

Atılım University
Department of Software Engineering

Practical Software Testing

Ali Yazıcı* Tolga Üstüncök Özge Tekin Zeynep Yaren Oğuz

January 2023

Atılım University

* E-mail: ali.yazici@atilim.edu.tr

Disclaimer

You can edit this page to suit your needs. For instance, here we have a no copyright statement, a colophon and some other information. This page is based on the corresponding page of Ken Arroyo Ohori's thesis, with minimal changes.

No copyright

© This book is released into the public domain using the CC0 code. To the extent possible under law, I waive all copyright and related or neighbouring rights to this work.

To view a copy of the CC0 code, visit:

<http://creativecommons.org/publicdomain/zero/1.0/>

Colophon

This document was typeset with the help of KOMA-Script and L^AT_EX using the kaobook class.

The source code of this book is available at:

<https://github.com/fmarotta/kaobook>

(You are welcome to contribute!)

Publisher

First printed in May 2019 by Atilim University

“In many ways, being a good tester is harder than being a good developer because testing requires not only a very good understanding of the development process and its products, but it also demands an ability to anticipate likely faults and errors.”

–John D. McGregor

Practical Guide to Testing Object-Oriented Software

Addison Wesley, 2001

Preface

I am of the opinion that every \LaTeX geek, at least once during his life, feels the need to create his or her own class: this is what happened to me and here is the result, which, however, should be seen as a work still in progress. Actually, this class is not completely original, but it is a blend of all the best ideas that I have found in a number of guides, tutorials, blogs and tex.stackexchange.com posts. In particular, the main ideas come from two sources:

- ▶ [Ken Arroyo Ohori's Doctoral Thesis](#), which served, with the author's permission, as a backbone for the implementation of this class;
- ▶ The [Tufte-Latex Class](#), which was a model for the style.

The first chapter of this book is introductory and covers the most essential features of the class. Next, there is a bunch of chapters devoted to all the commands and environments that you may use in writing a book; in particular, it will be explained how to add notes, figures and tables, and references. The second part deals with the page layout and design, as well as additional features like coloured boxes and theorem environments.

I started writing this class as an experiment, and as such it should be regarded. Since it has always been intended for my personal use, it may not be perfect but I find it quite satisfactory for the use I want to make of it. I share this work in the hope that someone might find here the inspiration for writing his or her own class.

Federico Marotta

Contents

Preface	v
Contents	vii
SOFTWARE TESTING	1
1 Basic Concepts of Testing	3
1.1 Software Quality and Testing	3
1.2 Error, Fault, Defect and Failure	4
1.3 Objectives of Testing	5
1.4 Test Levels in the V-Model	5
1.5 White-box and Black-box Testing	6
1.6 Testing Activities	6
1.7 Problems	7
2 Unit (Component) Testing	9
2.1 Unit Testing Basics	9
2.2 Static Unit Testing – Code Reviews	10
2.3 Dynamic Unit Testing	14
2.4 Mutation Testing	14
2.5 Debugging	18
2.6 Unit Test Automation	19
2.7 Problems	19
3 Control Flow Testing	21
3.1 Introduction	21
3.2 Control Flow Graph (CFG)	22
3.3 McCabe Cyclomatic Complexity	23
3.4 Path Selection Criteria	23
3.5 Generating Test Cases	26
3.6 Problems	27
4 Data Flow Testing	29
4.1 Basic Concepts	29
4.2 Data Flow Graph (DFG)	30
4.3 Problems	34
5 Integration Testing	35
5.1 Objectives	35
5.2 Types of Interfaces and Interface Errors	35
5.3 Integration Techniques	35
5.3.1 Incremental Approach	35
5.3.2 Top-down Approach	35
5.3.3 Bottom-up approach	35
5.3.4 Sandwich and Bing-bang Approaches	35
5.4 Problems	35

6	System Test Categories	37
6.1	System Test Taxonomy	37
6.2	Basic Tests	37
6.3	Functionality Tests	37
6.4	Robustness Tests	37
6.5	Interoperability Tests	37
6.6	Performance Tests	37
6.7	Stress Tests	37
6.8	Scalability Tests	37
6.9	Reliability Tests	37
6.10	Regression Tests	37
6.11	Problems	37
7	Functional Testing	39
7.1	Howden's Approach	39
7.2	Pairwise Testing with Orthogonal Arrays	39
7.3	Orthogonal Array Generation with DEVELVE	39
7.4	Equivalence Class Partitioning (ECP)	39
7.5	Boundary Value Analysis	39
7.6	Cause-Effect Graphs	39
7.7	Decision Tables	39
7.8	Error Guessing	39
7.9	Problems	39
8	Software Testing Metrics	41
8.1	Measurement vs Metrics	41
8.2	Product Metrics for Testing	41
8.3	Process Metrics for Testing	41
8.4	Problems	41
9	Acceptance Testing	43
9.1	Type of Acceptance testing	43
9.2	Acceptance Criteria	43
9.3	Test plan	43
9.4	Problems	43
	LABORATORY STUDIES AND EXERCISES	45
10	Introduction to Lab Activities	47
10.1	Brief Summary of Java	47
10.1.1	Java Terminology	47
10.1.2	An Overview of a Hello World Program	48
10.2	Dependency Management with Maven	49
10.3	Installing Eclipse and Setting up a Maven Project for Testing	50
10.4	Introduction to JUnit Jupiter API	54
11	Unit Testing with JUnit	57
11.1	Developing Tests	57
11.2	Working with Assert Types	59
11.3	Running and Reporting Tests	59
11.4	Exercises	60

12 Control Flow Testing 1	65
12.1 McCabe Cyclomatic Complexity	65
12.2 Statement and Branch Coverage	66
12.3 Exercises	66
13 Data Flow Testing	77
13.1 Data Flow Graph	77
13.2 Exercises	77
14 Static Unit Testing	81
14.1 Code Review	81
14.2 Exercises	82
15 Black-Box Testing Strategies	85
15.1 Equivalence Class Partitioning	85
15.2 Boundary Value Analysis	85
15.3 Cause-and-Effect Graphing	87
15.4 Decision Tables	88
15.5 Error Guessing	88
15.6 Exercises	89
16 Web Testing with Selenium 1	97
16.1 Installation	97
16.1.1 Detailed Steps for Selenium Installation	98
16.2 Usage	99
17 Web Testing with Selenium 2	103
17.1 Examples	103
17.1.1 Additional Exercise with Selenium	103
17.1.2 Testing Atılım University Moodle Page	104
APPENDIX	107
A XML Path Language (XPath)	109
B Oracle VM VirtualBox Image	111
Bibliography	113
Notation	115
Index	117

List of Figures

1.1	Validation and Verification with the V-model	5
2.1	Validation and verification with the V-model.	14
2.2	A screenshot of Dev-C++Debugger.	18
2.3	A screenshot of Eclipse Debugger.	19
3.1	Basic nodes (computation, decision and merge) in a CFG.	22
3.2	if-then-else, switch, and while-do constructs in a CFG.	22
3.3	A CFG for the C program given in Example 3.2.1.	23
3.4	McCabe program graph for the C program given in Example 3.2.1.	23
3.5	McCabe program graph for the <code>isPalindrome</code> program.	25
3.6	CFG for the prime number test program.	26
3.7	McCabe program graph for the C program given in Example 3.2.1.	28
4.1	Data flow graph for the Newton-Raphson iteration to compute the square root of a real number.	31
4.2	Data flow graph for the program given in Example 3.4.1	31
10.1	Welcome screen of Eclipse.	51
10.2	Create a new project window.	51
10.3	Maven project wizard landing page.	52
10.4	Maven project wizard final page.	53
10.5	Java compiler properties for a Java project.	54
12.1	Flow graph for the Listing 12.1.	65
12.2	The flow graph of the <code>calculateTotalBill</code> function.	67
12.3	The flow graph of the <code>calculateFee</code> function.	69
12.4	The flow graph of the <code>firstCriteria</code> function.	72
12.5	The flow graph of the <code>bubbleSort</code> method.	74
13.1	Data flow graph example.	77
14.1	Passenger Service System passenger checks in activity diagram.	84
15.1	Cause-and-effect graph for the previously defined rules.	88
15.2	Cause-and-effect graph for the question.	94
16.1	Selenium dependency on the Maven repository website.	98
16.2	WebDriver installation page of the Selenium website.	99
16.3	Run configurations dialog window.	101
17.1	Number of search results on the Moodle web page.	103
B.1	Oracle Virtual Box Manager screenshot.	111

List of Tables

3.1	Path predicate interpretation.	27
4.1	Definitions, c-uses, p-uses and terminations (undefine) of the variables.	32
4.2	This caption should be filled with an appropriate sentence.	33
10.1	Maven directory layout.	49
11.1	Some of the assert methods that are defined in JUnit API.	59
12.1	Independent paths of the program that is described by Listing 12.1.	65
12.2	Independent paths.	67
12.3	Test cases.	67
12.4	Independent paths.	70
12.5	Test cases.	70
12.6	Independent paths of the <code>firstCriteria</code> method.	72
12.7	Test cases.	72
12.8	Independent paths of the <code>bubbleSort</code> method.	74
12.9	Test cases for <code>bubbleSort</code> method.	74
13.1	Def-Use pairs and related test cases.	79
13.2	Def-Use pairs and related test cases.	79
14.1	List of found defects.	83
15.1	Equivalence class reporting table.	86
15.2	Summary of test inputs using equivalence class partitioning and boundary value analysis for sample module. Table taken from [14].	86
15.3	Decision table for the previously defined cause-and-effect graph.	88
15.4	Test cases for equivalence classes.	89
15.5	Test cases for BVA strategy.	90
15.6	Suggested test cases for equivalence classes.	90
15.7	Suggested test cases for BVA strategy.	91
15.8	Test cases for equivalence classes.	91
15.9	Test cases for BVA strategy.	92
15.10	Suggested test cases for equivalence classes.	93
15.11	Suggested test cases for BVA strategy.	93
15.12	Decision table for the previously defined cause-and-effect graph.	95

List of Listings

2.1	A C function that solves the quadratic equation for the two real roots.	9
2.2	Binary search a C implementation.	10
2.3	A recursive C program to compute and print $0!, 1!, \dots, 10!$	15
2.4	Mutant 1: (Decision mutation - Change $\text{num} \leq 1$ to $\text{num} < 1$)	15
2.5	Mutant 2: (Value mutation - Change $\text{fact}(\text{num} - 1)$ to $\text{fact}(\text{num} - 100)$)	15
2.6	Mutant 3: (Decision mutant – Change if $(\text{num} \leq 1)$ to if $(\text{num} < 1)$)	16
2.7	A C program to compute and print primes from 2 to n	16
2.8	Mutant 1: (Decision mutant – Change $i \leq n$ to $i \geq n$)	17
2.9	Mutant 2: (Decision mutant – Change $i \leq n$ to $i < n$)	17

2.10	Mutant 3: (Value mutant – Change $k == 2$ to $k == 3$)	17
3.1	A C program that prints the negative integers in an array of 10 integer values.	22
3.2	A C program to check if a given string is a palindrome or not.	24
3.3	A C program to test if a number is prime or not.	25
4.1	A C function to compute the positive square root of a real number.	31
10.1	Hello world example written in Java.	48
10.2	An example pom.xml file.	50
10.3	JUnit Jupiter API and Engine version 5.8.2 dependency elements.	55
10.4	A compatible Maven Surefire Plugin with Eclipse and JUnit Jupiter API v5.8.2.. . . .	55
10.5	pom.xml file for future exercises.	55
11.1	A test case for testing the addition functionality of the Calculator class.	57
11.2	A standard test class.	58
11.3	A log output from a Maven test run.	59
11.4	A Calculator class implementation in Java.	60
11.5	Trivial unit tests for the Calculator class.	61
11.6	A unit test to testing the mean method of the Calculator class.	61
11.7	The simplest implementation to pass the test.	62
11.8	Another test that invalidates the previous implementation.	62
11.9	A correct implementation of the mean operation.	62
11.10	An implementation for multiplication operation with a loop.	63
11.11	A floating-point assert statement with precision.	63
12.1	pos_sum finds the sum of all positive numbers stored in an integer array a. Input parameter is a, an array of integers. The output of the function is sum, the sum of integers inside the array a.	65
12.2	A proposed solution for the above requirement.	67
12.3	A proposed answer for the item d).	68
12.4	A hotel fee calculation program for the above requirement.	68
12.5	All the tests thare defined by the independent paths of the HotelFee program.	70
12.6	Program for recruitment participation requirements of D.o.D.. . . .	71
12.7	All of the test cases that are defined by the independent paths of the recruitment program.	72
12.8	A bubble sort algorithm implementation in Java.	73
12.9	Test case implementations of the Bubble Sort class.	74
13.1	A program to calculate a ranged average in a list.	78
14.1	A C++ program that confirms a number is multiple of another.	82
15.1	The implementation of the program that should not supposed to be known.	92
16.1	A Java program that opens the official Selenium website through a Chrome-based browser.	97
16.2	The Selenium dependency for Maven.	97
16.3	Using the driver from a hard-coded path.	98
16.4	Start a Selenium-controlled browser instance.	99
16.5	The complete example to open Google and search for Selenium.	100
17.1	Testing Moodle with a few warm-up tests.	104
A.1	Several examples of XPath from Elmasri et al..	109

SOFTWARE TESTING



1 Basic Concepts of Testing

1.1 Software Quality and Testing

Software testing is the process of evaluating and verifying that the complete software product or parts of it work and perform as specified by the customer’s requirements (SRSD). This process helps in detecting the bugs during the development stage, hence reducing the cost of development and improving performance. In addition, the testing process plays a vital role in achieving and assessing the quality of a software product [1].

Quality assessment activities are discussed in two categories, namely, static analysis and dynamic analysis. Static analysis is based on SRSDs, Software Design Documents (SDD), models, and typically source code. Code review, walkthrough, and inspection are the most common static analysis techniques. The actual execution of the code is not involved in static analysis but rather the source code, byte code, or application binaries are included in the analysis. Static analysis is the more thorough approach and may also prove to be more cost-efficient with the ability to detect bugs at an early phase of the SDLC. Dynamic analysis of a software product involves the actual execution of the code in an attempt to detect bugs. During the dynamic analysis functionality and performance of a program are also observed. Static analysis and dynamic analysis approaches are complementary to improve the quality of the software product and can be performed in a synchronized and/or planned manner.

In software testing, verification and validation (V&V) are two similar abstract concepts. Software Engineering standards known as IEEE/ISO/IEC 24765:2017¹ (ISO/IEC/IEEE International Standard - Systems and Software Engineering - Vocabulary) defines verification and validation as the process of determining whether the requirements for a system or component are complete and correct, the products of each development phase fulfills the requirements or conditions imposed by the previous

- 1.1 Software Quality and Testing 3
- 1.2 Error, Fault, Defect and Failure 4
- 1.3 Objectives of Testing 5
- 1.4 Test Levels in the V-Model 5
- 1.5 White-box and Black-box Testing 6
- 1.6 Testing Activities 6
- 1.7 Problems 7

1: The standard can be found in <https://www.iso.org/standard/71952.html>.

phase, and the final system or component complies with specified requirements. In the software world, it is common to define verification process as an answer to the questions, "Are we building the product right?", and validation process as an answer to "Am I building the right product?".

Code reviews, inspections, walkthroughs, and low level tests such as unit testing, integration testing, static checking of SRSD and SDD and files are some of the activities for verification, and applying various testing techniques such as acceptance testing, usability testing and other non-functional testing are some of the validation activities. Some of the testing methods such as beta testing, regression testing can be listed as activities at the intersection of V&V. In this book, the testing activities mentioned here will be explained in detail.

1.2 Error, Fault, Defect and Failure

Error, fault (bug), defect, and failure are very common terms used in software testing. And these terms are sometimes used interchangeably, albeit incorrectly. To avoid confusion and establish their consistent use in software testing area, ISO/IEC/IEEE 24765:2017 "Systems and software engineering — Vocabulary" standard will be used. The definitions taken from this standard and used in this book are given below.

Error (1) human action that produces an incorrect result, (2) difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition, (3) erroneous state of the system.

Fault (bug) (1) manifestation of an error in software, (2) incorrect step, process, or data definition in a computer program, (3) situation that can cause errors to occur in an object, (4) defect in a hardware device or component, (5) defect in a system or a representation of a system that if executed/activated could potentially result in an error.

Defect (1) imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced, (2) an imperfection or deficiency in a project component where that component does not meet its requirements or specifications and needs to be either repaired or replaced, (3) generic term that can refer to either a fault (cause) or a failure (effect).

Failure (1) termination of the ability of a system to perform a required function or its inability to perform within previously specified limits; an externally visible deviation from the system's specification.

1.3 Objectives of Testing

The foremost objective of Software Testing is to improve the quality of the product by detecting errors and removing faults made during the development phase of SDLC.

Providing quality products is the ultimate goal of testing. Customer satisfaction is ensured and as a result, the competitive power of the software company is increased.

In order to comply with these goals, software test teams are established and the product is tested using different methods at each stage (level) of development. These methods of software testing will be elaborated in the chapters that follow.

1.4 Test Levels in the V-Model

Software testing is applied at different levels of development. Considering the so-called V-model, the levels of testing can be identified as unit (component), integration, system, and acceptance testing.

While all software development methodologies have different approaches, product development tasks will be performed in the order of (1) requirements gathering and analysis, (2) high-level design, (3) comprehensive design, and (4) coding. These tasks match the software testing levels mentioned above as shown in Figure 1.1. Unit, Integration and system tests are usually performed by the developers, and acceptance tests by the customer in collaboration with the software developers.

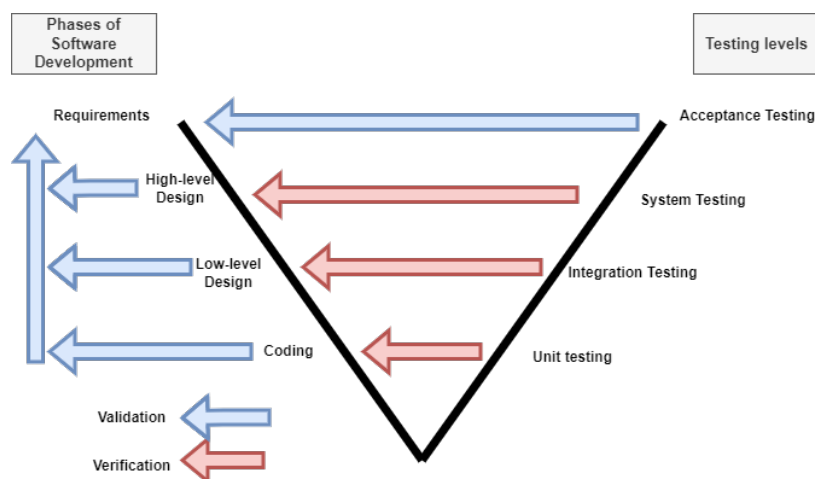


Figure 1.1: Validation and Verification with the V-model

In unit testing, individual units/components such as functions, procedures (in developments with procedural languages) and methods and classes (in developments with object-oriented languages) are tested. Following the successful completion of the unit tests, related units and components are systematically brought together and subjected to integration tests. The objective of integration testing is to build a stable product configuration ready for system tests.

System level testing includes a wide spectrum of testing, such as functionality testing, security testing, robustness testing, load testing, stability testing, stress testing, performance testing, and reliability testing [2]. After the successful completion of system level testing, the product is delivered to the customer. The customer performs a series of tests, known as acceptance testing. The main objective of acceptance testing is to confirm that the system meets the agreed-upon criteria (functional and non-functional requirements), identify and resolve discrepancies, if there are any and determine the readiness of the system for live operations. The final acceptance of a system for deployment is conditioned upon the outcome of the acceptance testing. The acceptance test team produces an acceptance test report which outlines the acceptance conditions.

Another test level applied at the first three levels is the regression test. This test is performed whenever a part of the system (unit, component, or module) is replaced or modified. The main idea of regression testing is to determine if the change creates a side effect (new bugs) on other parts of the software that have already been completed and passed the tests.

In regression testing, new tests are not designed. Instead, tests are selected, prioritized, and executed from the existing pool of test cases to ensure that nothing is broken in the new version of the software [2].

1.5 White-box and Black-box Testing

White-box (glass box, clear box, transparent box) testing is a low-level testing technique which deals with the internal working of the software system. During white-box testing, assignment and predicate statements, and the branches and execution paths are considered and checked for possible faults. On the contrary to white-box testing, black-box (data-driven, functional) testing does not care about the internal workings of the code but deals with the overall behaviour and functionality of the parts of a code and as such can be applied at all levels of testing.

White-box testing is suitable at unit and integration levels, whereas black-box testing is ideal at system and acceptance levels. Programming knowledge is necessary for white-box testing. On the contrary, for black-box testing, this is not essential.

1.6 Testing Activities

Main testing activities are planning, analysis, design, implementation, and execution. Testing team (or developer) comes up with some requirements to be tested. Test planning considers the team management, cost and the duration of the testing process, and some other related quality assurance metrics. Analysis activity involves understanding the nature of tests to be conducted, complexity of the test, and the risks involved in the execution of the tests. Depending on the tasks to be completed, and the input data required, tests are designed. At the test implementation step tests are chosen and prioritized. Software test execution is the last activity to perform. Test results are systematically collected and reported along with the required testing metrics.

