, new int[]{1, 2, 3, 4, 5,
→ 6, 7, 8, 9, 10}},
        // Your Test cases end here
    };
    return Arrays.asList(data);
}

@Test
public void testBubble() {
    Assert.assertArrayEquals(result, Bubble.bubble(data, size, order));
}
}

```

```

Result result = JUnitCore.runClasses(TestBubble.class);
for (Failure failure : result.getFailures()) {
    System.out.println(failure.toString());
}
System.out.println(String.format("Total run count: %s, Failed run count: %s", result.
→getRunCount(), result.getFailureCount()));

```

```
Total run count: 1, Failed run count: 0
```

## 14.3 Your Tasks

### 14.3.1 Task 1

First, make sure you understand the program (expected to be done before the tutorial).

### 14.3.2 Task 2

Determine the input domains and output domains for the functions `bubble`.

Next, what are the input conditions?

### 14.3.3 Task 3

Perform a static data-flow analysis on `bubble`. Draw a table such as in the notes to complete this.

#### **14.3.4 Task 4**

Roughly sketch a set of black box test cases using equivalence partitioning.

#### **14.3.5 Task 5**

What extra information does your data-flow analysis give you that your black-box test cases do not?



## SWEN90006 TUTORIAL 5

### 15.1 Introduction

Encapsulation is an abstraction mechanism that aids in programming, but that adds complexity to testing. We often need to break the information hiding utilised by classes in order to examine the class state for the testing purposes.

The aim of this tutorial is for you to explore some of the issues in object oriented testing through the simple Graph given below. The graph uses an adjacency matrix that records which vertices are “*adjacent*” in the graph.

### 15.2 Working With the Program

#### 15.2.1 Prepare the Java Kernel

Since Java is not natively supported by Colab, we need to run the following code to enable Java kernel on Colab.

1. Run the cell bellow (click it and press Shift+Enter),
2. Refresh the Notebook (F5)
3. Change the kernel to Java (Runtime -> Change Runtime Type -> Java)

```
#!/wget https://github.com/SpencerPark/IJava/releases/download/v1.3.0/ijava-1.3.0.zip
#!/unzip ijava-1.3.0.zip
#!/python install.py --sys-prefix
```

```
%%loadFromPOM

<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13.2</version>
</dependency>
```

The following is a basic Java implementation of Graph.

```
import java.util.*;

public class Graph {

    //----- Private Attributes -----
    //    The representation of a graph consists of an array
    //    of vertices that map nodes (just integers) to array
```

(continues on next page)

(continued from previous page)

```

//    indexes. The adjacency matrix _matrix sets matrix[i][j]
//    to true if there is an edge between _vertices[i] and
//    _vertices[j].
//
//    The operations must maintain the following invariant:
//    _vertices[i] is defined iff i < _allocated
//    _allocated <= _order
static final String EMPTY_GRAPH = "Empty Graph";

private int    _order;        // The number of vertices allowed
private int    _allocated;    // The next free space in the vertex array
private int    _vertices[];   // A list of the actual vertices
private boolean _matrix[][];  // The adjacency matrix

public Graph(int n) {
    // Create a matrix of size n and initial all of the
    // state variables so that the invariant is maintained.
    _order = n;
    _allocated = 0;
    _vertices = _vertices(n);
    _matrix = _allocate(n);
}

public Graph(int n, int allocated, int[] vertices, boolean matrix[][]) {
    _order = n;
    _allocated = allocated;
    _vertices = vertices;
    _matrix = matrix;
}

private static boolean[][] _allocate(int n) {
    return new boolean[n][n];
}

private static int[] _vertices(int n) {
    return new int[n];
}

private int _lookup(int m) {
    int index = 0;
    while (index < _allocated && _vertices[index] != m) {
        index = index + 1;
    }

    if (index == _allocated) {
        return _order + 1;
    }
    else {
        return index;
    }
}

public void addVertex(int v) {
    // Add a vertex to the vertex graph. For the moment we
    // assume that vertices are just integers.
    if (_allocated < _order) {
        _vertices[_allocated] = v;
    }
}

```

(continues on next page)



(continued from previous page)