A test plan can have more than one test suite. Each test suite has one or more related test cases. A test case (TC) is usually defined as a pair of **<input, expected outcome>**. If a code under test is to compute the absolute value of a real number, say, x , then one can list three test cases as follows:

- ▶ TC1: <0.0, 0.0>
- ▶ TC2: <-26.4, +26.4>
- ▶ TC3: <+14.0, +14.0>

A test case is prepared using the requirements and functional specifications, source code, and input/output domains. Expected outcome can be as simple as a single numerical value, a more complex data type like audio, photo or video, a state change or a set of values which needs to be interpreted together for the outcome to be valid [2].

Another important concept in testing is test oracle. A test oracle (or simply oracle) is a mechanism for determining whether a test has passed or failed. A test oracle contains two essential parts: oracle information that represents expected output; and an oracle procedure that compares the oracle information with the actual output [3]. By varying the level of detail of oracle information and changing the oracle procedure, a test designer can create different types of test oracles. An oracle will respond with a pass or a fail verdict on the acceptability of any test sequence for which it is defined. When executing a test case, the correctness of the implementation is confirmed by an oracle.

In most testing environments, test oracle is the human who designs the test [4]. A simple example of a test oracle is given below².

Given a list of 3 integers, 11, 23, and 19, the expected value from a function, say, `computeMax()`, which returns the maximum is 23. In this case, obviously, the test oracle is easy to identify.

- ▶ `int expected = 23;`
- ▶ `int actual = computeMax(11, 23, 19);` `computeMax` is a method which returns the maximum of the three numbers.
- ▶ `assertEquals(expected, actual);`

Note that `assertEquals()` is a method in the JUnit testing library to compare two objects for equality. This method will be introduced during laboratory sessions.

2: Detailed information about test oracle can be found in the link: <https://bit.ly/3K9iqZj>

1.7 Problems

1. Create test cases for a computer program to find the positive square root of a real number.
2. Discuss the difference between white-box and black-box testing strategies.
3. What is a stateless software system? Give an example.
4. ATM is an example of a state-oriented system. Create a test case for withdrawing money from a bank account. First define the input, and write down the expected output(s) during the transaction.
Hint. In this case, a test case will contain more than one <input, expected> pairs and input may be choosing an item from a menu.

5. What is agile software testing?
6. What are the main differences between classical testing and agile testing approaches?
7. What is Test Driven Development (TDD)?
8. What is a test oracle? Explain by using a simple testing activity.
9. Differentiate between verification and validation. Describe various verification and validation methods.
10. What is the main difference between inspections and walkthroughs?



2 Unit (Component) Testing

2.1 Unit Testing Basics

First level of testing is unit testing. Unit (component) is a manageable piece of code which may produce a result, perform a computation and return its result(s) to its caller, or may call some other function or method. In procedural programming, a unit could be an individual function or procedure (subroutine). In an object-oriented setting, a unit could be an individual method or even a class.

Unit testing is a low-level testing technique by means of which individual units are tested to determine if they produce expected outcomes or not. All assignment statements and predicates (logical expressions) in the code must be run, as well as all paths, while achieving the expected output with the unit test. In general, unit testing activity can be described as a type of testing process used to verify and validate a specific unit of the code for its correctness to cover the coding standards, functionality, integration, code coverage, security features, compatibility, performance, and so on. Unit testing is done by the person who develops the unit. The following unit examples are written in C and Java.

2.1 Unit Testing Basics 9

2.2 Static Unit Testing – Code
Reviews 10

2.3 Dynamic Unit Testing . . . 14

2.4 Mutation Testing 14

2.5 Debugging 18

2.6 Unit Test Automation . . . 19

2.7 Problems 19

Example 2.1.1 Write a C function solveQuadratic to solve the quadratic $ax^2 + bx + c = 0$ for the two real roots x1, and x2. Coefficients of the quadratic polynomial is to be provided by the calling program.

```
1 double solveQuadratic(double a, double b, double c) {
2     double disc, x1, x2;
3     disc = b * b - 4.0 * a * c;
4     if (disc < 0) {
5         printf("Discriminant = %f7.4 :no real roots!", disc);
6     } else {
7         x1 = (-b + sqrt(disc) / (2.0 * a);
8         x2 = (-b - sqrt(disc)) / (2.0 * a);
9         printf("x1 = %f7.4, x2 = %f7.4\n", x1, x2);
```

Listing 2.1: A C function that solves the quadratic equation for the two real roots.

```

10     }
11     return (0);
12 }

```

Listing 2.2: Binary search a C implementation.

```

1 // Iterative C Program for Binary Search (GNU GCC v7.1.1)
2 // Adapted from https://www.tutorialspoint.com/codingground.htm
3 // 3 severe bugs are inserted!
4 #include <stdio.h>
5 int BinarySearch(int array[], int left, int right, int element){
6     while (left <= right){
7         int middle = left + (right + left)/2;
8         if (array[middle] == element)
9             return middle;
10        if (array[middle] < element)
11            left = middle - 1;
12        else
13            right = middle + 1;
14    }
15    return -1;
16 }
17 int main(void){
18     int array[] = {11, 14, 17, 21, 28, 43, 41, 52, 56, 70, 76,
19                    82, 99};
20     int n = 13;
21     int element = 99;
22     int index = BinarySearch(array, 0, n-1, element);
23     if(index == -1 ) {
24         printf("Element not found!");
25     }
26     else {
27         printf("Element found at position : %d",index);
28     }
29     return 0;
30 }

```

Unit testing is conducted in two different but complementary phases, namely, static unit testing and dynamic unit testing. In static unit testing, the unit under the test is examined, without actually executing it, to infer conclusions about its functionality, and causes of some possible faults. In dynamic unit testing, a program unit is actually executed and its outcomes are recorded and analysed.

2.2 Static Unit Testing – Code Reviews

Static unit testing is accomplished by using several code review techniques. Code review is not restricted to units. In practice, during this process, completed parts of the code (unit, module, or component) are reviewed using inspection and walkthrough techniques. Inspection

is a step by step peer group review of a work product (including a document), with each step checked against predetermined criteria [5]. Walkthrough is a similar review technique in which the developer leads the team through a manual or simulated execution of the product using predefined scenarios [6].

The main goal of the code review is to review the code in an attempt to find the errors in the code before it is passed on to another activity (e.g., integration testing). The code review should be performed in a systematic manner by executing planned activities. To facilitate the process a review team is established. Review team is usually made up of members who will assume the roles of facilitator, code developer (author), presenter, record keeper, reviewer, and observer. The general guidelines for performing code review consists of six steps [2].

Readiness Unit developer assures that the unit is ready for inspection.

Preparation Each reviewer goes over the code and the related documents. A list of potential CR is prepared by the reviewers.

Examination and Producing a Change Request (CR) Document In this step, the author, presenter, the record keeper, and the moderator are involved and, if needed, a CR document is prepared and passed on to the next step.

Rework The CR list that needs to be resolved is forwarded to the developer.

Validation The rework done by the developer is validated by one of the software team members. The validation process involves checking the improvements in the code as suggested by the other group members.

Reporting If all CRs are fulfilled, a detailed report is prepared and shared with the team members.

To facilitate the code review process, software teams use predefined checklists. One such checklist is given below¹.

1: This document <https://github.com/mgreiler/code-review-checklist> is protected under the MIT license

Implementation

1. Does this code change do what it is supposed to do?
2. Can this solution be simplified?
3. Does this change add unwanted compile-time or run-time dependencies?
4. Was a framework, API, library, service used that should not be used?
5. Was a framework, API, library, service not used that could improve the solution?
6. Is the code at the right abstraction level?
7. Is the code modular enough?
8. Would you have solved the problem in a different way that is substantially better in terms of the code's maintainability, readability, performance, security?
9. Does similar functionality already exist in the codebase? If so, why isn't this functionality reused?
10. Are there any best practices, design patterns, or language-specific patterns that could substantially improve this code?

11. Does this code follow Object-Oriented Analysis and Design Principles, like the Single Responsibility Principle, Open-close principle, Liskov Substitution Principle, Interface Segregation, Dependency Injection?

Logic Errors and Bugs

12. Can you think of any use case in which the code does not behave as intended?
13. Can you think of any inputs or external events that could break the code?

Error Handling and Logging

14. Is error handling done the correct way?
15. Should any logging or debugging information be added or removed?
16. Are error messages user-friendly?
17. Are there enough log events and are they written in a way that allows for easy debugging?

Usability and Accessibility

18. Is the proposed solution well designed from a usability perspective?
19. Is the API well documented?
20. Is the proposed solution (UI) accessible?
21. Is the API/UI intuitive to use?

Ethics and Morality

22. Does this change make use of user data in a way that might raise privacy concerns?
23. Does the change exploit behavioral patterns or human weaknesses?
24. Might the code, or what it enables, lead to mental and physical harm for (some) users?
25. If the code adds or alters ways in which people interact with each other, are appropriate measures in place to prevent/limit/report harassment or abuse?
26. Does this change lead to an exclusion of a certain group of people or users?
27. Does this code change introduce any algorithm, AI or machine learning bias?
28. Does this code change introduce any gender/racial/political/religious/ableist bias?

Testing and Testability

29. Is the code testable?
30. Does it have enough automated tests (unit/integration/system tests)?
31. Do the existing tests reasonably cover the code change?
32. Are there some test cases, input, or edge cases that should be tested in addition?

Dependencies

- 33. If this change requires updates outside of the code, like updating the documentation, configuration, readme files, was this done?
- 34. Might this change have any ramifications for other parts of the system, backward compatibility?

Security and Data Privacy

- 35. Does this code open the software for security vulnerabilities?
- 36. Are authorization and authentication handled in the right way?
- 37. Is sensitive data like user data, credit card information securely handled and stored? Is the right encryption used?
- 38. Does this code change reveal some secret information (like keys, usernames, etc.)?
- 39. If code deals with user input, does it address security vulnerabilities such as cross-site scripting, SQL injection, does it do input sanitization and validation?
- 40. Is data retrieved from external APIs or libraries checked accordingly?

Performance

- 41. Do you think this code change will impact system performance in a negative way?
- 42. Do you see any potential to improve the performance of the code?

Readability

- 43. Was the code easy to understand?
- 44. Which parts were confusing to you and why?
- 45. Can the readability of the code be improved by smaller methods?
- 46. Can the readability of the code be improved by different function/method or variable names?
- 47. Is the code located in the right file/folder/package?
- 48. Do you think certain methods should be restructured to have a more intuitive control flow?
- 49. Is the data flow understandable?
- 50. Are there redundant comments?
- 51. Could some comments convey the message better?
- 52. Would more comments make the code more understandable?
- 53. Could some comments be removed by making the code itself more readable?
- 54. Is there any commented out code?

Experts Opinion

- 55. Do you think a specific expert, like a security expert or a usability expert, should look over the code before it can be committed?
- 56. Will this code change impact different teams? Should they have a say on the change as well?

The code review checklist list above is comprehensive enough to be used at different levels of testing. However, for unit testing, it would be more appropriate to use a modified and shortened version of the list.

Test Driver
A test driver is simply a program (main) that calls the unit under test. Test input data is provided by the test driver, unit is called, and the results are returned from the unit and assessed (test fails/passes) by the test driver.

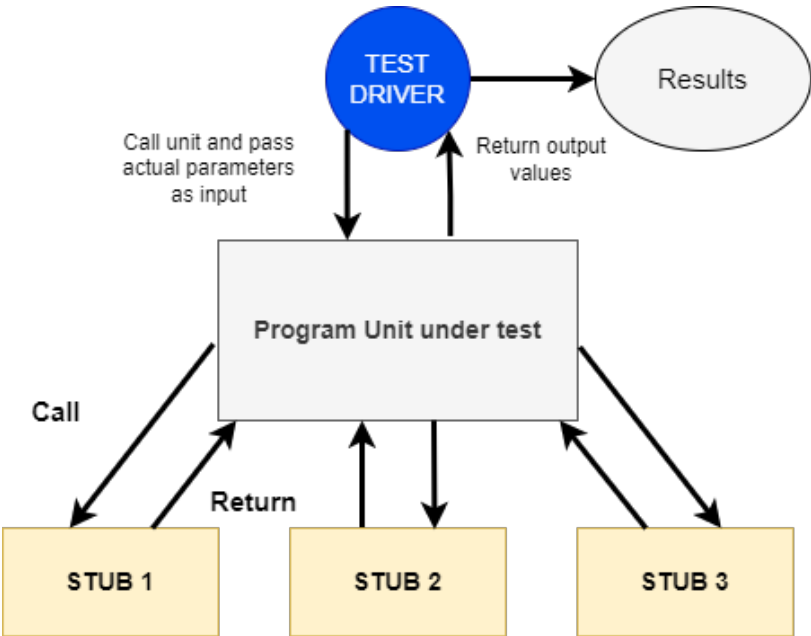
Stub
A stub as mentioned before, is a dummy function which replaces the actual unit called by the unit under test. A stub returns a preassigned or decided value (e.g., null, 0, or another expected result from the actual unit) so that the unit under test can resume its execution in a meaningful way.

Figure 2.1: Validation and verification with the V-model.

2.3 Dynamic Unit Testing

In dynamic unit testing, the unit under investigation is executed in order to detect the errors. An execution environment is created by writing a test driver and compiling it together with the actual unit. If the unit under the test invokes (calls) some other units, dummy ones called stubs are created to emulate the called units.

It is important to note that, the test driver and the stubs are retained and reused in the future in regression testings if required. In Figure 2.1, a dynamic unit testing environment is illustrated.



Test case generation (manual or automatic) is one of the main issues of software testing research. In unit testing, the main concern is to detect the faults (bugs) in the assignment and control statements, predicates (logical expressions in the control statements), and the flow of data from one statement to another. Test input selection is mainly based on Control Flow Testing, Data Flow Testing, and Functional Program Testing techniques. These techniques will be discussed in the chapters to follow.

2.4 Mutation Testing

Mutation testing is a white-box testing technique to measure the adequacy of test cases [7]. The idea in mutation testing is to test the code by modifying/altering the program intentionally by applying one of the there mutation types, and observe how the code behaves when tested with the same test data. Each modified code is known as a mutant.

There are three types of mutation:

Value mutation Change one constant to a larger value or to a smaller value.

Statement mutation Change the statements by removing/deleting or modifying the line.

Decision mutation Change the decisions/conditions to check for the design errors. Typically, one changes the arithmetic operators (e.g., -, +, /, *) and mutating all relational operators (==, !=, <, <=, >, >=) and logical operators (AND, OR, NOT)).

A mutant is said to be killed if the execution of a test case causes it to fail. Some mutants are equivalent to the given program producing the same output as the original one. The mutants still alive are survived mutants and need to be killed by adding additional tests to the test suite. A mutation score, MS , for a set of test cases is defined as the percentage of nonequivalent mutants killed by the test suite. If K is the number of killed mutants, N is the total number of mutants, and E is the number of equivalent mutants, then $MS = \frac{K}{N-E} * 100\%$. The test suite is mutation adequate if the mutation score is 100%.

Example 2.4.1 The following recursive C program prints the factorial of integers from 0 to 10. Create two mutants by applying value, and decision mutations and compute the mutation score.

```
1 #include <stdio.h>
2 int fact (int);
3
4 main(){
5     int k;
6     for(k = 0; k <= 10; k++)
7         printf("%2d! = %4d\n", k, fact(k));
8 }
9 int fact(int num){
10     if (num <= 1)
11         return (1);
12     else
13         return(num * fact(num - 1));
14 }
```

Output is a list of factorial values of numbers from 0 to 10.

```
1 #include <stdio.h>
2 int fact (int);
3
4 main(){
5     int k;
6     for(k = 0; k <= 10; k++)
7         printf("%2d! = %4d\n", k, fact(k));
8 }
9 int fact(int num){
10     if (num >= 1)
11         return (1);
12     else
13         return(num * fact(num - 1));
14 }
1 #include <stdio.h>
```

Listing 2.3: A recursive C program to compute and print 0!, 1!, ..., 10!

Listing 2.4: Mutant 1: (Decision mutation - Change num <= 1 to num < 1)

Listing 2.5: Mutant 2: (Value mutation - Change fact(num - 1) to fact(num - 100))

Listing 2.6: Mutant 3: (Decision mutant
– Change if (num <= 1) to if (num < 1))

```

2 int fact (int);
3
4 main(){
5     int k;
6     for(k = 0; k <= 10; k++)
7         printf("%2d! = %4d\n", k, fact(k));
8 }
9 int fact(int num){
10     if (num <= 1)
11         return (1);
12     else
13         return(num * fact(num - 100));
14 }

```

```

1 #include <stdio.h>
2 int fact (int);
3
4 main(){
5     int k;
6     for(k = 0; k <= 10; k++)
7         printf("%2d! = %4d\n", k, fact(k));
8 }
9 int fact(int num){
10     if (num < 1)
11         return (1);
12     else
13         return(num * fact(num - 1));
14 }

```

For the example above the test suite consists of 3 test cases:

- ▶ Mutation1 - Test case 1: <no input, error message >
- ▶ Mutation2 - Test case 2: <no input, <1, 2, 3, 4, 5, 6, 7, 8, 9, 10> > (result is incorrect!)
- ▶ Mutation3 - Test case 3: <no input, correct result>

Test case 1, and Test case 2 fail and therefore said to be killed by these test cases ($K = 2$). However, Test case 3 produces correct result (factorial of numbers from 0 to 10 are printed correctly!) and is considered as an equivalent mutant ($E = 1$). The mutation score in this case is $\frac{2}{3-1} * 100 = 100\%$.

Example 2.4.2 The following C program prints the prime numbers from 2 to n , where the value of n is the input. Create three mutants by applying value, and decision mutations and compute the mutation score.

Listing 2.7: A C program to compute and print primes from 2 to n .

```

1 //Prime numbers from 2 to n
2 #include<stdio.h>
3 main()
4     printf("Enter the value of n>=2:\n");
5     scanf("%d",&n);
6     printf("Prime numbers are\n");
7     for(i=2;i<=n;i++){
8         int k=0;
9         for(j=1;j<=i;j++)
10             if(i%j==0)k++;
11         if(k==2)

```

```

12         printf("%d ",i);
13     }
14 }

```

```

1 //Prime numbers from 2 to n
2 #include<stdio.h>
3 main(){
4     int i,j,n;
5     printf("Enter the value of n>=2:\n");
6     scanf("%d",&n);
7     printf("Prime numbers are\n");
8     for(i=2;i>=n;i++){
9         int k=0;
10        for(j=1;j<=i;j++){
11            if(i%j==0)k++;
12            if(k==2)
13                printf("%d ",i);
14        }
15 }

```

Listing 2.8: Mutant 1: (Decision mutant – Change $i \leq n$ to $i \geq n$)

```

1 //Prime numbers from 2 to n
2 #include<stdio.h>
3 main(){
4     int i,j,n;
5     printf("Enter the value of n>=2:\n");
6     scanf("%d",&n);
7     printf("Prime numbers are\n");
8     for(i=2;i<n;i++){
9         int k=0;
10        for(j=1;j<=i;j++){
11            if(i%j==0)k++;
12            if(k==2)
13                printf("%d ",i);
14        }
15 }

```

Listing 2.9: Mutant 2: (Decision mutant – Change $i \leq n$ to $i < n$)

```

1 //Prime numbers from 2 to n
2 #include<stdio.h>
3 main(){
4     int i,j,n;
5     printf("Enter the value of n>=2:\n");
6     scanf("%d",&n);
7     printf("Prime numbers are\n");
8     for(i=2;i<=n;i++){
9         int k=0;
10        for(j=1;j<=i;j++){
11            if(i%j!=0)k++;
12            if(k==3)
13                printf("%d ",i);
14        }
15 }

```

Listing 2.10: Mutant 3: (Value mutant – Change $k == 2$ to $k == 3$)

For an input value of 19 for n, the original program produces the primes correctly from 2 to 19.

Output: 2, 3, 5, 7, 11, 13, 17, 19

Three test cases are as follows:

- Mutation1 - Test case 1: <19, null > Test fails! No primes are generated.

- Mutation2 - Test case 2: <19, <2, 3, 5, 7, 11, 17> > Test fails! 19 is missing from the list of primes.
- Mutation3 - Test case 3: <19, 5> Test fails! Only one prime number, 5, is generated.

All tests fail, and there are no equivalent mutants. The mutation score in this case is $\frac{3}{3} * 100 = 100\%$.

2.5 Debugging

Debugging is a systematic process of finding and fixing the number of faults (bugs), or defects, in a piece of code so that the software is behaving as expected. For small to medium size software, developers can use brute force approach to find faults by inserting print statements in the neighborhood of the possible fault(s). For large size projects this approach is so tedious and therefore the use dynamic debuggers provided in the Integrated Development Environments (IDEs) should be utilized. A screenshot from Dev-C++IDE debugger of a program is shown in Figure 2.2. Breakpoints are placed at certain lines (red color), and the change in the values of a variable is observed by adding a watch. Next line tab takes the execution to the next statement and the changes are observed on the Debug tab of the IDE.

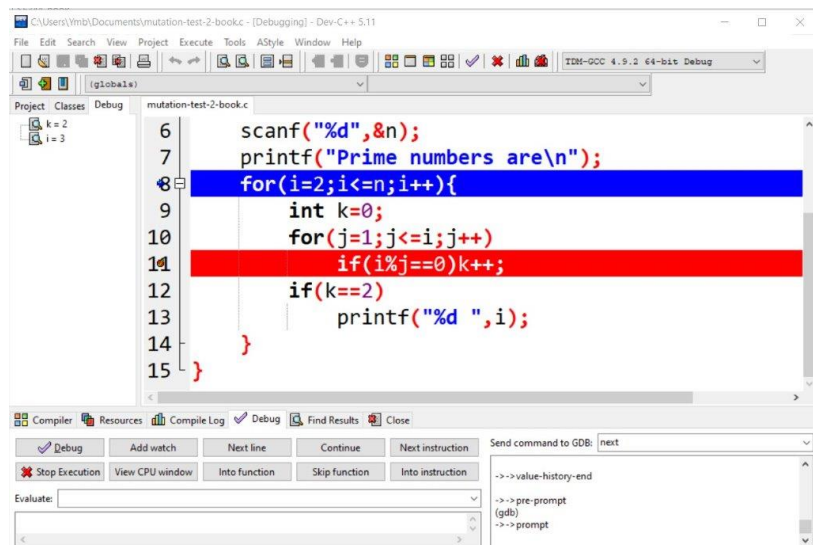


Figure 2.2: A screenshot of Dev-C++Debugger.

The reader is encouraged to learn the use of the debugger on DEV-C++IDE by running and debugging the prime number generation program given above. Eclipse Java IDE used in the laboratory sessions, also offers similar debugging functionality. A screenshot from the debugging option of the Eclipse can be found in Figure 2.3.

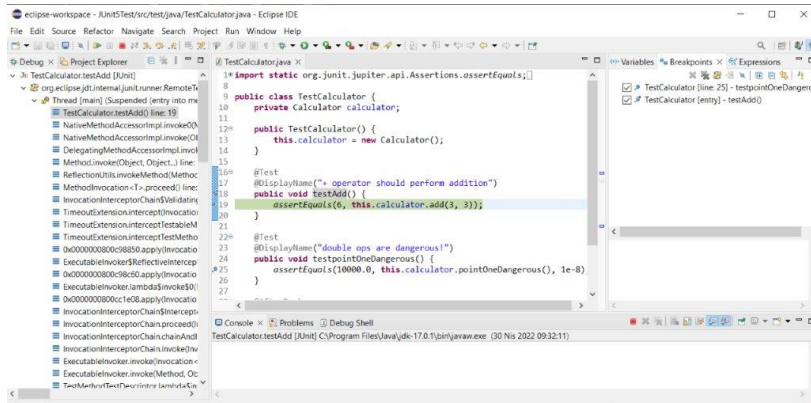


Figure 2.3: A screenshot of Eclipse Debugger.

2.6 Unit Test Automation

Unit tests can be performed manually or with the help of automation tools. JUnit (Java), TestNG (Java), NUnit (.net languages), Jasmine (Javascript), Html Unit (HTML), and Simple Test (PHP) are among the most popular open source unit testing tools and frameworks. In the second part of this book, JUnit tool will be introduced within the Eclipse Java IDE.

2.7 Problems

1. Create a new code review checklist, by modifying the one given in this chapter to facilitate its use for units (functions, procedures, methods, or modules).
2. Write a recursive C program to sort n numbers by the Quicksort algorithm. Create value, decision and statement mutants, and test cases. Compute the mutation score based on your test cases.
3. Explain the difference between static and dynamic unit testing.
4. What is regression testing? At what levels of testing it is utilized?
5. In debugging, what is a breakpoint (toggle point) in a program segment?
6. Use Dev-C++ IDE to debug the prime number generation program given in this chapter to learn the debugging process. Insert breakpoints (toggle points), and use add watch facility to trace the execution.
7. Repeat the same using an Eclipse IDE for C programming language.
8. Explain the difference between debugging and testing.
9. What is static code analysis? Name a few popular open-source static code analyzer.
10. Explain how mutation testing improves the quality of testing.



3 Control Flow Testing

3.1 Introduction

Control flow testing is a software testing strategy that depicts the execution order of the assignment and control statements in a program unit such as a function. This strategy is implemented by developing test cases of a unit and executing and tracing the execution flow. Flow of execution of the assignment and I/O statements in a program unit is sequential and change whenever a control statement (if-then-else, for, switch/case and while) is encountered and executed. A program unit has an entry and an exit point. Commands in a program unit are executed from the command at the entry point to the command at the exit point, depending on the program flow. The sequence of these flow steps is defined as the program execution path. Depending on the number and complexity of control commands, multiple paths can occur in a unit. The paths in the program are shaped depending on the input values in the program unit.

A program unit can have multiple paths. Testing all paths with the input value that determines the path is not a very efficient approach and is costly. There are many approaches to increasing testing efficiency and minimizing its cost. The McCabe cyclomatic complexity analysis is one of them and will be discussed in the following sections.

The control flow testing steps are summarized below:

1. Creation of control flow graph (CFG).
2. Determination of the path to be tested according to the path selection criteria.
3. Creation of necessary inputs and relevant test case for the determined path.

Path selection criteria will be discussed in detail in the next subsections.

3.1 Introduction	21
3.2 Control Flow Graph (CFG)	22
3.3 McCabe Cyclomatic Complexity	23
3.4 Path Selection Criteria	23
3.5 Generating Test Cases	26
3.6 Problems	27

3.2 Control Flow Graph (CFG)

A control flow graph is a directed graph with an entry and an exit point. It is similar to a flowchart and is used to represent the overall flow in a unit. In a CFG, a rectangular node represents a sequential computation encompassing a set of statements in order, a decision node is used for branching (if-then-else), and a circle represents a merge point. A directed edge is used to connect nodes. In order to identify a path uniquely, each node is labeled with a unique integer. A decision node provides branching to the sides of the box by a True (T) or a Yes (Y) label. A complete execution path in a CFG is defined with a set of ordered nodes from 1 to N, where 1 and N are the labels of the entry point, and the exit point respectively. The basic nodes of a CFG is shown in Figure 3.1.

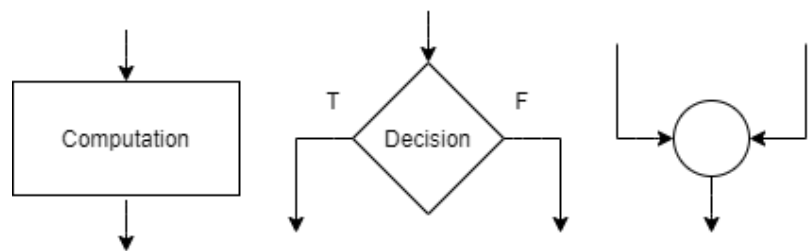


Figure 3.1: Basic nodes (computation, decision and merge) in a CFG.

Using these nodes one can construct CFG elements for the basic programming structures (selection/if-then-else, multi-way branching/switch or case, and iteration/while-do or repeat-until). Some of these CFG constructs are shown in Figure 3.2.

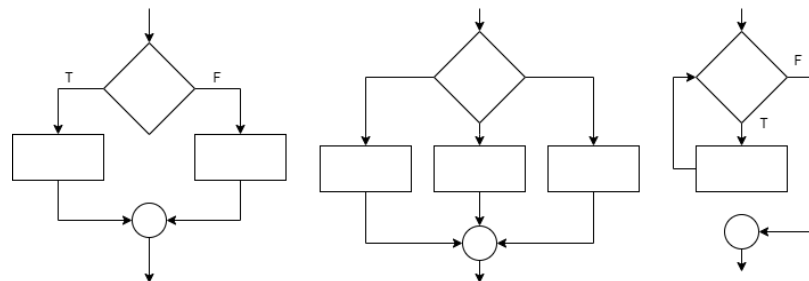


Figure 3.2: if-then-else, switch, and while-do constructs in a CFG.

Other constructs such as repeat-until, nested if, and for loop structures can be constructed in a similar way from these basic constructs.

Listing 3.1: A C program that prints the negative integers in an array of 10 integer values.

Example 3.2.1 For the C program below, create a CFG and identify the paths.

```

1 #include <stdio.h>
2 int main(){
3     int array[10] = {-11, 24, 32, -7, 28, 14, 1, -3, -16, 21};
4     int k = 0, count = 0;
5     printf("Displaying negative integers:\n");
6     while (k <= 10){
7         if (array[k] < 0){
8             printf("%d\n", array[k]);
9             count++;
10        }
11        k++;

```

```

12 }
13 printf("Number of negative integers:%d\n", count);
14 return 0;
15 }

```

A CFG for this program is given in Figure 3.3.

Some of the paths are identified as follows:

- Path1: 1-2-3-4(F)-8-9
- Path2: 1-2-3-4(T)-5(T)-6-7-4(F)-8-9
- Path3: 1-2-3-4(T)-5(F)-7-4(F)-8-9
- Path4: 1-2-3-4(T)-5(T)-6-7-4(T)-5(F)-7-4(F)-8-9

3.3 McCabe Cyclomatic Complexity

The concept of cyclomatic complexity metric of a software unit is first defined by McCabe in 1976 [8]. Given a program unit, and a corresponding flow graph (program graph) G with N nodes and E edges, the cyclomatic complexity of the graph $V(G)$ is computed using the formula $V(G) = E - N + 2$. $V(G)$ is always greater than or equal to 1. Each node in the graph G indicates one or more statements (assignment, I/O or control) in the program and the flow of control is represented by directed edges. The basic structures in the McCabe complexity graph are the same as those in the CFG. However, in the McCabe graph, the decision nodes and assignment statement nodes are represented as circled nodes. The CFG given in Figure 3.3 is redrawn in Figure 3.4 using the McCabe graph notation. For simplicity, some of the nodes representing the initialization (assignment) statements are merged into a single node.

Cyclomatic complexity of this graph is simply $V(G) = E - N + 2 = 8 - 7 + 2 = 3$. Three independent paths are shown below:

- Path1: 1-2(F)-6-7
- Path2: 1-2(T)-3(T)-4-5-2(F)-6-7
- Path3: 1-2(T)-3(F)-5-2(F)-6-7

The McCabe complexity metric guides the tester about the number of paths to be tested by providing the number of independent paths in the program graph. Independent path is defined as a path that has at least one edge which has not been traversed before in any other paths. In the software world, it is generally accepted or tried case that this complexity value is less than ten. It is common practice to simplify or subdivide a unit into manageable units with values greater than ten.

3.4 Path Selection Criteria

The control flow graph for a unit can contain multiple paths. It is desirable to create and run a test case for each of these paths. However, in most cases this is costly and time consuming. The test cases prepared by the unit tester should not run the same path more than once. In such a case, resources will be wasted. After all, the path selection must be made

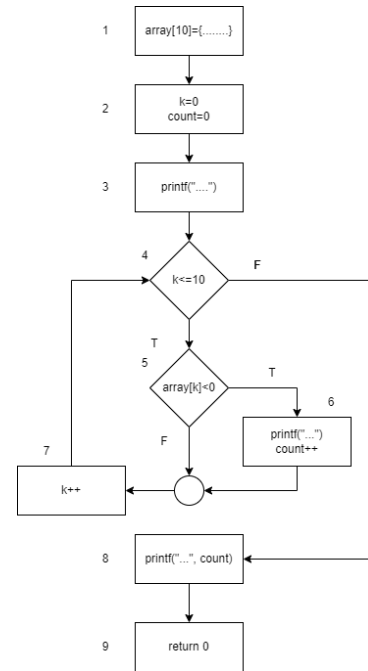


Figure 3.3: A CFG for the C program given in Example 3.2.1.

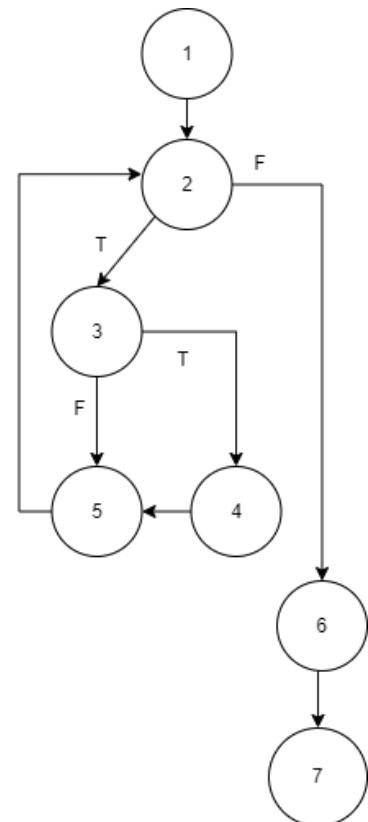


Figure 3.4: McCabe program graph for the C program given in Example 3.2.1.

according to certain criteria. The recommended criteria for path selection in unit tests are listed below:

- ▶ All paths: If selected, one can detect almost all the faults. However, for a unit with very large number of paths, this option is not feasible.
- ▶ Paths that will meet the full statement coverage requirement: Paths that will meet the full statement coverage requirement: In this option, one aims at executing every statement (assignment, I/O, or decision) at least once in the program unit to achieve 100% statement coverage ($= \text{statementscovered} / \text{totalnumberofstatements} * 100$). This is the weakest coverage criterion in comparison with branch and predicate coverage criteria.
- ▶ Paths that will meet the full branch coverage requirement: Paths that will meet the complete branch coverage requirement: Branch coverage ($= \text{numberofexecutedbranches} / \text{totalnumberofbranches} * 100$) means selecting a path that includes the branch. Complete branch coverage means selecting a number of paths such that every branch is included in at least one path.
- ▶ Paths that will meet the full predicate coverage: Predicates are expressions that can be evaluated to a Boolean value, i.e., true or false. A predicate may contain Boolean variables, other variables that are compared with the relational operators $\{ >, <, =, \geq, \leq, \neq \}$, and Boolean function calls (which returns a true or a false value). In a predicate, logical operators such as, AND, OR, and NOT are used to form logical expressions. A clause is a predicate that does not contain any of the logical operators. For full predicate coverage, test cases need to be constructed so that each predicate is evaluated to both true and false.

Listing 3.2: A C program to check if a given string is a palindrome or not.

Example 3.4.1 For the C function `isPalindrome` below, create the corresponding McCabe program graph and identify the paths for 100% statement coverage and complete branch coverage.

```

1 #include <stdio.h>
2 #include <string.h>
3
4 void isPalindrome(char str[])
5 {
6     // Clean up the sentence from special characters and spaces!
7     char cleanstr[200];
8     int left, right, i, j = 0;
9     for (i = 0; i <= strlen(str); i++){
10         if ((str[i] >= 'a' && str[i] <= 'z') || (str[i] >= 'A'
11             && str[i] <= 'Z')){
12             cleanstr[j] = str[i];
13             j++;
14         }
15     }
16     // Set up left and right pointers using j
17     left = 0;
18     right = j - 1;
19
20     // Keep comparing characters while they are same
21     while (right > left){
22         if (cleanstr[left++] != cleanstr[right--])

```

```

22     {
23         printf("%s is not a palindrome\n", cleanstr);
24         return;
25     }
26 }
27 printf("%s is a palindrome\n", cleanstr);
28 }
29 // Examples are taken from https://czechtheworld.com/best-
    palindromes/
30 int main()
31 {
32     isPalindrome("borrow or rob?");
33     isPalindrome("a man, a plan, a canal - panama");
34     isPalindrome("everything we see in the world is the creative
        work of women.");
35     isPalindrome("no, it is open on one position.");
36     return 0;
37 }

```

A McCabe program graph for Example 3.4.1 is given in Figure 3.5. Five independent paths are given as follows:

- Path1: 1-2(F)-6-7(F)-8-12
- Path2: 1-2(F)-6-7(T)-9(T)-10-12
- Path3: 1-2(F)-6-7(T)-9(F)-11-7(F)-8-12
- Path4: 1-2(T)-3(F)-5-2(F)-6-7(F)-8-12
- Path5: 1-2(T)-3(T)-4-5-2(F)-6-7(F)-8-12

When executed, Path2, Path3, and Path5 provide 100% statement coverage. Complete branch coverage is possible by adding the execution of Path4.

Example 3.4.2 For the C program below, create a CFG and identify the paths for 100% statement coverage, and complete branch coverage.

```

1  #include <stdio.h>
2  int main (){
3      int test, i = 2, flag = 0;
4      printf("Enter a positive int >=2:");
5      scanf("%d", &test);
6      while (i <= test / 2){
7          if (test % i == 0){
8              flag = 1;
9              break;
10         }
11         i++;
12     }
13     if (test == 0 || test == 1)
14         printf ("%d is neither prime nor composite.\n", test);
15     else{
16         if(flag == 0)
17             printf ("%d is a prime number.\n", test);
18         else
19             printf ("%d is not a prime number.\n", test);
20     }
21     return(0);
22 }

```

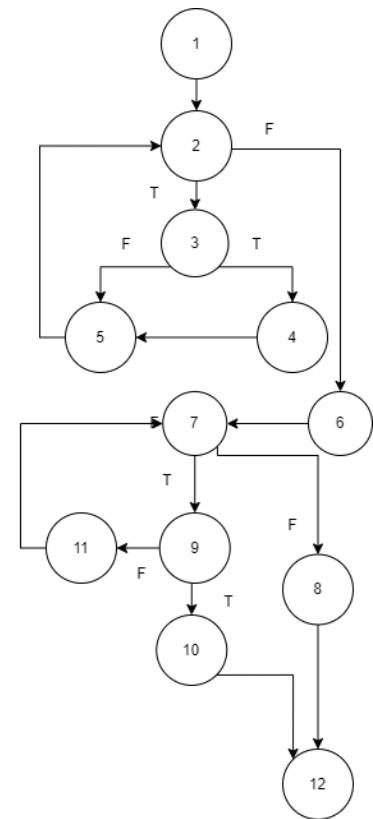


Figure 3.5: McCabe program graph for the isPalindrome program.

Listing 3.3: A C program to test if a number is prime or not.

A CFG for Example 3.4.2 is given in Figure 3.6. Five independent paths are given as follows:

- Path1: 1-2-3(F)-7(T)-12
- Path2: 1-2-3(F)-7(F)-9(T)-10-12
- Path3: 1-2-3(F)-7(F)-9(F)-11-12
- Path4: 1-2-3(T)-4(F)-5-3(F)-7(T)-8-12
- Path5: 1-2-3(T)-4(T)-6-7(T)-8-12

Path2, Path3, Path4, and Path5 provides complete statement and branch coverage. Execution of all these paths provides full predicate coverage.

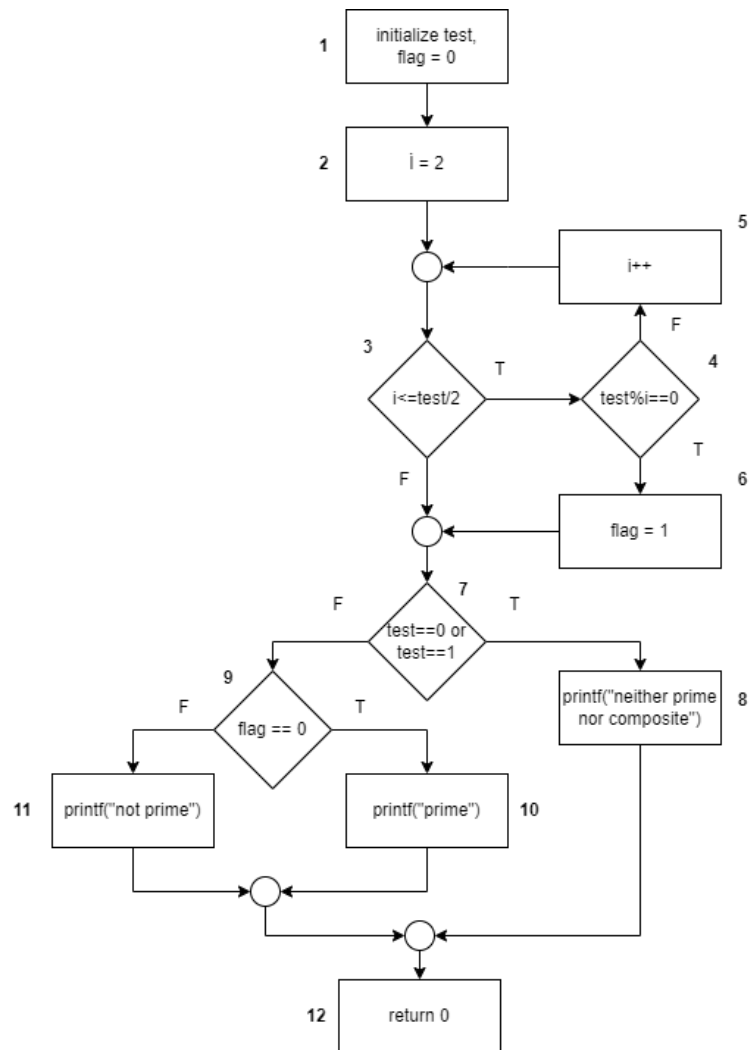


Figure 3.6: CFG for the prime number test program.