```

        _allocated = _allocated + 1;
    }
}

public void addEdge(int m, int n) {
    // Add an edge to the graph. Edges are specified by pairs
    // of vertices. To add the edge correctly it is necessary
    // that m and n have already been added to graph as vertices.

    int mIndex = _lookup(m);
    int nIndex = _lookup(n);

    if (mIndex < _order && nIndex < _order) {
        _matrix[mIndex][nIndex] = true;
        _matrix[nIndex][mIndex] = true;
    }
}

public void deleteVertex(int v) {
    // We can only delete a node if it is not part of some edge
    // in the graph and it exists as an actual vertex in the
    // graph.

    int vIndex = _lookup(v);

    if (vIndex < _order) {
        boolean isEdge = false;
        for (int i = 0; i < _allocated; i++) {
            isEdge = isEdge || _matrix[vIndex][i] || _matrix[i][vIndex];
        }

        if (!isEdge) {
            for (int i = vIndex; i < _allocated-1; i++) {
                _vertices[i] = _vertices[i+1];
            }
            _allocated = _allocated - 1;
        }
    }
}

public void deleteEdge(int m, int n) {
    // We can only delete an edge if the two specified
    // vertices are in the graph.

    int mIndex = _lookup(m);
    int nIndex = _lookup(n);

    if (mIndex < _order && nIndex < _order)
        _matrix[mIndex][nIndex] = false;
}

// Overrides ToString()
public String toString() {
    StringBuilder s = new StringBuilder();
    if (_allocated == 0)
    {
        return EMPTY_GRAPH;
    }
}

```

(continues on next page)

(continued from previous page)

```

    }
    else {
        s.append("\n ");
        for (int i = 0; i < _allocated; i++)
        {
            s.append(i + " ");
        }
        s.append("\n");
        for (int i = 0; i < _allocated; i++) {
            s.append(i + ": ");
            for (boolean j : _matrix[i]) {
                s.append((j ? "T" : " ") + " ");
            }
            s.append("\n");
        }
        return s.toString();
    }
}
}

```

The code block below demonstrates how to create graph

```

// Method 1: pass all params into the constructor
// Pre-define a matrix
boolean[][] matrix = new boolean[6][6];
for (int i = 0; i < matrix.length ; i++) {
    Arrays.fill(matrix [i], false);
}

matrix[1][3] = true;
matrix[3][1] = true;

// Graph(int n, int allocated, int[] vertices, boolean matrix[][])
// int n: The number of vertices allowed
// int allocated: The next free space in the vertex array
// int[] vertices: A list of the actual vertices
// boolean matrix[][]: The adjacency matrix
Graph graph_1 = new Graph(6, 6, new int[] {0, 1, 2, 3, 4, 5}, matrix);

// Print the Graph
System.out.println(graph_1);

// Method 2:
Graph graph_2 = new Graph(6);
System.out.println(graph_2);
System.out.println();
// Add Vertex
graph_2.addVertex(0);
graph_2.addVertex(1);
graph_2.addVertex(2);
graph_2.addVertex(3);
System.out.println(graph_2);
graph_2.addVertex(4);
graph_2.addVertex(5);

// Add Edge
graph_2.addEdge(1, 3);

```

(continues on next page)

(continued from previous page)

```
// Print the Graph
System.out.println(graph_2);

// Check if they are identical or not
// In this case, it should be true
System.out.println(graph_1.toString().equals(graph_2.toString()))
```

```
    0 1 2 3 4 5
0:
1:      T
2:
3:    T
4:
5:
```

Empty Graph

```
    0 1 2 3
0:
1:
2:
3:
```

```
    0 1 2 3 4 5
0:
1:      T
2:
3:    T
4:
5:
```

true

```
import java.util.Arrays;
import java.util.Collection;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runner.Result;
```

(continues on next page)

(continued from previous page)

```

import org.junit.runner.RunWith;
import org.junit.runner.notification.Failure;
import org.junit.runners.Parameterized;

import junit.framework.TestCase;