3.5 Generating Test Cases

In the preceding section, several path selection criteria are discussed. Identification of a path is an intermediate step in unit testing process. The next step is to choose the input values to force the execution of the selected path.

Test input generation for a given path is accomplished using the following steps [2]:

Table 3.1: Path predicate interpretation.

Node (McCabe)	Node description	Interpretation
1	Input vector <str>, j = 0	
2(T)	i < strlen(str)	0 < strlen(str)
3(T)	((str[0] >= 'a' && str[0] <= 'z') (str[0] >= 'A' && str[0] <= 'Z'))	
4	cleanstr[j] = str[i]	cleanstr[j] = str[0]
5	i++	i = 1
2(F)	i < strlen(str)	1 < strlen(str)
6	left = 0, right = j - 1	left = 0, right = j - 1
7(F)	right > left	right > 0
8	print("str is a palindrome")	
12	return	

- Construct the so-called input vector <a, b, c,> using the input formal parameters of the unit, constants and the values of the global variables. Create a vector of local variables.
- Determine the path predicate, i.e., the set of predicates associated with the path.
- Interpret the path by symbolically substituting operations along the path in order to express the predicates only in terms of the input vector and a vector of constants. The interpreted path is called the path predicate expression.
- Using this expression, generate input data to force the execution of the selected path.

Example 3.5.1 Given the `isPalindrome` function, and the corresponding McCabe program graph, apply the steps above to generate the path predicate, the path predicate expression, and test input data for Path 5: 1-2(T)-3(T)-4-5-2(F)-6-7(F)-8-12

For this path, the path predicate is given below based on the input vector: <str>, and vector of locals: <left, right, i, j>

- For the given path, the path predicate is
 - 2(T): $i < \text{strlen}(\text{str}) \equiv \text{True}$
 - 3(T): $((\text{str}[i] \geq 'a' \ \&\& \ \text{str}[i] \leq 'z') \ || \ (\text{str}[i] \geq 'A' \ \&\& \ \text{str}[i] \leq 'Z')) \equiv \text{True}$
 - 2(F): $i < \text{strlen}(\text{str}) \equiv \text{False}$
 - 7(F): $\text{right} > \text{left} \equiv \text{False}$

For an instance of program execution, a path predicate expression is found and given in the last column of Table 3.1.

A test input to force the execution of Path5 can be easily obtained from the path predicate expression by choosing `str = "X"`, so that `strlen("X")` is 1, `j = 1`, `cleanstr[1] = str[0] = "X"`.

3.6 Problems

1. Create a CFG for the `BinarySearch` function given in Example 2.1.2. Give a list of possible paths.

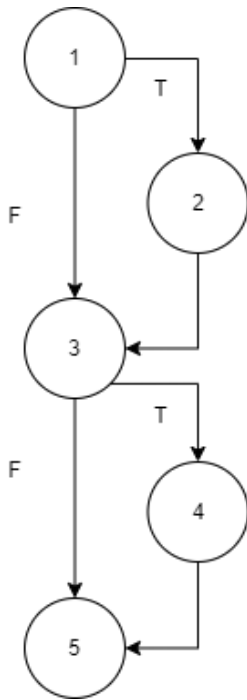


Figure 3.7: McCabe program graph for the C program given in Example 3.2.1.

1: The algorithm is licensed under a [Creative Commons -Attribution -ShareAlike 4.0 \(CC-BY-SA 4.0\)](https://creativecommons.org/licenses/by-sa/4.0/). More information can be found in <https://bit.ly/3ynBEIw>.

Algorithm 1: The pseudo-code of the Euclidean algorithm.

```

input :  $a, b \in \mathbb{Z}^+$ 
output: GCD of  $a$  and  $b$ .
1 begin
2   repeat
3      $R \leftarrow a \bmod b$ 
4      $a \leftarrow b$ 
5      $b \leftarrow R$ 
6   until  $R = 0$ 
7    $GCD \leftarrow b$ 

```

2. Write a C function to sort n integers in descending order using Bubble sort. Draw the McCabe program graph and compute the McCabe cyclomatic complexity for this program and state the number of independent paths accordingly.
3. Write a C function to find the greatest common divisor (GCD) of two integers M , and N by using the Euclidean algorithm. Draw the McCabe program graph, compute the McCabe cyclomatic complexity for this program and determine the independent paths accordingly. The pseudo-code of the Euclidean algorithm is given in Algorithm 1.
4. Write the pseudo-code corresponding to the McCabe program graph given in Figure 3.7. Compute $V(G)$, and determine the independent paths.
5. For Example 3.4.2, find the McCabe cyclomatic complexity (number of independent paths) by converting the CFG given in Figure 3.6 to a program graph.
6. What is an infeasible path in software testing?
7. Consider Algorithm 2¹ below consisting of two successive decision structures. Is there an infeasible path? If yes, express it in terms of statement labels.
8. For the C program given in Example 3.4.2, create a McCabe control graph. Determine the path predicate and path predicate expressions for Path1 and Path4. Generate test inputs accordingly.

Algorithm 2: Infeasible path detection.

```

input :  $height, a, b, c \in \mathbb{Z}^+$ 
1 begin
2   if  $height \leq 26$  then
3      $a \leftarrow 11$ 
4   else
5      $b \leftarrow 13$ 
6   if  $height > 75$  then
7      $c \leftarrow 15$ 

```



4 Data Flow Testing

4.1 Basic Concepts

4.1 Basic Concepts 29

4.2 Data Flow Graph (DFG) . . 30

4.3 Problems 34

The control flow testing strategy uses the program’s control flow to identify the faults in a unit. The main emphasis of control flow testing is to execute the paths on the control flow to cover all statements, branches, and predicates. On the other hand, data flow testing deals with catching faults due to the changes in the variables through incorrect assignments, uninitialized variables, unused variables, and unintentionally making successive assignments of values to the same variable.

Data flow testing can be performed at either static or dynamic levels, as with other unit testing methods. Static data flow testing is done by analyzing the code without running the program unit. Dynamic testing, on the other hand, is performed by running the program unit on the previously determined paths.

For representing these faults systematically, the following three terms are commonly used to identify the status of a variable during the execution of the code.

1. **Defined (d):** Variable is initialized using an assignment, an input statement, or by a parameter in a function call. For example, in `a = b + 1`, `a` is defined.
2. **Referenced** or used (`r`): Variable is used on the right hand side of an assignment statement, or on the left hand side of an assignment as an index of an array, or in a predicate of a branching instruction. For example, in the if statement, `if(list[m] == oldlist[n])`, `list[m]`, `oldlist[n]`, and indices `m`, and `n` are referenced. Similarly, in `a = b + 1`, `b` is referenced (used).
3. Killed or **Undefined (u)**: Variable is used after its termination (e.g, by a `free()` statement in dynamic memory allocation in C), or undefined (uninitialized) before its first use).

The faults caused by the misuse of variables in the data flow are collectively named data anomalies and are classified into three types using the sequences of actions defined by d, u, and r as follows:

1. **Type 1.** Defined and redefined (**dd**): This anomaly is caused as a result of (at least) two successive assignments without using the first one. For example, in the following code segment, salary is defined and then redefined. One possibility is, unintentionally, instead of **totalSalary**, **salary** is used in the first occurrence. Another possibility is a missing statement or code segment between the two. Although this anomaly is generally considered harmless, it should be closely monitored and the cause of its occurrence should be explained.
 - ▶ ...
 - ▶ **salary** = computeGrossSalary(totalWeeklyEarnings);
 - ▶ **salary** = computeNetSalary(totalSalary, taxDeduction);
 - ▶ ...
2. **Type 2.** Undefined but referenced (**ur**): This type of anomaly occurs as a result of uninitialized but later referenced variables. For example, assuming that variable **deduction** is not initialized (u), in the assignment statement, **mygrade = mygrade - deduction**;, the variable **deduction** is referenced (r), and hence causes **ur** type anomaly. This type of fault may have serious consequences and the initialization of a variable should not be left to the grace of a compiler.
3. **Type 3.** Defined but not referenced (**du**): This anomaly is caused by defining (declaring) a variable and undefining it without making any reference to it in the rest of the computation. For example, if the value of **salary** defined by the assignment statement **salary=computeGrossSalary(totalWeeklyEarnings)** is never used in the rest of the code, this type of anomaly is observed. Coding resulting in this type of anomaly is considered a bad programming practice.

Apparently, one can form 6 more binary combinations (**dr**, **ud**, **uu**, **rr**, **rr**, and **rd**) using the actions d, u, and r. These combinations do not cause any data flow anomaly.

4.2 Data Flow Graph (DFG)

In order to detect the anomalies, a path predicate expression should be derived for the path under consideration. For this purpose, the so called data flow graph is utilized. A variable which is defined (d) in the code can be later used (r) on the right hand side of an assignment statement or in a predicate. These uses are names as c-use (computation use), and (predicate use) respectively.

A data flow graph (DFG) is a directed graph consisting of rectangular nodes and connecting edges with the following properties:

- ▶ Each node represents a sequence of data definitions and c-uses.
- ▶ A set of p-uses is associated with each edge.

- The entry node has a definition of each parameter and each non-local variable of the function (subprogram).
- The exit node (return from a function) has an undefinition (kill) of each local variable.
- A null node is used to represent the decision control point (if-then-else, for(), and while-do).

Example 4.2.1 For the C function `isPalindrome` given in Example 3.4.1, draw a DFG.

A DFG for `isPalindrome` is given in Figure 4.2.

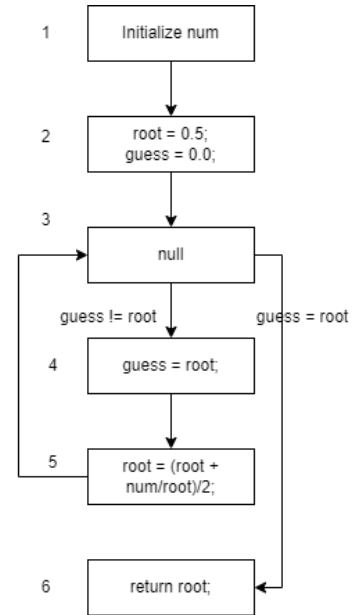
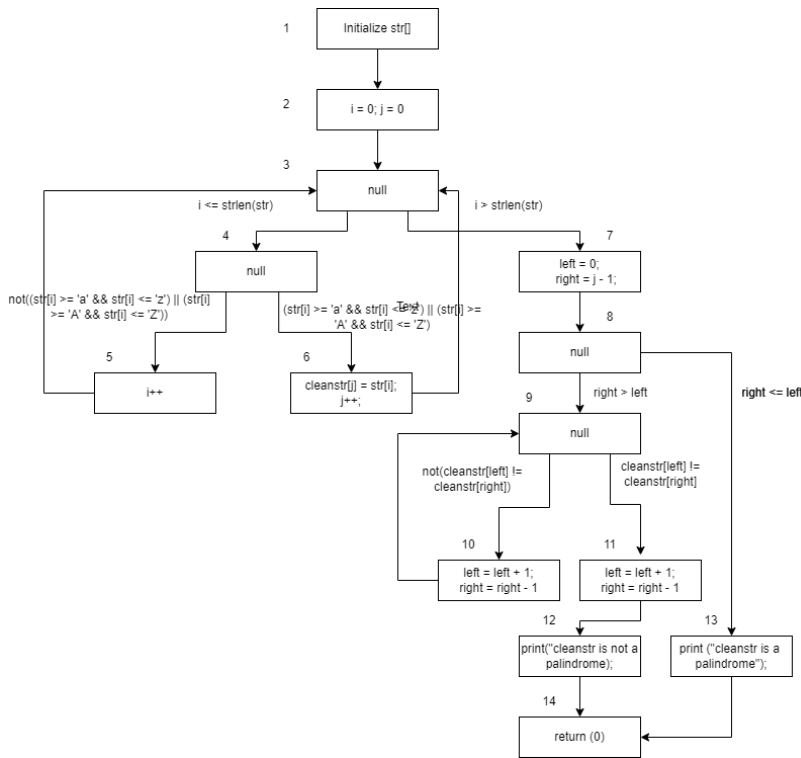


Figure 4.1: Data flow graph for the Newton-Raphson iteration to compute the square root of a real number.

Figure 4.2: Data flow graph for the program given in Example 3.4.1

Example 4.2.2 Square root of a non-negative number can be easily computed using the Newton-Raphson iteration formula. The C function below is an implementation of this technique to compute the square root of `num`. Create a DFG, and identify the definitions, c-uses, and p-uses, of the variables `root`, `guess`, and `num`.

```
1 double squareRoot(double num){
2     double root = 1.0, guess = 0.0;
3     while(guess != root){
4         guess = root;
5         root = (root + num/root) / 2.0;
6     }
7     return root;
8 }
```

Listing 4.1: A C function to compute the positive square root of a real number.

A DFG is given in Figure 4.1. The definitions, c-uses, p-uses, and terminations (kill/undefine) of the three variables used in the function are shown in Table 4.1.

Table 4.1: Definitions, c-uses, p-uses and terminations (undefine) of the variables.

Line	root				guess				num			
	def	c-use	p-use	undef	def	c-use	p-use	undef	def	c-use	p-use	undef
1									✓			
2	✓				✓							
3			✓				✓					
4		✓			✓							
5	✓	✓								✓		
6												
7		✓										
8				✓				✓				✓

Each variable in a function (unit) has a life-cycle which can be expressed as a sequence of events in terms of definition (def), c-use, p-use, and undefinition (undef) of it. For example, for the variable **root**, in the DFG given in Figure 4.1, one can express the events as follows:

- $\text{def}(2)\text{-(p-use}(3)\text{-c-use}(4)\text{-c-use}(5), \text{def}(5))^+\text{-undef}(8)$, where '+' means one or more repetitions of the events in the bracket (node numbers on the DFG).

The goal of data flow testing is to generate test data in such a way as to execute all the paths that may have incorrect or suspicious event sequences (data anomalies). To explain the steps in data flow testing, two definitions will be used [9]:

Definition 4.2.1 A path in the program and on the corresponding DFG is a **definition-use path (du-path)** for some variable x iff it starts with some statement that defines x and ends with a statement that uses (c-use or p-use) x .

Definition 4.2.2 A path in the program and on the corresponding DFG is a **definition-clear (def-clear) path** for some variable x iff it is a du-path for x and the definition statement (node) with which it starts is the only definition statement (node) for that variable in the path.

For example, for the variable **root** in the example above, (2-3), (2-3(T)-4), (2-3(T)-4-5) are du-paths, and (2-3(T)-4) is a def-clear path.

Based on these definitions, one can describe the data flow testing process for a program (unit) as follows:

1. Create a DFG corresponding to the program to be tested.
2. Make a list of all the variables (input, function parameters, and others).
3. For each variable of the program, list all the definitions, c-uses, and p-uses.
4. For each variable, determine a complete path (from the entry node to the exit node) including the def-use, and/or def-clear path for

Node	Description	Interpretation
1	Input vector <num>	
2	root = 1.0, guess = 0.0	
3(T)	guess != root	0.0 != 1.0
4	guess = root	guess = 1.0
5	root = (root + num / root) / 2	root = (1.0 + num / 1.0) / 2.0
6(F)	guess != root	1.0 != (1.0 + num / 1.0) / 2.0
7	return root	

Table 4.2: This caption should be filled with an appropriate sentence.

the variable. Determination of a path is based on different path selection criteria. In this book, only three of them will be mentioned as follows:

- Use of **all du-paths**: According to this criteria, all du-paths are chosen including def-clear paths.
 - Use of **all c-uses**: According to this criteria, all c-use paths are chosen. A **c-use path** for a program with respect to a variable x , is a def-clear path from a definition of x to a c-use of x .
 - Use of **all p-uses**: According to this criteria, all pc-use paths are chosen. A **p-use path** for a program with respect to a variable x , is a def-clear path from a definition of x to a p-use of x .
5. For the selected path, first identify a complete path covering this path, and then obtain the corresponding path predicate, and the path predicate expression (see Section 3.5).
 6. For dynamic data flow testing, to exercise this path, identify a complete path covering the c-use path, generate test data, and run the program.

Example 4.2.3 Considering the function `squareRoot` in Example 4.2.2, and the variable `root`, use **all c-uses** path selection criteria, to obtain a path predicate expression and a complete path covering the c-use path. Then, find a test input data to force the execution of this path.

By Definition 4.2.2, (2-3(T)-4) is a def-clear and a c-use path with respect to the variable `root`. The path predicate is `guess != root` \equiv True. A complete (aggregate) path covering (2-3(T)-4) is (1-2-3(T)-4-5-3(F)-7). Interpretation of this path is given in Table 4.2.

Tablo caption'ına place-holder koydum.

A test input data to force the execution of this path is constructed as follows:

- num = 1.0
- root = 1.0
- guess = 0.0

Observe that, for this input set, the body of the while loop is executed only once and the square root is obtained in one iteration of the Newton-Raphson method.

4.3 Problems

1. Create a table of all possible binary combinations of d, u, and r actions for data flow anomalies. Give an example for each combination and state whether the combination causes an anomaly or not.
2. In C programming language, dynamic memory allocation and deallocation are possible using standard library functions **malloc()**, **calloc()**, **free()** and **realloc()**. Give an example for each of these library functions.
3. Give an example of a Type1 data flow anomaly using the library functions given in the previous Exercise.
4. Give an example of a Type2 data flow anomaly using the library functions given in the previous Exercise.
5. Give an example of a Type3 data flow anomaly using the library functions given in the previous Exercise.
6. For the DFG given in Figure 4.2, make a table consisting of all the variables and their uses (def, undef, c-use, and p-use).
7. Using the table created in the previous question, use **all du-paths** criteria to generate test data.
8. Repeat the same for the **all c-uses** criteria to generate test data.
9. Repeat the same for the **all p-uses** criteria to generate test data.
10. Considering the function `squareRoot` given in Example 4.2.2, and the variable `root`, use **all p-uses** path selection criteria, to obtain a path predicate expression and a complete path covering the p-use path. Then, find a test input data to force the execution of this path.



5 Integration Testing

5.1 Objectives

5.2 Types of Interfaces and Interface Errors

5.3 Integration Techniques

5.3.1 Incremental Approach

5.3.2 Top-down Approach

5.3.3 Bottom-up approach

5.3.4 Sandwich and Bing-bang Approaches

5.4 Problems

5.1	Objectives	35
5.2	Types of Interfaces and Interface Errors	35
5.3	Integration Techniques .	35
5.3.1	Incremental Approach . .	35
5.3.2	Top-down Approach . . .	35
5.3.3	Bottom-up approach . . .	35
5.3.4	Sandwich and Bing-bang Approaches	35
5.4	Problems	35

6.1 System Test Taxonomy

6.2 Basic Tests

6.3 Functionality Tests

6.4 Robustness Tests

6.5 Interoperability Tests

6.6 Performance Tests

6.7 Stress Tests

6.8 Scalability Tests

6.9 Reliability Tests

6.10 Regression Tests

6.11 Problems

6.1 System Test Taxonomy . . 37

6.2 Basic Tests 37

6.3 Functionality Tests 37

6.4 Robustness Tests 37

6.5 Interoperability Tests . . . 37

6.6 Performance Tests 37

6.7 Stress Tests 37

6.8 Scalability Tests 37

6.9 Reliability Tests 37

6.10 Regression Tests 37

6.11 Problems 37

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

7.1 Howden's Approach

7.2 Pairwise Testing with Orthogonal Arrays

7.3 Orthogonal Array Generation with DEVELVE

7.4 Equivalence Class Partitioning (ECP)

7.5 Boundary Value Analysis

7.6 Cause-Effect Graphs

7.7 Decision Tables

7.8 Error Guessing

7.9 Problems

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

7.1 Howden's Approach	39
7.2 Pairwise Testing with Orthogonal Arrays	39
7.3 Orthogonal Array Genera- tion with DEVELVE	39
7.4 Equivalence Class Partition- ing (ECP)	39
7.5 Boundary Value Analysis .	39
7.6 Cause-Effect Graphs	39
7.7 Decision Tables	39
7.8 Error Guessing	39
7.9 Problems	39

8.1 Measurement vs Metrics

8.1 Measurement vs Metrics . 41

8.2 Product Metrics for Testing

8.2 Product Metrics for Testing 41

8.3 Process Metrics for Testing

8.3 Process Metrics for Testing 41

8.4 Problems

8.4 Problems 41

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

9.1 Type of Acceptance testing

9.1 Type of Acceptance testing 43

9.2 Acceptance Criteria

9.2 Acceptance Criteria 43

9.3 Test plan

9.3 Test plan 43

9.4 Problems

9.4 Problems 43

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

LABORATORY STUDIES AND EXERCISES

Software testing and test design are very important and inseparable parts of the software development process. The application of testing methods is as important as the theory of testing. This manual aims to be a supplementary document to the theoretical lectures on software testing. It can be used as an introductory guide to JUnit Jupiter API or can be followed as a text for laboratory studies. Java is chosen as the primary programming language for the exercises. Each section has two essential parts; the first one introduces the focused subject and the second one presents some exercises on the subject. Sections are designed to cover a 10-weeks semester schedule. However, some sections can be taught as two-week sections.

10.1 Brief Summary of Java

Java is an object-oriented programming language created by Sun engineers James Gosling, Mike Sheridan, and Patrick Naughton in 1991. Java programs are compiled into a special bytecode before being interpreted by Java Virtual Machine (JVM). This bytecode can be thought of as a high-level version of low-level machine languages such as Assembly. JVM interprets the bytecode and makes the system calls and other necessary operations on behalf of the running bytecode. This additional layer sometimes causes Java programs to be a little bit slower than their C/C++ counterparts. Because of this "compilation then interpretation" stage Java can be classified as a hybrid programming language.

One advantage of using a JVM is that every operating system has its own implementation of JVM. Therefore, the same Java program can be run in multiple operating systems without any change in the code. This is why Java is known as a platform-independent language.

10.1.1 Java Terminology

Before moving on to more complex topics, let's discuss the most common terminology of Java. Some of the below terms are discussed before and some are new.

Java Virtual Machine (JVM) A written program is first converted to a low-level language called bytecode by a program called *javac*. Then, JVM interprets the bytecode and makes the operation on behalf of it.

Bytecode As discussed previously, it is a low-level language model to communicate with JVM. It can be physically found in projects as files with *.class* file extension.

Java Development Kit (JDK) JDK is a complete development environment to develop Java programs. It contains a compiler, Java Runtime Environment (JRE), Java debuggers, Java documentations, etc.

Java Runtime Environment (JRE) JRE is just an environment only capable of running pre-compiled Java programs. One cannot compile Java programs with only JRE installed. JRE includes a browser, JVM, applet supports, and plugins. To be able to run a Java program a computer should have at least JRE installed on it.

Garbage Collector Unlike C/C++, one cannot delete objects manually in Java. This responsibility is taken care of automatically by a special program called *Garbage Collector*. Garbage collector detects the objects which have not been referenced anymore and deletes them to recollect the memory occupied by them.

ClassPath The classpath is the file path where the Java runtime and Java compiler look for .class files to load. If you want to add external libraries, then you must add them to the classpath.

10.1.2 An Overview of a Hello World Program

1: Curious readers can find the full list on GeeksForGeeks website <https://www.geeksforgeeks.org/introduction-to-java/?ref=lbp>.

Java has many features. Some of them are object-oriented, multithreaded, allows sandbox execution, and the list goes on¹. The best way to explain some syntax rules is to go through an example. Let's look at the example Java program shown in Listing 10.1.

Listing 10.1: Hello world example written in Java.

```

1 // Demo Java program
2
3 // Importing classes from packages
4 import java.io.*;
5
6 // Main class
7 public class SomeClassName {
8     // Main driver method
9     public static void main(String[] args) {
10
11         // Print statement
12         System.out.println("Hello World!");
13     }
14 }
```

The lines starting with `//` are called *comment* lines. Compilers ignore the comment lines. Comments can be single line or multiple lines. Multiple line comments start with `/*` and end with `*/`. e.g. `/* A multiline comment */`.

The line `import java.io.*;` means that import all the classes of `io` package which also belongs to another package called `java`. This is generally **not** a recommended way of importing packages.

Following the import statement, `public class SomeClassName` defines a class. Here, `public` is an *access modifier*. It defines the places that can access to this class. Then, keyword `class` indicates that we are defining a class with the name `SomeClassName`. In Java, each file can have only one class (except for the inner classes) and the name of the class and the file that holds the class must be the same. Otherwise, Java compiler raises an error.

Each Java program starts from a static method called `main` which takes a String array for command-line arguments. This method must be static because it does not actually belong to the class which defines it and must be called externally by the JVM. In Listing 10.1, this method is defined

as `public static void main(String[] args)`. Here, the access modifier is set to `public`. However, it is not necessary. Followed by the `static`, the return type of the method is set to `void`. This means that the method returns nothing after it is called which makes sense.

In Java, writing and reading operations always work on streams. In this case, by writing `System.out.println("Hello world!");` we want to get the `System.out` stream which is `stdout` pseudo-file (console) and print a newline character terminated string. We can also get the `stdin` pseudo-file to be able to read the inputs from the console by utilizing the `System.in`.

You are now ready to expand your knowledge about Java further with this basic Java introduction. Some important resources to learn Java are [10–12].

10.2 Dependency Management with Maven

Dependency management is a big part of every software development especially if multiple dependencies are involved. Each dependency can also depend on other dependencies and their specific versions. This is a huge problem because sometimes developers need to use an incompatible version as a dependency of the software they are developing which is a dependency of another dependency. These types of problems are hard to solve by humans. Because of this, various helper programs can be used. One of them is Maven, another popular one is Gradle. The list can be long for popular programming languages such as Java.

In this lab, we are going to focus on Maven. Without saying much, one can reach all the detailed information about Maven on Apache Maven Project website*. Let's return to the subject. Each Maven project has the same directory structure shown in Table 10.1.

There can be other directories in the project. A full list can be found in Apache Maven Project website†. The `src` directory holds the source code of the software and other resources such as images, database files, etc. The `target` directory contains the output of a build. The most important file for a Maven project is the `pom.xml` file. This file is an Extensible Markup Language (XML) file to hold all the necessary information about the project and its dependencies. An example `pom.xml` file is shown in Listing 10.2.

Directory/File	Description
<code>src/main/java</code>	Application/Library sources
<code>src/main/resources</code>	Application/Library resources
<code>src/test/java</code>	Test sources
<code>src/test/resources</code>	Test resources
<code>target</code>	Output of the build
<code>pom.xml</code>	Description of the project

Table 10.1: Maven directory layout.

* <https://maven.apache.org/guides/getting-started/index.html>

† <https://bit.ly/34VlPvC>

Listing 10.2: An example pom.xml file.

```

1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="
  http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation
  ="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
  xsd/maven-4.0.0.xsd">
2   <modelVersion>4.0.0</modelVersion>
3
4   <groupId>com.mycompany.app</groupId>
5   <artifactId>my-app</artifactId>
6   <version>1</version>
7
8   <properties>
9     <mavenVersion>3.0</mavenVersion>
10  </properties>
11
12  <dependencies>
13    <dependency>
14      <groupId>org.apache.maven</groupId>
15      <artifactId>maven-artifact</artifactId>
16      <version>${mavenVersion}</version>
17    </dependency>
18    <dependency>
19      <groupId>org.apache.maven</groupId>
20      <artifactId>maven-core</artifactId>
21      <version>${mavenVersion}</version>
22    </dependency>
23  </dependencies>
24 </project>

```

Each pom.xml file is actually an XML Schema Definition (XSD). At the top level, a project element defines the namespace information in its attributes. In the subelements, the version of the project, group id, artifact id, and properties are declared. Following the properties, a complex element called <dependencies> declares the dependencies of the project. There is a wide variety of elements to use inside the project. Full list can be obtained from here[‡].

10.3 Installing Eclipse and Setting up a Maven Project for Testing

Fortunately, most of the time we do not have to deal with constructing the directory structure or writing the pom.xml file. Most Integrated Development Environments (IDE) prepare those structures with project creation wizards and automate the building tasks. You can write Java code even in the simple Notepad program. However, using an IDE greatly reduces the overhead of building and writing software. There are many great choices when it comes to IDEs. In this manual, Eclipse IDE is chosen. To install Eclipse in Windows, just go to the official website[§] and download the download manager and install Eclipse for Java. For Linux, just use the package manager of the distribution that you are using. Notice that, it is assumed that either OpenJDK or Oracle JDK has been already installed

[‡] <https://maven.apache.org/pom.html>

[§] <https://www.eclipse.org/downloads/>

in your computer. Alternatively, you can install Eclipse independently from the distribution via Snap[¶] in Linux or via Chocolatey in Windows 10 or above.

After installing Eclipse, a standard welcome screen should be opened as shown in Figure 10.1. In this screen, close the welcome tab and go to

File » New » Project...

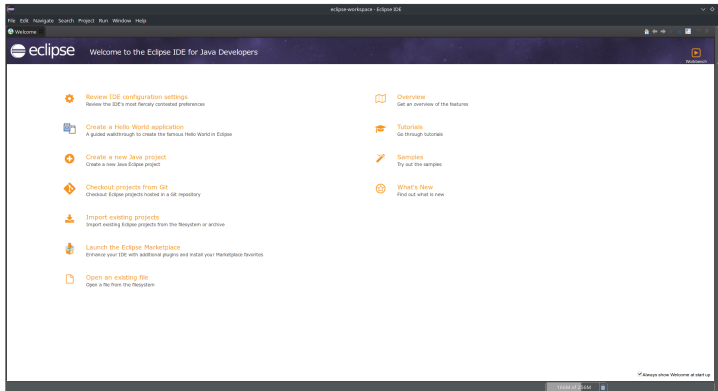


Figure 10.1: Welcome screen of Eclipse.

A new dialog window should be showing up as in Figure 10.2. Search and choose the *Maven Project*. After that, a wizard (shown in Figure 10.3) asks you a few questions about how you want to configure your new project.

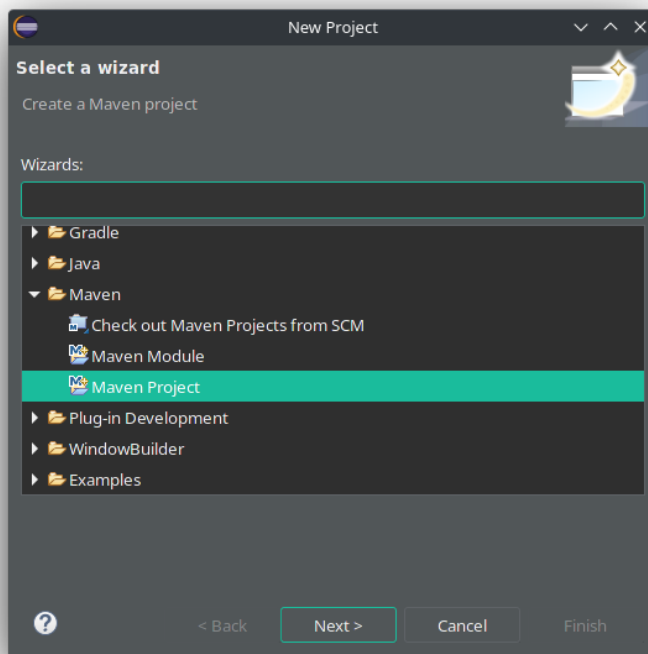


Figure 10.2: Create a new project window.

[¶] <https://snapcraft.io/store>
<https://community.chocolatey.org/packages>

In Figure 10.3, make sure that *Create a simple project* checkbox is checked. This will allow us to skip some unnecessary configuration options in the next pages. Click **Next >** and you should see the dialog window shown in Figure 10.4.

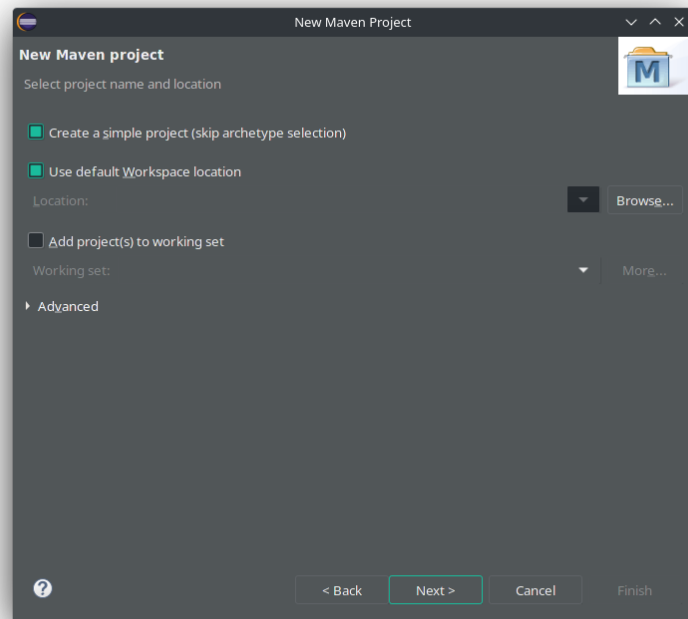


Figure 10.3: Maven project wizard landing page.

In Figure 10.4, there are only two fields that must be filled before clicking to the **Finish** button. The first one is the *Group Id* field. Here, you should write a general package name that normally should hold all of your similar projects. For example, since we are going to write a project specifically for the SE344 lab that is offered at Atılım University, we can write something like this: `edu.atilim.se344`. The first part, `edu`, indicates the top-level domain as on the Internet. Then, it is followed by the company name; in this case, it is our university. Finally, followed by the name of the course. In *Artifact Id*, we just give a name to our build target/executable. In this case, it is `lab1`. When we build the project, our build target will be named like `lab1-0.0.1-SNAPSHOT.jar`. When you have finished filling the necessary fields, click **Finish** button. The new project should be created.

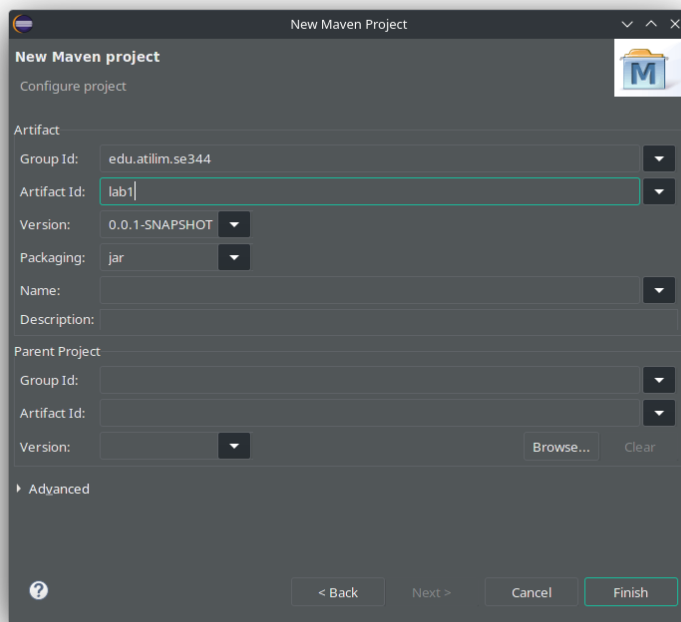


Figure 10.4: Maven project wizard final page.

Before moving on to the next topic, there is a small problem that we need to address. In some versions of Eclipse, Java projects have a default Java version of 1.5. This is a problem for us. Therefore, right-click to the newly created project and click `Properties`. The dialog window shown in Figure 10.5 should be opened. Here, click *Java Compiler* in the list view. Untick the checkbox starting *Use compliance from execution...* and select 1.8 from *Compiler compliance level* checkbox. Click `Apply and Close` button when you have finished with the project properties.

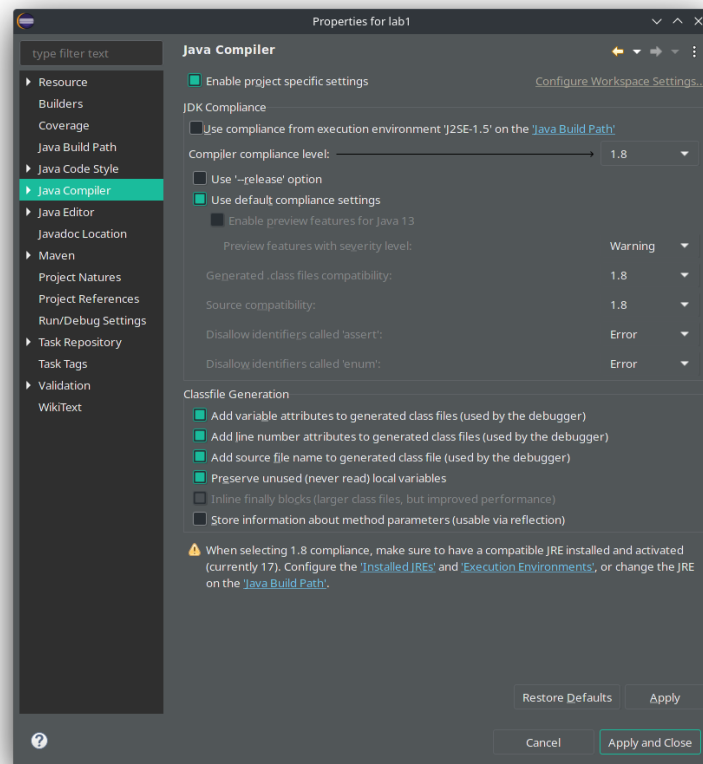


Figure 10.5: Java compiler properties for a Java project.

This concludes our project creation step. Now, we need to add necessary dependencies to our project's `pom.xml` file. In this case, there are two dependencies. They are JUnit Jupiter API and JUnit Jupiter Engine. In the next section, we are going to see how to add such dependencies to our project. Also, we need to make sure that our Maven build system is compatible with JUnit. We will add a plugin to our project to comply with JUnit.

10.4 Introduction to JUnit Jupiter API

JUnit is one of the leading unit testing frameworks for Java. It has a standard API that supplies all the necessary methods and classes for a complete test suite. In fact, it influences many other unit test frameworks in other programming languages. In Java, libraries are added to the projects by adding related `.jar` files to the classpath and importing them into the program. The responsibility of managing those libraries completely belongs to the developer. That is a challenging problem as stated in a previous section. Therefore, using a dependency manager is always a good idea.

In this section, we will add JUnit Jupiter API (a.k.a. JUnit 5) and Jupiter Engine to our previously created project and choose a Maven plugin version that works with the JUnit API. Remember from the previous `pom.xml` examples, each dependency that we want to add to our project must go to a complex element called `<dependencies>`. We will use JUnit

JUnit API and Engine version 5.8.2 in this example. Go to the Maven repository website, search for the package *junit*, and click the version 5.8.2**. In the middle of the page, you should see a code snippet as shown in Listing 10.3. Copy the code snippet and paste it inside the `<dependencies>` element.

```

1 <dependency>
2   <groupId>org.junit.jupiter</groupId>
3   <artifactId>junit-jupiter-api</artifactId>
4   <version>5.8.2</version>
5   <scope>test</scope>
6 </dependency>
7
8 <dependency>
9   <groupId>org.junit.jupiter</groupId>
10  <artifactId>junit-jupiter-engine</artifactId>
11  <version>5.8.2</version>
12  <scope>test</scope>
13 </dependency>

```

Listing 10.3: JUnit Jupiter API and Engine version 5.8.2 dependency elements.

After adding JUnit Jupiter API, we need to indicate a specific version of Maven Surefire Plugin that works with the newest JUnit Jupiter API and Eclipse. In this case, it is the version 3.0.0-M5. Add the code snippet shown in Listing 10.4 to your `pom.xml` file under the top-level `<project>` element.

```

1 <build>
2   <plugins>
3     <plugin>
4       <groupId>org.apache.maven.plugins</groupId>
5       <artifactId>maven-surefire-plugin</artifactId>
6       <version>3.0.0-M5</version>
7     </plugin>
8   </plugins>
9 </build>

```

Listing 10.4: A compatible Maven Surefire Plugin with Eclipse and JUnit Jupiter API v5.8.2..

In the end, your `pom.xml` file should look like Listing 10.5.

```

1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="
  http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation
  ="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/
  xsd/maven-4.0.0.xsd">
2   <modelVersion>4.0.0</modelVersion>
3   <groupId>edu.atilim.se344</groupId>
4   <artifactId>LabManualExercises</artifactId>
5   <version>0.0.1-SNAPSHOT</version>
6
7   <properties>
8     <maven.compiler.source>1.8</maven.compiler.source>
9     <maven.compiler.target>1.8</maven.compiler.target>
10    <exec.mainClass>Main</exec.mainClass>
11  </properties>
12
13  <dependencies>
14    <dependency>
15      <groupId>org.junit.jupiter</groupId>

```

Listing 10.5: `pom.xml` file for future exercises.

** <https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api/5.8.2>

```

16         <artifactId>junit-jupiter-api</artifactId>
17         <version>5.8.2</version>
18         <scope>test</scope>
19     </dependency>
20     <dependency>
21         <groupId>org.junit.jupiter</groupId>
22         <artifactId>junit-jupiter-engine</artifactId>
23         <version>5.8.2</version>
24         <scope>test</scope>
25     </dependency>
26 </dependencies>
27
28 <build>
29     <plugins>
30         <plugin>
31             <groupId>org.apache.maven.plugins</groupId>
32             <artifactId>maven-surefire-plugin</artifactId>
33             <version>3.0.0-M5</version>
34         </plugin>
35     </plugins>
36 </build>
37 </project>

```

As a final note, we prepared a Windows-based all-in-one Oracle VM VirtualBox disk image. Further information about the import process and the details of the image is given in Appendix Chapter B. It has all the packages necessary to run all the exercises in the manual. It can be directly obtained through the PCs in the lab with a 64GB USB flash drive.

JUnit Jupiter API has a very stable and consistent API. There are two essential resources to learn it. The first one is the official JUnit 5 User Guide* and the second one is the official JUnit 5 Java Docs†. In this section, the important assert types of the JUnit 5 is introduced without any meaningful context or formal methods.