@RunWith(Parameterized.class)
public class TestGraph extends TestCase {
    @Parameterized.Parameter(0)
    public String actual;
    @Parameterized.Parameter(1)
    public String expected;

    @Parameterized.Parameters(name = "{index}:\ngraph 1: \n{0}\ngraph 2:\n{1}\n")
    public static Collection<Object[]> data() {

        // Define your params and constructor here
        boolean[][] matrix = new boolean[6][6];
        for (int i = 0; i < matrix.length; i++) {
            Arrays.fill(matrix[i], false);
        }

        matrix[1][3] = true;
        matrix[3][1] = true;

        Graph expectedGraph = new Graph(6, 6, new int[]{0, 1, 2, 3, 4, 5}, matrix);

        Graph actualGraph = new Graph(6);
        actualGraph.addVertex(0);
        actualGraph.addVertex(1);
        actualGraph.addVertex(2);
        actualGraph.addVertex(3);
        actualGraph.addVertex(4);
        actualGraph.addVertex(5);
        actualGraph.addEdge(1, 3);

        Graph emptyGraph = new Graph(6);
        // End of pre-define

        Object[][] data = new Object[][]{
            // Your Test cases start here
            // Please follow the pattern: expected graph, actual graph

            // Success
            {expectedGraph.toString(), actualGraph.toString()},
            // Success
            {Graph.EMPTY_GRAPH, emptyGraph.toString()},
            // Fail
            {Graph.EMPTY_GRAPH, actualGraph.toString()}
            // Your Test cases end here
        };
        return Arrays.asList(data);
    }

    @Test
    public void testGraph() {

```

(continues on next page)

(continued from previous page)

```

        assertEquals(expected, actual);
    }
}

```

```

Result result = JUnitCore.runClasses(TestGraph.class);
for (Failure failure : result.getFailures()) {
    System.out.println(failure.toString());
}
System.out.println(String.format("Total run count: %s, Failed run count: %s", result.
    ↪getRunCount(), result.getFailureCount()));

```

```

testGraph[2:
graph 1:
Empty Graph
graph 2:

    0 1 2 3 4 5
0:
1:      T
2:
3:    T
4:
5:

] (REPL.$JShell$44$TestGraph): expected:<[
    0 1 2 3 4 5
0:
1:      T
2:
3:    T
4:
5:
]> but was:<[Empty Graph]>

```

```
Total run count: 3, Failed run count: 1
```

## 15.3 Your Tasks

### 15.3.1 Task 1

Consider the `addEdge` and `deleteEdge` methods in the `Graph` class above. Derive test cases for path coverage and condition coverage for these two methods. **Note** that it may be necessary to examine the state variables in your test cases. Sketch how you would achieve this.

### 15.3.2 Task 2

Given that a `Graph` object has already been initialized with the number `K`, draw a finite state automaton for this `Graph` object. **Note** that it is not always possible to add an edge and it is not always possible to delete a vertex. You will need to consider the states and the guards on transitions to ensure all of the conditions.

### 15.3.3 Task 3

Derive a set of test cases to test every transition in your graph.

## SWEN90006 TUTORIAL 6

### 16.1 Introduction

The aim of this tutorial is for you to familiarise yourself with the test oracles and random testing. This tutorial will help to gain an understanding of how to specify a test oracle to decide whether an arbitrary test case is correct, incorrect or undecidable, as well as how to randomly generate test cases that fit an operational profile.

### 16.2 Working With the Program

#### 16.2.1 First Program

We shall start this tutorial by using a rather simple example to illustrate the concepts, techniques and issues faced when working with oracles. This is followed by applying these techniques to the root finding program from tutorial 4 to gain some practice with practical examples.

This *toLower* is an application which takes a string on standard input and puts a corresponding string on standard output, with all upper case letters changed to the matching lower case letters (i.e. 'A' to 'a', 'B' to 'b', etc.). Strings may consist of all ASCII characters between 32 and 126 inclusive; characters outside this range are control characters and will cause an error message.

#### 16.2.2 Prepare the Java Kernel

Since Java is not natively supported by Colab, we need to run the following code to enable Java kernel on Colab.

1. Run the cell bellow (click it and press Shift+Enter),
2. Refresh the Notebook (F5)
3. Change the kernel to Java (Runtime -> Change Runtime Type -> Java)