11.1 Developing Tests

Let's assume that we are developing a *Calculator* program and we are using the Test-Driven Development approach. We want to test the addition functionality of the calculator. In JUnit, our test case would look like Listing 11.1. In the code, a class called *Calculator* is assumed to exist in the package `edu.atilim.se344`. One of its methods is `add(int a, int b)`; and we are testing this method.

To create a test case, first, we need to create a `.java` file inside `src › test › java`. To do that, right-click to the `src › test › java` and click `New >> Class` from the context menu. The name of the file is not important as long as the same with the name of the class inside. Inside of the file, we need a class. The contents of the class need to comply with some rules.

```
1 import static org.junit.jupiter.api.Assertions.assertEquals;
2
3 import edu.atilim.se344.Calculator;
4 import org.junit.jupiter.api.Test;
5
6 class CalculatorTests {
7
8     private final Calculator calculator = new Calculator();
9
10    @Test
11    void addition() {
12        assertEquals(2, calculator.add(1, 1));
13    }
14 }
```

Listing 11.1: A test case for testing the addition functionality of the *Calculator* class.

The test class cannot be abstract and must have at least one method that is annotated with `@Test`. In our example, *CalculatorTests* class have a constant *Calculator* instance `calculator` and a `void addition()` method that is annotated by `@Test` which causes the method to be automatically called by the JUnit framework. There are lots of annotations that help both JUnit to identify the role of the method and the developer to troubleshoot any problem.

A more complete and standard test class should have some other special methods. Such as a method that is called before any of the other methods, a method that is called before each test case, test cases themselves, a method that is called after each test case, and a method that is called

* <https://junit.org/junit5/docs/current/user-guide/>

† <https://junit.org/junit5/docs/current/api/>

after every test case is called. Such a standard test class is proposed by the official JUnit guide. You can see it in Listing 11.2.

Listing 11.2: A standard test class.

```

1 import static org.junit.jupiter.api.Assertions.fail;
2 import static org.junit.jupiter.api.Assertions.assertTrue;
3
4 import org.junit.jupiter.api.AfterAll;
5 import org.junit.jupiter.api.AfterEach;
6 import org.junit.jupiter.api.BeforeAll;
7 import org.junit.jupiter.api.BeforeEach;
8 import org.junit.jupiter.api.Disabled;
9 import org.junit.jupiter.api.Test;
10
11 class StandardTests {
12
13     @BeforeAll
14     static void initAll() {
15     }
16
17     @BeforeEach
18     void init() {
19     }
20
21     @Test
22     void succeedingTest() {
23     }
24
25     @Test
26     void failingTest() {
27         fail("a failing test");
28     }
29
30     @Test
31     @Disabled("for demonstration purposes")
32     void skippedTest() {
33         // not executed
34     }
35
36     @Test
37     void abortedTest() {
38         assertTrue("abc".contains("Z"));
39         fail("test should have been aborted");
40     }
41
42     @AfterEach
43     void tearDown() {
44     }
45
46     @AfterAll
47     static void tearDownAll() {
48     }
49 }

```

The developer can also assign more readable names to her/his test cases with the `@DisplayName` annotation. It takes a string e.g. `@DisplayName("Some interesting test case")`. It can take all valid Unicode encoded strings even emojis.

11.2 Working with Assert Types

There are many assert types that can be used in various situations. The full list can be found in the documentation[‡]. Most of them are different overloads of the same assert function. The widely used ones are given in Table 11.1.

Method Prototype	Description
<code>assertAll(String, Collection<Executable>)</code>	Assert that all supplied executables do not throw exceptions.
<code>assertArrayEquals(boolean[] expected, boolean[] actual)</code>	Assert that expected and actual boolean arrays are equal.
<code>assertEquals(byte expected, byte actual)</code>	Assert that expected and actual are equal.
<code>assertFalse(boolean condition)</code>	Assert that the supplied condition is false.
<code>assertNotEquals(byte unexpected, byte actual)</code>	Assert that expected and actual are not equal.
<code>assertTrue(boolean condition)</code>	Assert that the supplied condition is true.

Table 11.1: Some of the assert methods that are defined in JUnit API.

Notice that there are a huge amount of overloads of these methods. Since our main objective is not presenting the whole list, only a very small amount of them are projected into the Table 11.1.

11.3 Running and Reporting Tests

Running test in Eclipse is fairly straightforward. You can right click to the current project in the *Package Explorer*. In the context menu, follow the menu option `Run As` `Maven test`. This will trigger the automatic build process of Maven and runs all the tests inside the `src > test > java` path.

A successful test run should produce an output like this:

```

1 [INFO] Scanning for projects...
2 [INFO]
3 [INFO] -----< edu.atilim.se344:lab1
   >-----
4 [INFO] Building lab1 0.0.1-SNAPSHOT
5 [INFO] -----[ jar
   ]-----
6 [INFO]
7 [INFO] --- maven-resources-plugin:2.6:resources (default-resources
   ) @ lab1 ---
8 [WARNING] Using platform encoding (UTF-8 actually) to copy
   filtered resources, i.e. build is platform dependent!
9 [INFO] Copying 0 resource
10 [INFO]
11 [INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @
   lab1 ---
12 [INFO] Nothing to compile - all classes are up to date
13 [INFO]
14 [INFO] --- maven-resources-plugin:2.6:testResources (default-
   testResources) @ lab1 ---
15 [WARNING] Using platform encoding (UTF-8 actually) to copy
   filtered resources, i.e. build is platform dependent!
16 [INFO] Copying 0 resource
17 [INFO]
18 [INFO] --- maven-compiler-plugin:3.8.1:testCompile (default-
   testCompile) @ lab1 ---

```

Listing 11.3: A log output from a Maven test run.

[‡]<https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html>

```

19 [INFO] Nothing to compile - all classes are up to date
20 [INFO]
21 [INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ lab1
22 [INFO] Surefire report directory: /home/tustunkok/eclipse-
    workspace/lab1/target/surefire-reports
23
24 -----
25 T E S T S
26 -----
27 Running lab1.CalculatorTest
28 Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
    0.002 sec
29
30 Results :
31
32 Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
33
34 [INFO]
    -----
35 [INFO] BUILD SUCCESS
36 [INFO]
    -----
37 [INFO] Total time: 1.218 s
38 [INFO] Finished at: 2022-02-03T15:54:36+03:00
39 [INFO]
    -----

```

11.4 Exercises

Assume that the following calculator class is given to you. You are responsible for writing the necessary test cases for the given class. Remember that there might be mistakes in the given implementations throughout all exercises in the manual. Do not take them as exact true cases.

Listing 11.4: A Calculator class implementation in Java.

```

1 package edu.atilim.se344;
2
3 public class Calculator {
4     public int add(int n1, int n2) {
5         return n1 + n2;
6     }
7
8     public int sub(int n1, int n2) {
9         return n1 - n2;
10    }
11
12    public int div(int n1, int n2) {
13        return n1 / n2;
14    }
15
16    public int mul(int n1, int n2) {
17        return n1 * n2;

```

```

18     }
19 }

```

Exercise 11.4.1 Write the individual test cases for each of the methods of the Calculator class.

```

1 import static org.junit.jupiter.api.Assertions.assertEquals;
2
3 import edu.atilim.se344.Calculator;
4 import org.junit.jupiter.api.Test;
5
6 class CalculatorTests {
7
8     private final Calculator calculator = new Calculator();
9
10    @Test
11    void testAddition() {
12        assertEquals(6, calculator.add(1, 5));
13    }
14
15    @Test
16    void testSubtraction() {
17        assertEquals(-5, calculator.sub(2, 7));
18    }
19
20    @Test
21    void testDivision() {
22        assertEquals(0, calculator.div(5, 8));
23    }
24
25    @Test
26    void testMultiplication() {
27        assertEquals(16, calculator.mul(8, 2));
28    }
29 }
30

```

Listing 11.5: Trivial unit tests for the Calculator class.

Exercise 11.4.2 Suppose you are going to extend the functionality of the Calculator class by adding mean calculation for a list. You are utilizing Test-Driven Development (TDD) technique to write the feature. Add the new feature to the class step by step.

First, we have to write the test to see it fails.

```

1 import static org.junit.jupiter.api.Assertions.assertEquals;
2
3 import edu.atilim.se344.Calculator;
4 import org.junit.jupiter.api.Test;
5
6 class CalculatorTests {
7
8     private final Calculator calculator = new Calculator();
9
10    ...
11
12    @Test
13    void testMean() {

```

Listing 11.6: A unit test to testing the mean method of the Calculator class.

```

14     assertEquals(4.5f, calculator.mean(2, 3, 4));
15 }
16 }
17

```

Listing 11.7: The simplest implementation to pass the test.

Then, add the simplest implementation that should pass the test.

```

1 package edu.atilim.se344;
2
3 public class Calculator {
4
5     ...
6
7     public float mean(int... numbers) {
8         return 4.5f;
9     }
10 }
11

```

Listing 11.8: Another test that invalidates the previous implementation.

Write another test that will be failed by the previous implementation.

```

1 import static org.junit.jupiter.api.Assertions.assertEquals;
2
3 import edu.atilim.se344.Calculator;
4 import org.junit.jupiter.api.Test;
5
6 class CalculatorTests {
7
8     private final Calculator calculator = new Calculator();
9
10    ...
11
12    @Test
13    void testMean() {
14        assertEquals(12.0f, calculator.mean(10, 20, 8, 4, 18));
15    }
16 }
17

```

Listing 11.9: A correct implementation of the mean operation.

Finally, write the actual implementation that you think correct and run the tests again to see that all of them are passed.

```

1 package edu.atilim.se344;
2
3 public class Calculator {
4
5     ...
6
7     public float mean(int... numbers) {
8         float sum = 0.0f;
9         for (int i : numbers) {
10             sum += i;
11         }
12         return sum / numbers.length;
13     }
14 }
15

```

This is a very long way of writing code with TDD. Normally, the first steps are not taken into account and directly passed through the correct implementation part after writing the first test.

Exercise 11.4.3 Suppose that we are adding multiplication operation to our Calculator implementation. However, we are going to implement it as a repeated addition operation as in a primitive microcontroller architecture. Our implementation must only work on floating-point numbers and not integers.

```

1 package edu.atilim.se344;
2
3 public class Calculator {
4
5     ...
6
7     public double crudeMultiplication(double num1, double num2)
8     {
9         double result = 0.0;
10        for (int i = 0; i < num2; i++) {
11            result += num1;
12        }
13
14        return result;
15    }
16 }

```

Write the necessary test case to test such an implementation. Decrease and increase the precision value to see the test case is failed for smaller values.

```

1 import static org.junit.jupiter.api.Assertions.assertEquals;
2
3 import edu.atilim.se344.Calculator;
4 import org.junit.jupiter.api.Test;
5
6 class CalculatorTests {
7
8     private final Calculator calculator = new Calculator();
9
10    ...
11
12    @Test
13    @DisplayName("Operations with floating-point numbers are dangerous!")
14    void testMean() {
15        assertEquals(10000.0, calculator.crudeMultiplication
16        (0.1, 100000, 1e-8));
17    }
18 }

```

Listing 11.10: An implementation for multiplication operation with a loop.

Listing 11.11: A floating-point assert statement with precision.

McCabe's cyclomatic complexity (CC) is a software quality metric that quantifies the complexity of a software program. Complexity is inferred by measuring the number of linearly independent paths through the program. The higher the number the more complex the code. The basis of CC calculation is the *control flow graph* [13]. In software testing, CC determines the upper bound for the number of test cases that are necessary to achieve a complete branch coverage.

12.1 McCabe Cyclomatic Complexity

To calculate CC, a flow graph is utilized to depict control flow. Each node in the graph indicates several statements in the program and the flow of control is represented by directed edges. An example for such a graph is given in Figure Figure 12.1 for the code shown in Listing 12.1.

```

1 int pos_sum(int[] a) {
2     int sum = 0;
3     int i = 0;
4     while (i < a.length) {
5         if (a[i] >= 0) {
6             sum += a[i];
7         }
8         i++;
9     }
10    return sum;
11 }

```

Intuitively, CC gives us the number of independent paths in the flow graph. There are several ways for calculating CC, $V(G)$. Maybe the simplest one is adding one to the number of closed loops inside the graph. However, a more formal way of calculating CC is given in Eq. 12.1.

$$V(G) = E - N + 2 \quad (12.1)$$

Here, E is the number of edges, N is the number of nodes. Considering the graph in Figure Figure 12.1, we can calculate CC as $7 - 6 + 2 = 3$. This can also be verified by counting the number of closed loops, which is two, and adding one to it, three.

This result hints us there should be three independent paths in the graph. One can find independent paths by adding a path that is not previously covered by any of the found paths. With this information, we can identify the three independent paths as:

Independent Path	
Path 1	1-2-3-4-10
Path 2	1-2-3-4-5-8-10
Path 3	1-2-3-4-5-6-8-10

Listing 12.1: pos_sum finds the sum of all positive numbers stored in an integer array a. Input parameter is a, an array of integers. The output of the function is sum, the sum of integers inside the array a.

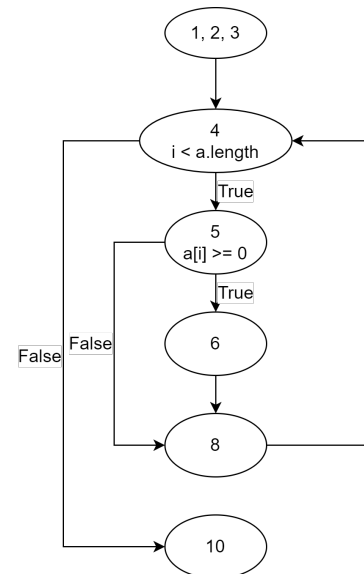


Figure 12.1: Flow graph for the Listing 12.1.

Table 12.1: Independent paths of the program that is described by Listing 12.1.

From these independent paths, we can identify three test cases that force the flow to pass through independent paths. This can be achieved by carefully crafting input variables that satisfy path conditions.

To go through path one, the array should be empty with zero length. After the check of the while condition the flow is directed to the return statement and flow has finished. For path two, the array can have many values but all of them should be negative numbers such as $a = [-5, -1, -2]$. In this path, the flow passes the inside of the if statement and, after a few circulations inside the while statement, is directed to the return statement. For path three, the array can have many integers and some of them must be positive integers such as $a = [-5, 5, 6, -1, 7]$. Here, the flow can travel inside of the if statement as well.

12.2 Statement and Branch Coverage

There are four types of coverage criterion; (1) All-Path Coverage, (2) Statement Coverage, (3) Branch Coverage, and (4) Predicate Coverage. All-Path Coverage covers all of the possible paths and finds all faults. However, some programs even have an infinite number of branches. Therefore, it is not always feasible to perform All-Path Coverage and certainly not in this manual.

Statement Coverage is the weakest one among the others. If a test goes through each statement at least once, we say that the test has 100% statement coverage. In this chapter's exercises, we will perform Statement Coverage. Writing one test case for each of the independent paths of a program generates 100% statement coverage.

To achieve full Branch Coverage, we need to select all paths that include at least one branch. The programmer needs to make sure that both true and false outcomes of the conditions are covered at least once.

12.3 Exercises

In this section, there are exercises about CC calculation and test case design. Students should try to solve the exercises without looking to the solutions as much as possible.

Exercise 12.3.1 Given the following requirement and its implementation. Test if the given implementation is correct or not by following the below procedure.

1. Draw the flow graph of the function.
2. Identify the independent paths.
3. Design the test cases.
4. Implement the test cases in a class called `TotalBillTest`.

Keith's Sheet Music needs a program to implement its music teacher's discount policy. The program prompts the user to enter the purchase total and indicates whether the purchaser is a teacher. Music teachers receive a 10% discount on their sheet music purchases unless the

purchase total is \$100 or higher. Otherwise, the discount is 12%. The discount calculation occurs before the addition of the 5% sales tax.

```
1 double calculateTotalBill(char customertype, double purchase) {
2     double total = purchase;
3     if (customertype == 't') {
4         if(purchase > 100) {
5             total = purchase * 0.91;
6         } else {
7             total = purchase * 0.88;
8         }
9     }
10    total = total * 1.05;
11    return total;
12 }
13
```

Listing 12.2: A proposed solution for the above requirement.

Answer for item a):

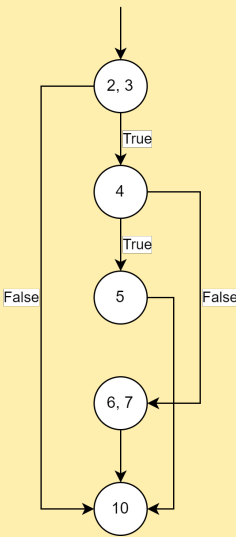


Figure 12.2: The flow graph of the calculateTotalBill function.

Answer for item b):

Independent Path	
Path 1	(2,3)-10
Path 2	(2,3)-4-(6,7)-10
Path 3	(2,3)-4-5-10

Table 12.2: Independent paths.

Answer for item c):

	Independent Path	Test Case	Expected Value	Pass/Fail
Path 1	(2,3)-10	calculateTotalBill('s', 200.0);	210.0	Pass
Path 2	(2,3)-4-(6,7)-10	calculateTotalBill('t', 50.0);	47.25	Fail
Path 3	(2,3)-4-5-10	calculateTotalBill('t', 200.0);	184.8	Fail

Table 12.3: Test cases.

Answer for the item d):

Listing 12.3: A proposed answer for the item d).

```

1 import static org.junit.jupiter.api.Assertions.assertEquals;
2 import org.junit.jupiter.api.Test;
3
4 public class TotalBillTest {
5
6     @Test
7     public void testPath1() {
8         double result = calculateTotalBill('s', 200.0);
9         assertEquals(result, 210.0);
10    }
11
12    @Test
13    public void testPath2() {
14        double result = calculateTotalBill('t', 50.0);
15        assertEquals(result, 47.25);
16    }
17
18    @Test
19    public void testPath3() {
20        double result = calculateTotalBill('t', 200.0);
21        assertEquals(result, 184.8);
22    }
23 }
24

```

Exercise 12.3.2 A hotel offers two types of rooms to their guests. Single rooms are 250TL per day, double rooms are 350TL per day. If a guest chooses a single room and if (s)he prefers to stay more than two days, hotel charges extra 225TL for each day after a single base price of 250TL. If the guest prefers to stay in a double room more than three days, (s)he is supposed to pay 200TL extra for each extra day after a single base price of 350TL.

Listing 12.4: A hotel fee calculation program for the above requirement.

```

1 public class HotelFee {
2     public int calculateFee(char type, int day) {
3         char roomType = type;
4         int duration = day;
5         int price = 3500;
6         if (roomType == 'd') {
7             if (duration < 2) {
8                 price = 350;
9             } else if (duration >= 2 && duration <= 10) {
10                 price = 250 + (duration - 2) * 225;
11             }
12         } else if (roomType == 's') {
13             if (duration < 3) {
14                 price = duration * 550;
15             } else if (duration > 3) {
16                 price = 250 + (duration - 2) * 225;
17             }
18         }
19         return price;
20     }
21 }
22

```

1. Draw the flow graph.
2. Calculate cyclomatic complexity and independent paths.
3. Write test cases.
4. Write the test methods and their results for the test cases in JUnit.
5. Use a coverage tool (EclEmma) to check the coverage percentage.

How to Install EclEmma: <http://www.eclemma.org/installation.html>

Installation from Update Site

The update site for EclEmma is <http://update.eclemma.org/>. Perform the following steps to install EclEmma from the update site:

1. From your Eclipse menu select Help → Install New Software...
2. In the Install dialog enter <http://update.eclemma.org/> at the Work with field.

For the usage instructions, follow the link: <https://www.eclemma.org/userdoc/index.html>

Answer for the item a):

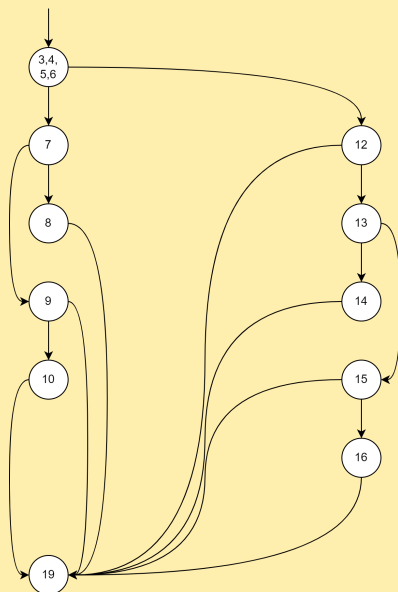


Figure 12.3: The flow graph of the `calculateFee` function.

Answer for the item b):

CC of the graph can be found using the Eq. 12.1; $V(G) = 16 - 11 + 2 = 7$. Therefore, there are seven independent paths in the graph. They are:

Table 12.4: Independent paths.

Independent Path	
Path 1	(3,4,5,6)-7-9-19
Path 2	(3,4,5,6)-7-9-10-19
Path 3	(3,4,5,6)-7-8-19
Path 4	(3,4,5,6)-12-19
Path 5	(3,4,5,6)-12-13-15-19
Path 6	(3,4,5,6)-12-13-14-19
Path 7	(3,4,5,6)-12-13-15-16-19

Answer for the item c):

Table 12.5: Test cases.

	Independent Path	Test Case	Expected Value	Pass/Fail
Path 1	(3,4,5,6)-7-9-19	calculateFee('d', 15);	2750	Fail
Path 2	(3,4,5,6)-7-9-10-19	calculateFee('d', 5);	750	Fail
Path 3	(3,4,5,6)-7-8-19	calculateFee('d', 1);	350	Pass
Path 4	(3,4,5,6)-12-19	calculateFee('k', 1);	Error	Fail
Path 5	(3,4,5,6)-12-13-15-19	calculateFee('s', 3);	475	Fail
Path 6	(3,4,5,6)-12-13-14-19	calculateFee('s', 2);	250	Fail
Path 7	(3,4,5,6)-12-13-15-16-19	calculateFee('s', 4);	700	Pass

Answer for the item d):

Listing 12.5: All the tests there defined by the independent paths of the HotelFee program.

```

1 import static org.junit.jupiter.api.Assertions.assertEquals;
2 import static org.junit.jupiter.api.Assertions.assertThrows;
3
4 import org.junit.jupiter.api.Test;
5
6 public class HotelFeeTest {
7
8     private final HotelFee hotelFee = new HotelFee();
9
10    @Test
11    public void testCalculateFeePath1() {
12        assertEquals(2750, hotelFee.calculateFee('d', 15));
13    }
14
15    @Test
16    public void testCalculateFeePath2() {
17        assertEquals(750, hotelFee.calculateFee('d', 5));
18    }
19
20    @Test
21    public void testCalculateFeePath3() {
22        assertEquals(350, hotelFee.calculateFee('d', 1));
23    }
24
25    @Test
26    public void testCalculateFeePath4() {
27        assertThrows(UnsupportedOperationException.class, () ->
28            { hotelFee.calculateFee('k', 1); });
29    }

```

```

29
30     @Test
31     public void testCalculateFeePath5() {
32         assertEquals(475, hotelFee.calculateFee('s', 3));
33     }
34
35     @Test
36     public void testCalculateFeePath6() {
37         assertEquals(250, hotelFee.calculateFee('s', 2));
38     }
39
40     @Test
41     public void testCalculateFeePath7() {
42         assertEquals(700, hotelFee.calculateFee('s', 4));
43     }
44 }
45

```

Answer for the item e): 100% source code coverage should be reported.

Exercise 12.3.3 The Department of Defense identifies soldiers according to some criteria. Only single males whose age is greater than 20 are accepted (marital status: s/S for single, m/M for married, gender: m/M for male, f/F for female). The Department of Defense needs to find a number of candidates that fit this criterion.

```

1 public class Criteria {
2     public int firtsCriteria(int nOfCandid, char[] s, char[]
3         gender, int[] age) {
4         int cnt = 1, cntFits = 0;
5         while (cnt < nOfCandid) {
6             if (s[cnt] == 's' && s[cnt] == 'S' )
7                 if(gender[cnt] == 'm' || gender[cnt] == 'M')
8                     if(age[cnt] > 02)
9                         cntFits = cntFits + 1;
10             cnt++;
11         }
12         return cntFits;
13     }
14 }

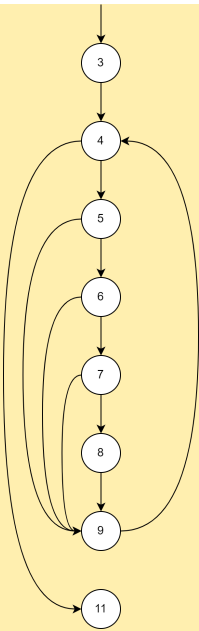
```

Listing 12.6: Program for recruitment participation requirements of D.o.D..

1. Draw the flow graph.
2. Calculate cyclomatic complexity and independent paths.
3. Write test cases.
4. Write the test methods and their results for the test cases in JUnit.
5. Use a coverage tool (Eclemma) to check the coverage percentage.

Answer for the item a):

Figure 12.4: The flow graph of the firstCriteria function.



Answer for the item b):

CC of the graph can be found using the Eq. 12.1; $V(G) = 11 - 8 + 2 = 5$. Hence, there are five independent paths in the graph. They are:

Table 12.6: Independent paths of the firstCriteria method.

Independent Path	
Path 1	3-4-11
Path 2	3-4-5-9-4-11
Path 3	3-4-5-6-9-4-11
Path 4	3-4-5-6-7-9-4-11
Path 5	3-4-5-6-7-8-9-4-11

Answer for the item c):

Table 12.7: Test cases.

Independent Path	Test Case firtsCriteria(...)	Expected Value	Pass/Fail
Path 1 3-4-11	(1, new char[] {'s'}, new char[] {'m'}, new int[] {22}); (2, new char[] {'s', 'm'}, new char[] {'m', 'm'}, new int[] {22,	1	Fail
Path 2 3-4-5-9-4-11	(2, new char[] {'s', 'm'}, new char[] {'f', 'f'}, new int[] {22,	1	Fail
Path 3 3-4-5-6-9-4-11	(2, new char[] {'s', 's'}, new char[] {'m', 'm'}, new int[] {18,	0	Pass
Path 4 3-4-5-6-7-9-4-11	(2, new char[] {'s', 's'}, new char[] {'m', 'm'}, new int[] {25,	0	Pass
Path 5 3-4-5-6-7-8-9-4-11	(2, new char[] {'s', 's'}, new char[] {'m', 'm'}, new int[] {25,	2	Fail

Answer for the item d):

Listing 12.7: All of the test cases that are defined by the independent paths of the recruitment program.

```
1 import static org.junit.jupiter.api.Assertions.assertEquals;
2
3 import org.junit.jupiter.api.Test;
4
5 public class CriteriaTest {
6
7     private final Criteria criteria = new Criteria();
8
9     @Test
```

```

10 public void testPath1() {
11     assertEquals(1, criteria.firstCriteria(1, new char[] {'s'
12     }, new char[] {'m'}, new int[] {22}));
13 }
14
15 @Test
16 public void testPath2() {
17     assertEquals(1, criteria.firstCriteria(2, new char[] {'s'
18     ', 'm'}, new char[] {'m', 'm'}, new int[] {22, 25}));
19 }
20
21 @Test
22 public void testPath3() {
23     assertEquals(0, criteria.firstCriteria(2, new char[] {'s'
24     ', 'm'}, new char[] {'f', 'f'}, new int[] {22, 25}));
25 }
26
27 @Test
28 public void testPath4() {
29     assertEquals(0, criteria.firstCriteria(2, new char[] {'s'
30     ', 's'}, new char[] {'m', 'm'}, new int[] {18, 19}));
31 }
32
33 @Test
34 public void testPath5() {
35     assertEquals(2, criteria.firstCriteria(2, new char[] {'s'
36     ', 's'}, new char[] {'m', 'm'}, new int[] {25, 35}));
37 }
38 }

```

Answer for the item e): 60% source code coverage should be reported.

Exercise 12.3.4 Bubble sort is one of the simplest sorting algorithms. It works by repeatedly swapping adjacent elements in a list to achieve a sorted list in descending or ascending order. However, it is not the first choice when it comes to business use because of its relative inefficiency, $\mathcal{O}(n^2)$. An example implementation is given in Listing 12.8.

```

1 public class BubbleSort {
2     public void bubbleSort(int arr[]) {
3         int n = arr.length;
4         for (int i = 0; i < n - 1; i++) {
5             for (int j = 0; j < n - i - 1; j++) {
6                 if (arr[j] > arr[j + 1]) {
7                     int temp = arr[j];
8                     arr[j] = arr[j + 1];
9                     arr[j + 1] = temp;
10                }
11            }
12        }
13    }
14 }

```

Listing 12.8: A bubble sort algorithm implementation in Java.

1. Draw the flow graph.
2. Calculate cyclomatic complexity and independent paths.
3. Write test cases.

4. Write the test methods and their results for the test cases in JUnit.
5. Use a coverage tool (EclEmma) to check the coverage percentage.

Answer for the item a):

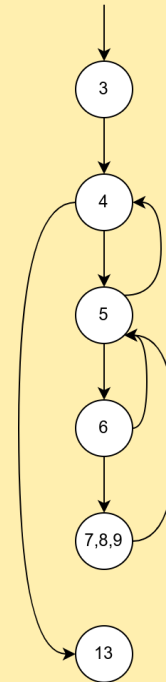


Figure 12.5: The flow graph of the bubbleSort method.

Answer for the item b):

CC of the graph can be found using the Eq. 12.1; $V(G) = 8 - 6 + 2 = 4$. Hence, there are four independent paths in the graph. They are:

Table 12.8: Independent paths of the bubbleSort method.

Independent Path	
Path 1	3-4-13
Path 2	3-4-5-4-13
Path 3	3-4-5-6-5-4-13
Path 4	3-4-5-6-7,8,9-5-4-13

Answer for the item c):

Table 12.9: Test cases for bubbleSort method.

	Independent Path	Test Case bubbleSort(...)	Expected Value	Pass/Fail
Path 1	3-4-13	(new int[] {5});	[5]	Pass
Path 2	3-4-5-4-13	Impossible in first iter.	-	-
Path 3	3-4-5-6-5-4-13	(new int[] {5, 8});	[5, 8]	Pass
Path 4	3-4-5-6-7,8,9-5-4-13	(new int[] {8, 5});	[5, 8]	Pass

Answer for the item d):

Listing 12.9: Test case implementations of the Bubble Sort class.

```

1 import static org.junit.jupiter.api.Assertions.assertEquals
  ;
2

```



```
3 import org.junit.jupiter.api.Test;
4
5 public class BubbleSortTest {
6
7     private final BubbleSort bubbleSort = new BubbleSort();
8
9     @Test
10    public void testPath1() {
11        int[] testData = new int[] {5};
12        bubbleSort.bubbleSort(testData);
13        assertEquals(new int[] {5}, testData);
14    }
15
16    @Test
17    public void testPath3() {
18        int[] testData = new int[] {5, 8};
19        bubbleSort.bubbleSort(testData);
20        assertEquals(new int[] {5, 8}, testData);
21    }
22
23    @Test
24    public void testPath4() {
25        int[] testData = new int[] {8, 5};
26        bubbleSort.bubbleSort(testData);
27        assertEquals(new int[] {5, 8}, testData);
28    }
29 }
```

Answer for the item e): 100% code coverage should be reported.

Data flow testing tries to identify flow anomalies based on the associations between values and variables. It uses the control flow graph to explore the anomalies. Examples of the anomalies include the usage of uninitialized variables and initialized variables are not used at all.

If a value is bound to a variable, then we call it as a *definition* of the variable. On the other hand, if the bound value is referred to in the program, we call it as a *use* of the variable. If a variable is used inside a predicate and has the right to decide about the execution path, then the type of use is called *predicate use (p-use)*. Similarly, if a variable's value is used to compute another variables value, we call it as *computation use (c-use)*.

13.1 Data Flow Graph

Data flow graphs (an example is shown in Figure 13.1) are generally produced by the special language translators instead of developers. The main objective of data flow graphs are to mark the definitions and uses of variables that are used inside of the program. A data flow graph is a directed graph and constructed as follows:

- ▶ A sequence of definitions and c-uses is associated with each node of the graph.
- ▶ A set of p-uses is associated with each edge of the graph.
- ▶ The entry node has a definition of each parameter and each non-local variable which occurs in the subprogram.
- ▶ The exit node has an undefinition of each local variable.

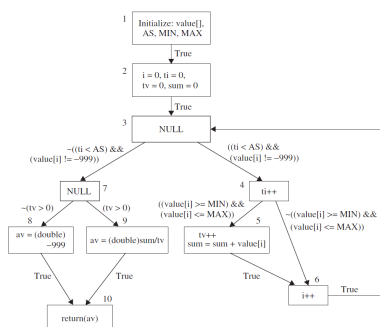


Figure 13.1: Data flow graph example.

13.2 Exercises

Given the following exercises. Fill the tables below according to the given examples using the code snippets of the exercises.

Exercise 13.2.1 ReturnAverage function computes the average of all those numbers in the input array in the positive range [MIN, MAX].

Listing 13.1: A program to calculate a ranged average in a list.

The maximum size of the array is AS. But, the array size could be smaller than AS in which case the end of input is represented by -999.

```

1 public double returnAverage (int value[], int AS, int MIN, int
   MAX) {
2     /*
3     * Function: ReturnAverage Computes the average of all those
   numbers in the
4     * input array in the positive range [MIN, MAX]. The maximum
   size of the array
5     * is AS. The array size could be smaller than AS in which
   case the end of
6     * input is represented by -999.
7     */
8
9     int i, ti, tv, sum;
10    double av;
11    i = 0;
12    ti = 0;
13    tv = 0;
14    sum = 0;
15    while (ti < AS && value[i] != -999) {
16        ti++;
17        if (value[i] >= MIN && value[i] <= MAX) {
18            tv++;
19            sum = sum + value[i];
20        }
21        i++; //i=i+1
22    }
23    if (tv > 0)
24        av = (double) sum / tv;
25    else
26        av = (double) -999;
27    return (av);
28 }
29

```

Variable	Definition line	Use line	Def-Use pair	Feasibility	Test case
value[]	1	15 (Predicate)	1-15	✓	value[] not empty AS = 8, MIN = 9, MAX = 50
value[]	1	17 (Predicate)	1-17	✓	MIN <= value[i] <= MAX
value[]	1	19 (Comp.)	1-19	✓	MIN <= value[i] <= MAX
tv	11	18	18-25	✗	tv increments here, cannot be 0

Table 13.1: Def-Use pairs and related test cases.

Some of the answers for the exercise are given in the table below.

Variable	Definition line	Use line	Def-Use pair	Feasibility	Test case
value[]	1	15 (Predicate)	1-15	✓	value[] not empty AS = 8, MIN = 9, MAX = 50
value[]	1	17 (Predicate)	1-17	✓	MIN <= value[i] <= MAX
value[]	1	19 (Comp.)	1-19	✓	MIN <= value[i] <= MAX
tv	11	18	18-25	✗	tv increments here, cannot be 0
AS	1	15 (Predicate)	1-15	✓	everything AS > 0
MIN	1	17 (Predicate)	1-17	✓	value[] not empty AS > 0
MAX	1	17 (Predicate)	1-17	✓	value[] not empty value[0]=3, MIN=1
i	11	15	11-15	✓	AS > 0
i	11	17	11-17	✓	AS>0, value[] not empty value[0]=2 MIN=3
i	11	17	11-17	✓	AS>0, value[] not empty value[0]=3 MIN=2
i	11	19	11-19	✓	AS>0, value[] not empty value[0]=3 MIN=2 MAX=5
i	11	21	11-21	✓	value[] not empty AS = 8, MIN = 9, MAX = 50

Table 13.2: Def-Use pairs and related test cases.

The main goal of static unit testing is to find the defects as close as to their point of origin. The review techniques that are applied in the static unit testing are *inspection* and *walkthrough* [2].

- ▶ **Inspection:** It is a step-by-step peer group review of a work product, with each step checked against predetermined criteria.
- ▶ **Walkthrough:** It is a review where the author leads the team through a manual or simulated execution of the product using predefined scenarios.

In this section, only inspection will be considered. A given code will be examined against a checklist by multiple groups of students.

14.1 Code Review

Code quality is a crucial concept for many organizations. Code review is a technique that aims to increase code quality. It can be performed by either an expert or a computer. Nowadays, almost all compilers have a static code analyzer that performs some of the code review tasks. Generally, a checklist is utilized to perform a code review. This checklist can be language-specific or general. A general checklist is given in the list below.

Organizations can develop their own checklists. One important point while doing so is that if a language-specific checklist is produced, then a new checklist should be produced for each language that is used.

After performing an inspection, a report is created which is signed by all participants. The report includes the found defects and problems, the degree of importance of the found problems, and the judgments of participants. The report has three acceptance criteria. They are *accept*, *conditional accept*, and *reinspect*.

1. Does the code do what has been specified in the design specification?
2. Does the procedure used in the module solve the problem correctly?
3. Does a software module duplicate another existing module which could be reused?
4. If library modules are being used, are the right libraries and the right versions of the libraries being used?
5. Does each module have a single entry point and a single exit point? Multiple exit and entry point programs are harder to test.
6. Is the cyclomatic complexity of the module more than 10? If yes, then it is extremely difficult to adequately test the module.
7. Can each atomic function be reviewed and understood in 10–15 minutes? If not, it is considered to be too complex.
8. Have naming conventions been followed for all identifiers, such as pointers, indices, variables, arrays, and constants? It is important to adhere to coding standards to ease the introduction of a new contributor (programmer) to the development of a system.

9. Has the code been adequately commented upon?
10. Have all the variables and constants been correctly initialized? Have correct types and scopes been checked?
11. Are the global or shared variables, if there are any, carefully controlled?
12. Are there data values hard coded in the program? Rather, these should be declared as variables.
13. Are the pointers being used correctly?
14. Are the dynamically acquired memory blocks deallocated after use?
15. Does the module terminate abnormally? Will the module eventually terminate?
16. Is there a possibility of an infinite loop, a loop that never executes, or a loop with a premature exit?
17. Have all the files been opened for use and closed at termination?
18. Are there computations using variables with inconsistent data types? Is overflow or underflow a possibility?
19. Are error codes and condition messages produced by accessing a common table of messages? Each error code should have a meaning, and all of the meanings should be available at one place in a table rather than scattered all over the program code.
20. Is the code portable? The source code is likely to execute on multiple processor architectures and on different operating systems over its lifetime. It must be implemented in a manner that does not preclude this kind of a variety of execution environments.
21. Is the code efficient? In general, clarity, readability, or correctness should not be sacrificed for efficiency. Code review is intended to detect implementation choices that have adverse effects on system performance.

14.2 Exercises

Given the following problem:

Write a function called `bool multiple(int, int)` that determines whether the second integer is a multiple of the first for a pair of integers. The function should take two integer arguments and return true if the second is a multiple of the first, otherwise false. Use this function in a program that inputs a series of pair of integers.

The following solution is proposed. It might be a correct implementation or not. It is not important for the context of this section.

Listing 14.1: A C++ program that confirms a number is multiple of another.

```

1 #include <iostream>
2 using namespace std;
3
4 bool multiple(int, int);
5 int func2(int);
6
7 int main() {
8     int x, num2;
9     bool a;
10    cout << "Enter two integers: ";

```



```

11     cin >> x >> num2;
12     a=multiple(x, num2);
13     if(a) {
14         cout << num2 << " is a multiple of " << x;
15     }
16     else
17         cout << num2 << " is not a multiple of " << x;
18 }
19
20 bool multiple(int X, int num2) {
21     if (num2 % X == 0)
22         return true;
23     else return false;
24 }
25
26 bool func2(int n) {
27     int i;
28     for(i = 2; i <= n / 2; i++) {
29         if(n % i == 0)
30             return false;
31     }
32     return true;
33 }

```

Perform the following exercises.

Exercise 14.2.1 Perform a code review to check problems/defect types in this C++ Program considering the list in the previous section. Write the line number and the type of the problem/defect to the following table.

#	Line number	Type of the problem/defect
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		

Table 14.1: List of found defects.

Exercise 14.2.2 Read the following description of the process of the passenger check of a Passenger Service System. Perform a review to check whether all the steps are met in terms of activities in the following activity diagram. Complete the diagram by adding the missing parts.

In a Passenger Service System, when a passenger arrives at the airport to check in, (s)he first shows his or her ticket at the check-in counter. The ticket will be checked for its validity. If the ticket is not OK, the passenger will be referred to customer service. If the ticket is OK, the passenger will check his or her luggage. If the luggage has excess weight he or she will pay an additional fee. The luggage will be forwarded to baggage transportation. The passenger receives his or her boarding pass. Note that this activity is between the passenger and the passenger service. Another solution is to make the system visible to show the interaction between the passenger service and the system.

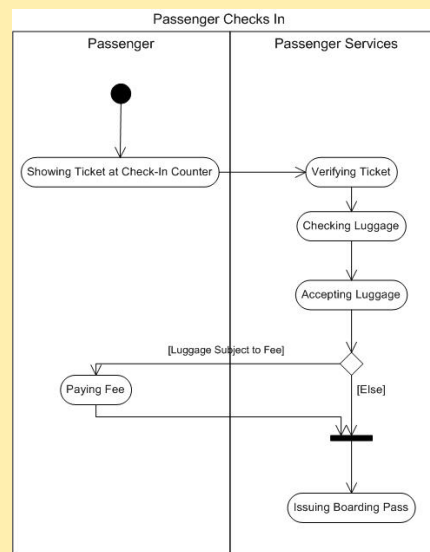


Figure 14.1: Passenger Service System passenger checks in activity diagram.

Black box testing deals with testing inputs and outputs considering a given requirement. The problem in black box testing is the size of the input space. The number of test inputs can quickly reach infeasible sizes [14]. For example, testing a calculator program's addition functionality can be made with an infinite number of test inputs, i.e. negative integers, positive integers, negative floating-point numbers, positive floating-point numbers, complex numbers and its versions, etc. Nobody has such resources or time. Because of that, there are different testing techniques that can be applied to different testing scenarios. In this chapter, some of them and their applications are presented through exercises.