```
#!/wget https://github.com/SpencerPark/IJava/releases/download/v1.3.0/ijava-1.3.0.zip
#!/unzip ijava-1.3.0.zip
#!/python install.py --sys-prefix
```

```
%%loadFromPOM

<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
```

(continues on next page)

(continued from previous page)

```
<version>4.13.2</version>
</dependency>
```

The following is a basic Java implementation of ToLower.

```
public class ToLower {

    static final int MAX_STRING = 80;

    public static char[] toLower(String input)
    {
        //Scanner scan = new Scanner(System.in);
        char[] string = subString(input, 0, MAX_STRING).toCharArray();
        for (int i = 0; i < string.length - 1; i++) {
            if (string[i] < 32 || string[i] == 127) {
                System.out.println("Illegal character found.\n");
                return "Illegal".toCharArray();
            }

            if (string[i] >= 65 || string[i] <= 90) {
                string[i] = (char) (string[i] + 'a' - 'A');
            }
        }
        return string;
    }

    public static void main(String[] args) {
        System.out.println(toLower("€"));
    }

    public static String subString(String myString, int start, int length) {
        return myString.substring(start, Math.min(start + length, myString.length()));
    }
}
```

```
import org.junit.Assert;
import java.util.Arrays;
import java.util.Collection;

import org.junit.Test;
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.RunWith;
import org.junit.runner.notification.Failure;
import org.junit.runners.Parameterized;

import junit.framework.TestCase;

@RunWith(Parameterized.class)
public class TestToLower extends TestCase {
    @Parameterized.Parameter(0)
    public String actual;
    @Parameterized.Parameter(1)
    public char[] expected;
```

(continues on next page)



(continued from previous page)

```

@Parameterized.Parameters(name = "{index}: actual: {0} expected:{1}")
public static Collection<Object[]> data() {

    Object[][] data = new Object[][]{
        // Your Test cases start here

        {"KKK", new char[] {'k', 'k', 'k'}},
        // Your Test cases end here
    };
    return Arrays.asList(data);
}

@Test
public void testLower() {
    Assert.assertArrayEquals(expected, ToLower.toLowerCase(actual));
}

public static void main(String[] args) {

}
}

```

```

Result result = JUnitCore.runClasses(TestToLower.class);
for (Failure failure : result.getFailures()) {
    System.out.println(failure.toString());
}
System.out.println(String.format("Total run count: %s, Failed run count: %s", result.
    ↪getRunCount(), result.getFailureCount()));

```

```

testLower[0: actual: KKK expected:[C@61d45796] (REPL.$JShell$26$TestToLower): arrays
    ↪first differed at element [2]; expected:<k> but was:<K>

```

```

Total run count: 1, Failed run count: 1

```

## 16.2.3 Second Program

The idea behind the algorithm for finding roots is to look at the interval  $[Lower, Upper]$  and bisect it (hence the name of the algorithm) and find the midpoint of the interval  $x_r$ . If we know that  $Lower$  is negative, and  $Upper$  is positive then there must be root in the interval, provided that the function is continuous. If the value of  $f$  at  $x_r$  is positive then the root must be in the interval  $[Lower, x_r]$ . If the value of  $f$  at  $x_r$  is negative then the root must be in the interval  $[x_r, Upper]$ . The algorithm should converge to the root because the length of the interval is getting smaller every time (in fact the length of the interval is halved every time).

The following is a basic Java implementation of BiSection.

```

public class Bisection{

    static final int MAX_INT = 65535;

```

(continues on next page)

(continued from previous page)

```

static double bisection(double lower, double upper, double error, int max) {

    double sign = 0.0; /* Test for the sign of the midpoint xr. */
    double ea = MAX_INT; /* Calculated error value. */
    double xrold = 0.0; /* Previous estimate. */
    double xr = 0.0; /* Current x estimate for the root. */
    double fr = 0.0; /* Current value of f. */
    double fl = 0.0; /* Value of f at the lower end of the interval. */
    int iteration = 0; /* For keeping track of the number of iterations. */

    fl = func(lower);
    while ((ea > error && iteration < max)) {

        /* Start by memorising the old estimate in xrold and then calculate
        the new estimate and store in fr */
        xrold = xr;
        xr = (lower + upper) / 2;
        fr = func(xr);
        iteration++;
        /* Estimate the percentage error and store in ea. */
        if (xr != 0) {
            ea = Math.abs((xr - xrold)/xr) * 100;
        }

        /* To know whether fr has the same sign as f(Lower) or f(Upper) is easy:
        we know that f(Lower) is negative and we know that f(Upper) is positive.
        Multiple fr by f(Lower) and if the result is positive then fr must be
        negative. If the result is negative then fr must be positive. */

        sign = func(lower) * fr;
        if (sign < 0)
            upper = xr;
        else if (sign > 0)
            lower = xr;
        else
            ea = 0;
        System.out.println(String.format("iteration %d = (%f, %f, %f, %f, %f)\n",
iteration, lower, upper, xr, ea, sign));
    }
    return xr;
}

static double func(double x) {
    return x - 2;
}
}

```

```

import java.util.Arrays;
import java.util.Collection;

import org.junit.Test;
import org.junit.runner.JUnit4;
import org.junit.runner.Result;
import org.junit.runner.RunWith;
import org.junit.runner.notification.Failure;

```

(continues on next page)

(continued from previous page)

```

import org.junit.runners.Parameterized;

import junit.framework.TestCase;

@RunWith(Parameterized.class)
public class TestBisection extends TestCase {

    @Parameterized.Parameter(0)
    public double lower;
    @Parameterized.Parameter(1)
    public double upper;
    @Parameterized.Parameter(2)
    public double error;
    @Parameterized.Parameter(3)
    public int max;
    @Parameterized.Parameter(4)
    public double results;

    @Parameterized.Parameters(name = "{index}: lower: {0} upper:{1} error:{2} iterations:{3} results:{5}")
    public static Collection<Object[]> data() {
        Object[][] data = new Object[][]{
            // Your Test cases start here
            // Please follow the pattern: lower, upper, error, iterations, expected results
            {-1.0, 7.0, 1, 10, 2.0},
            {-1.0, 7.0, 1, 10, 2.0}
            // Your Test cases end here
        };
        return Arrays.asList(data);
    }

    @Test
    public void testBisection() {
        assertEquals(results, Bisection.bisection(lower, upper, error, max));
    }
}

```

```

Result result = JUnitCore.runClasses(TestBisection.class);
for (Failure failure : result.getFailures()) {
    System.out.println(failure.toString());
}
System.out.println(String.format("Total run count: %s, Failed run count: %s", result.
    getRunCount(), result.getFailureCount()));

```

```
iteration 1 = (-1.000000, 3.000000, 3.000000, 100.000000, -3.000000)
```

```
iteration 2 = (1.000000, 3.000000, 1.000000, 200.000000, 3.000000)
```

```
iteration 3 = (1.000000, 3.000000, 2.000000, 0.000000, -0.000000)
```

```
iteration 1 = (-1.000000, 3.000000, 3.000000, 100.000000, -3.000000)
```

```
iteration 2 = (1.000000, 3.000000, 1.000000, 200.000000, 3.000000)
```

```
iteration 3 = (1.000000, 3.000000, 2.000000, 0.000000, -0.000000)
```

```
Total run count: 2, Failed run count: 0
```

## 16.3 Your Tasks

Repeat these tasks for both programs.

### 16.3.1 Task 1

Standard analysis: Determine the input/output domains and the input/output conditions.

### 16.3.2 Task 2

Using the specification and the input domain, write an automated test oracle to determine if an arbitrary test input is correct, incorrect or otherwise undecidable.

### 16.3.3 Task 3

Determine an automated means for generating random test cases by selection points from the input set.