15.1 Equivalence Class Partitioning

The problems with the input domain of a software-under-test can be resolved by a method called Equivalence Class Partitioning (ECP). When you look at the software-under-test as a black box, the input domain can be partitioned into several classes. The members of each class are assumed to cause the same effect on the software-under-test. When a defect is detected with input from a specific class, all of the other elements from that class are assumed to cause the same defect and vice versa. This is a limitation of ECP.

The classes from ECP are chosen in several different ways. The tester analyzes the requirements for *interesting* input conditions and partitions the input domain accordingly. Then, develops test cases from these partitions. A real-life example mirrors the test case generation techniques from white box testing techniques but instead of using the source code, the requirements are utilized for branch discovery. For example, from the textbook [14] the specifications and the relevant equivalence classes are given below:

- EC1. The input variable x is real, valid.
- EC2. The input variable x is not real, invalid.
- EC3. The value of x is greater than 0.0, valid.
- EC4. The value of x is less than 0.0, invalid.

15.2 Boundary Value Analysis

Boundary Value Analysis (BVA) is a nice addition to strengthen the ECP. Most defects generally occur at the class boundaries. These boundaries are valuable to find the defects. In ECP, any input value from a class can be used to test the black box. On the other hand, BVA requires testing of software-under-test with boundary values of equivalence classes. A tester should create test cases with valid inputs from the edges of the equivalence classes in addition to the test cases with invalid inputs. Suppose that the following specifications [14] are given to you.

The input specification for the module states that a widget identifier should consist of 3–15 alphanumeric characters of which the first two must be letters. We have three separate conditions that apply to the input: (i) it must consist of alphanumeric characters, (ii) the range for the total number of characters is between 3 and 15, and, (iii) the first two characters must be letters.

First, the tester should determine the equivalence classes:

- EC1. Part name is alphanumeric, valid.
- EC2. Part name is not alphanumeric, invalid.
- EC3. The widget identifier has between 3 and 15 characters, valid.
- EC4. The widget identifier has less than 3 characters, invalid.
- EC5. The widget identifier has greater than 15 characters, invalid.
- EC6. The first 2 characters are letters, valid.
- EC7. The first 2 characters are not letters, invalid.

After determining the ECs, the classes are split into invalid and valid as shown in Table 15.1.

Table 15.1: Equivalence class reporting table.

Condition	Valid Equivalence Classes	Invalid Equivalence Classes
1	EC1	EC2
2	EC3	EC4, EC5
3	EC6	EC7

Now, the tester can derive specific test cases from boundaries of both valid and invalid ECs. For example:

Table 15.2: Summary of test inputs using equivalence class partitioning and boundary value analysis for sample module. Table taken from [14].

Test Case ID	Input Values	Valid ECs and Bounds Covered	Invalid ECs and Bounds Covered
1	abc1	EC1, EC3 (ALB ¹), EC6	
2	ab1	EC1, EC3 (LB ²), EC6	
3	abcdef123456789	EC1, EC3 (UB ³), EC6	
4	abcde123456789	EC1, EC3 (BUB ⁴), EC6	
5	abc*	EC3 (ALB), EC6	EC2
6	ab	EC1, EC6	EC4 (BLB ⁵)
7	abcdefg123456789	EC1, EC6	EC5 (AUB ⁶)
8	a123	EC1, EC3 (ALB)	EC7
9	abcdef123	EC1, EC3, EC6	

¹ a value just above the lower boundary

² the value on the lower boundary

³ the value on the upper bound

⁴ a value just below the upper bound

⁵ a value just below the lower bound

⁶ a value just above the upper bound

After the determination of expected outputs, the logs of the test cases are recorded. The actual outputs of the tests are compared with the expected outputs to decide the fail/pass status of the tests. In addition to the boundary values, a midpoint from ECs should also be included in the

test cases as a typical case. Although BVA suggests a more specific zone to choose input values than ECP testing, these input values are merely non-unique. A tester can choose many different test input values.

15.3 Cause-and-Effect Graphing

Combining multiple conditions in EC cannot be performed intentionally. Some test cases may permit combining conditions by nature and some do not. The cause-and-effect graphing technique is developed to express causes and their effects in a graphical language. The visualization of causes and their effects greatly helps the tester to combine conditions to disclose inconsistencies that normally might not show up.

To produce a cause-and-effect graph, the tester must transform the specification to a graph that resembles a digital logic circuit. The process then starts with the decomposition of a complex software component into lower-level units. The tester identifies the causes and their effects for each of the specification units. A *cause* is a distinct input condition or an equivalence class of input conditions. An *effect* is an output condition or a system transformation. Nodes in a Boolean cause-and-effect graph are causes and effects. Causes are placed on the left side and effects on the right side of the graph. Logical relationships between causes and effects are represented by Boolean operators AND (\wedge), OR (\vee), and NOT (\sim). Let's continue with an example from [14]. Suppose we have a specification for a module that allows a user to perform a search for a character in an existing string. The specification states that:

The user must input the length of the string and the character to search for. If the string length is out-of-range an error message will appear. If the character appears in the string, its position will be reported. If the character is not in the string the message "not found" will be output.

The tester can identify the following causes and effects:

- C1: Positive integer from 1 to 80
- C2: Character to search for is in string

The effects are:

- E1: Integer out of range
- E2: Position of character in string
- E3: Character not found

Then, the following rules can be derived:

- If C1 and C2, then E2.
- If C1 and not C2, then E3.
- If not C1, then E1.

This set of rules are then converted into a cause-and-effect graph. The Figure 15.1 shows the corresponding graph.

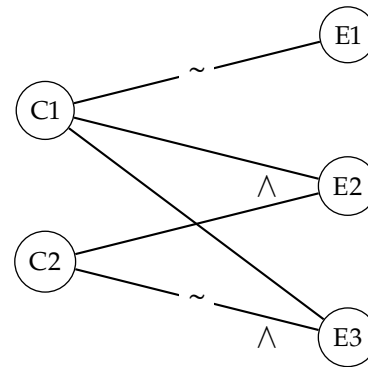


Figure 15.1: Cause-and-effect graph for the previously defined rules.

The cause-and-effect graph can be very hard to deal with if specifications are complex enough. Because of that, the tester should convert the graph to a decision table after developing the cause-and-effect graph. This way the test cases can be inferred from the decision table instead of the graph.

15.4 Decision Tables

The decision table shows the effects of all possible combinations of causes. Each column in the decision table represents a test case and lists each combination of causes. Each row represents a cause and effect. The entries of the decision table can be a "1" for a cause or effect that is present, a "0" to represent the absence of a cause or effect, and "—" to indicate a "don't care" value.

Table 15.3: Decision table for the previously defined cause-and-effect graph.

	T1	T2	T3
C1	1	1	0
C2	1	0	—
E1	0	0	1
E2	1	0	0
E3	0	1	0

The problem with decision tables is that there might be many causes and effects to consider for a complex specification. In those cases, the tester can decompose the specification into lower-level units. Then, (s)he develops cause-and-effect graphs and decision tables for these.

15.5 Error Guessing

Error guessing is based on the tester's past experience. The tester's experience with code similar to the code-under-test greatly helps her/him to find the defects. Some examples of defects that can be found by error guessing might be division by zero or conditions around array boundaries.

15.6 Exercises

Exercise 15.6.1 Bank account can be 500 to 1000 for special customers, 0 to 499 for ordinary customers, 2000 for companies (the field type is integer).

- 1. What are the equivalence classes?
- 2. Fill the Table 15.4 by finding appropriate test cases for the equivalence classes you found in previous question (a). *Add lines if necessary.*
- 3. Fill the Table 15.5 by finding appropriate test cases for the boundary testing method. *Add lines if necessary.*

Test Case #	Value	Equivalence Classes	Result (Val./Inval.)
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			

Table 15.4: Test cases for equivalence classes.

Table 15.5: Test cases for BVA strategy.

Test Case #	Value	Result (Valid/Invalid)
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		

Answer for the item a):

► Valid Classes

- (Special) $\rightarrow [500, 1000]$
- (Ordinary) $\rightarrow [0, 499]$
- (Company) $\rightarrow 2000$

► Invalid Classes

- (Special) $\rightarrow (-\infty, 499] \cup [1001, \infty)$
- (Ordinary) $\rightarrow (-\infty, -1] \cup [500, \infty)$
- (Company) $\rightarrow (-\infty, 1999] \cup [2001, \infty)$

Answer for the item b):

Table 15.6: Suggested test cases for equivalence classes.

Test Case #	Value	Equivalence Classes	Result (Valid/Invalid)
1	600	(Special) $\rightarrow [500, 1000]$	Valid
2	300	(Ordinary) $\rightarrow [0, 499]$	Valid
3	2000	(Company) $\rightarrow 2000$	Valid
4	2500	(Special) $\rightarrow (-\infty, 499] \cup [1001, \infty)$	Invalid
5	-10	(Ordinary) $\rightarrow (-\infty, -1] \cup [500, \infty)$	Invalid
6	1000	(Company) $\rightarrow (-\infty, 1999] \cup [2001, \infty)$	Invalid

Test Case #	Value	Result (Valid/Invalid)
1	499	Invalid
2	500	Valid
3	501	Valid
4	999	Valid
5	1000	Valid
6	1001	Invalid
7	-1	Invalid
8	0	Valid
9	1	Valid
10	499	Valid
11	500	Invalid
12	501	Invalid

Table 15.7: Suggested test cases for BVA strategy.

Exercise 15.6.2 The following is the interface of a function called `ConvertIntToString` in the Java language.

The requirements (pre-condition and post-condition) of the function are as follows:

- ▶ **Pre-condition:** input is a valid int
- ▶ **Post-condition:** return a string corresponding to the input integer value, e.g., return string value of "-9231" for integer value of -9231. Return NULL if input is an invalid integer

Choose an appropriate black-box technique (equivalence class partitioning, boundary value analysis) to derive test cases for this function. Note that each test case should have a concrete input value for input int and the expected output String. You should use the following format for the list of your test cases.

1. What are the equivalence classes? Fill the Table 15.8 by finding appropriate test cases for the equivalence classes.
 - ▶ Valid Classes:
 - ▶ Invalid Classes:
2. Fill the Table 15.9 by finding appropriate test cases for the boundary testing method. *Add lines if necessary.*

Test Case #	Value	Equivalence Classes	Result (Valid/Invalid)
1			
2			
3			
4			

Table 15.8: Test cases for equivalence classes.

Table 15.9: Test cases for BVA strategy.

Test Case #	Value	Result (Valid/Invalid)
1		
2		
3		
4		

Listing 15.1: The implementation of the program that should not supposed to be known.

```

1 public class IntToString {
2     public static String ConvertIntToString(int number) {
3         int StringConvert = 48;
4         int eachDigit = number;
5         int afterDivide = number;
6         String reVal = "";
7
8         while (afterDivide > 0) {
9             eachDigit = afterDivide % 10;
10            afterDivide = afterDivide / 10;
11            if(eachDigit == 0) {
12                reVal += "0";
13            }
14            else if(eachDigit == 1) {
15                reVal += "1";
16            }
17            else if(eachDigit == 2) {
18                reVal += "2";
19            }
20            else if(eachDigit == 3) {
21                reVal += "3";
22            }
23            else if(eachDigit == 4) {
24                reVal += "4";
25            }
26            else if(eachDigit == 5) {
27                reVal += "5";
28            }
29            else if(eachDigit == 6) {
30                reVal += "6";
31            }
32            else if(eachDigit == 7) {
33                reVal += "7";
34            }
35            else if(eachDigit == 8) {
36                reVal += "8";
37            }
38            else if(eachDigit == 9) {
39                reVal += "9";
40            }
41        }
42        String reVal2 = "";
43        for (int index = reVal.length() - 1 ; index >= 0 ; index
44            --) {
45            reVal2 += reVal.charAt(index);
46        }
47        return reVal2;
48    }
49 }

```

48 }
49

Answer for the item a):

- **Valid Classes:** (-inf, +inf) instance of integer
- **Invalid Classes:** String, (-inf, +inf) floating point numbers, boolean, Objects

Test Case #	Value	Equivalence Classes	Result (Valid/Invalid)
1	4785	(-inf, +inf) instance of integer	Valid
2	"hello"	String	Invalid
3	45.5	Floating point	Invalid
4	'c'	char	Invalid

Table 15.10: Suggested test cases for equivalence classes.

Answer for the item b):

Test Case #	Value	Result (Valid/Invalid)
1	-2147483649	Invalid
2	-2147483648	Valid
3	2147483647	Valid
4	2147483648	Invalid

Table 15.11: Suggested test cases for BVA strategy.

Exercise 15.6.3 The program accepts three integers, a , b , and c as inputs. These are taken to be sides of the triangle. The integers a , b , and c must satisfy following conditions:

Condition 1: $1 \leq a \leq 200$

Condition 2: $1 \leq b \leq 200$

Condition 3: $1 \leq c \leq 200$

Condition 4: $a < b + c$

Condition 5: $b < a + c$

Condition 6: $c < a + b$

The output of the program is the type of triangle determined by the three sides: Equilateral, Isosceles, Scalene, or NotATriangle. If an input value fails any of conditions Condition 1, Condition 2 or Condition 3, the program notes this with an output message such as "Value of b is not in the range of permitted values." If values of a , b , and c satisfy Condition 4, Condition 5, and Condition 6, one of four mutually exclusive outputs is given:

1. If all three sides are equal, the program output is Equilateral.
2. If exactly one pair of sides is equal, the program output is Isosceles.
3. If no pair of sides is equal, the program output is Scalene.
4. If any of conditions Condition 4, Condition 5, and Condition 6 is not met, the program output is NotATriangle.

Test the program with Decision Table-Based testing method by doing followings [15]:

- Draw the Cause-and-effect graph.
- Create a decision table for the problem.
- Create test cases.
- Run all test cases and write which ones are passed and which ones are failed.

Answer for the item a):

To draw a cause-and-effect graph, all the causes and effects should be extracted by elaborating the specification.

C1: The given side lengths permit to build a triangle.

C2: The length of side a is equal to side b.

C3: The length of side b is equal to side c.

C4: The length of side a is equal to side c.

E1: The lengths allow to build an equilateral triangle.

E2: The lengths allow to build an isosceles triangle.

E3: The lengths allow to build a scalene triangle.

E4: It is impossible.

E5: With the given lengths, it is impossible to form a triangle.

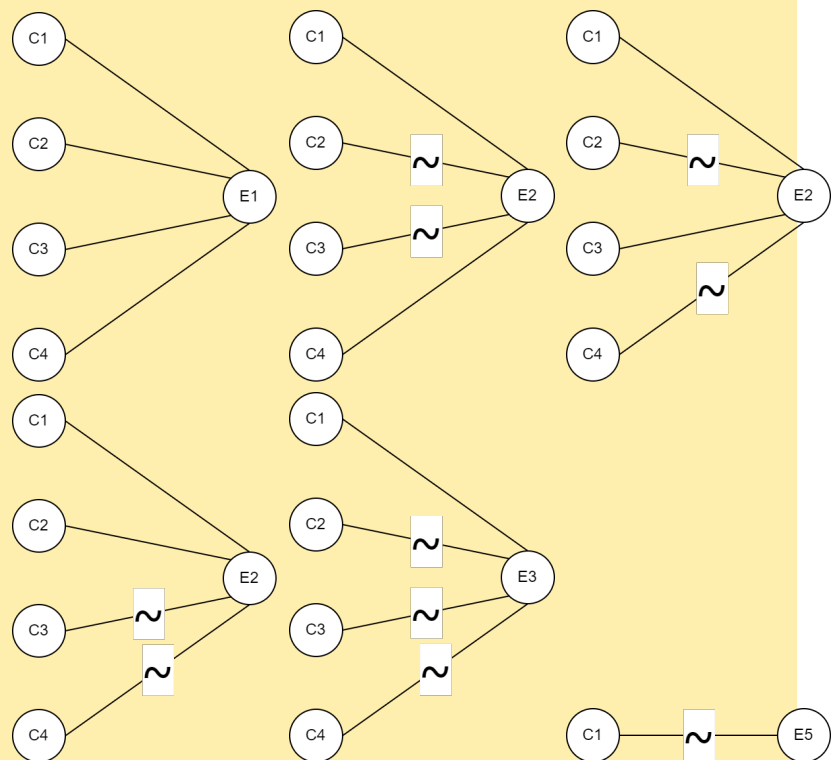


Figure 15.2: Cause-and-effect graph for the question.

Answer for the item b):

	T1	T2	T3	T4	T5	T6	T7	T8	T9
C1: Triangle	0	1	1	1	1	1	1	1	1
C2: a=b?	—	1	1	1	1	0	0	0	0
C3: b=c?	—	1	1	0	0	1	1	0	0
C4: a=c?	—	1	0	1	0	1	0	1	0
E1: Equilateral	0	1	0	0	0	0	0	0	0
E2: Isosceles	0	0	0	0	1	0	1	1	0
E3: Scalene	0	0	0	0	0	0	0	0	1
E4: Impossible	0	0	1	1	0	1	0	0	0
E5: Not a Triangle	1	0	0	0	0	0	0	0	0

Table 15.12: Decision table for the previously defined cause-and-effect graph.

Answer for the item c):

Test 1: $a = 1, b = 2, c = 7 \rightarrow E5$

Test 2: $a = 3, b = 3, c = 3 \rightarrow E1$

Test 3: $a = 2, b = 2, c = 3 \rightarrow E2$

Test 4: $a = 3, b = 2, c = 2 \rightarrow E2$

Test 5: $a = 2, b = 3, c = 2 \rightarrow E2$

Test 6: $a = 3, b = 4, c = 5 \rightarrow E3$

Answer for the item d):

The red marked test cases in Table 15.12 are failed tests.

Selenium, with its official name "The Selenium Browser Automation Project", is an umbrella project for various tools and libraries that automates certain web browser tasks according to the definition in the official website*. Selenium is a gateway to access browser features from a programming language through various language bindings. It presents a consistent API to access those features. An example that opens the official Selenium website is given in Listing 16.1 with Java programming language.

```
1 import org.openqa.selenium.WebDriver;
2 import org.openqa.selenium.chrome.ChromeDriver;
3
4 public class HelloSelenium {
5     public static void main(String[] args) {
6         WebDriver driver = new ChromeDriver();
7
8         driver.get("https://selenium.dev");
9
10        driver.quit();
11    }
12 }
```

Listing 16.1: A Java program that opens the official Selenium website through a Chrome-based browser.

At its core Selenium has a WebDriver. Once it has been installed, all Chromium and Gecko-based browsers can be controlled with a few lines of code through six different programming languages. One of these programming languages is, not surprisingly, Java. In this section, installation steps and a few examples to introduce the basics of Selenium are covered.

16.1 Installation

There are two ways to work with Selenium. With the first one, you have to install the language binding libraries for your language of choice, the browser you want to use, and the driver for that browser. This is the seemingly more professional way to work with Selenium. The other way to work is to install the Selenium IDE†. Selenium IDE provides a record-and-replay style workflow. It is easier to work with and it is not required to write code.

It is straightforward to install the Selenium library for Java with Maven. Just add the Selenium dependency which is shown in the code snippet in Listing 16.2 to the `pom.xml` file.

```
1 <dependency>
2   <groupId>org.seleniumhq.selenium</groupId>
3   <artifactId>selenium-java</artifactId>
4   <version>4.0.0</version>
5 </dependency>
```

Listing 16.2: The Selenium dependency for Maven.

* <https://www.selenium.dev/documentation/>
† <https://www.selenium.dev/selenium-ide/>

Figure 16.1: Selenium dependency on the Maven repository website.



After that, the WebDriver of choice should be installed. Notice that the major version number of the WebDriver must match with the major version number of the installed browser. All WebDriver utilities are provided by the vendor themselves and should be downloaded from their official websites[‡]. The installed WebDriver should be visible on PATH. Each operating system has a different way to maintain PATH. In Linux, if the WebDriver is installed via a package manager, no further action is needed since most package managers automatically create symbolic links to the directories which are scanned by default. Alternatively, it is possible to use the installed driver by hard-coding the driver path. Example of it is given in the code snippet in Listing 16.3.

Listing 16.3: Using the driver from a hard-coded path.

```
1 System.setProperty("webdriver.chrome.driver", "/path/to/
   chromedriver");
2 ChromeDriver driver = new ChromeDriver();
```

16.1.1 Detailed Steps for Selenium Installation

1. Create a Maven project as described in Chapter 10. Make sure to update the pom.xml file and change the Java compiler version.
2. Add the Selenium dependency also to the pom.xml file of the project (between 'dependencies' tags). To find the dependency:
 - a) Go to the website of Maven Repository.
 - b) Search for "Selenium" and select "Selenium Java" option.
 - c) Select the version 4.0.0 and copy the dependency on the opened page, which is displayed in Figure 16.1.
3. Install a WebDriver according to the browser you use (e.g. Chrome). For downloading the WebDriver:
 - a) Go to the website of Selenium, using the link <https://bit.ly/39iaKXU>.
 - b) On the opened page that is shown in Figure 16.2, according to the browser you use, click the "Downloads" section.
4. Select the proper option on the opened page, considering the key points described below:
 - a) Find the current version of the browser you use. The leftmost number of the version is its major version.

[‡] <https://www.selenium.dev/documentation/webdriver/getting-started/install-drivers/>

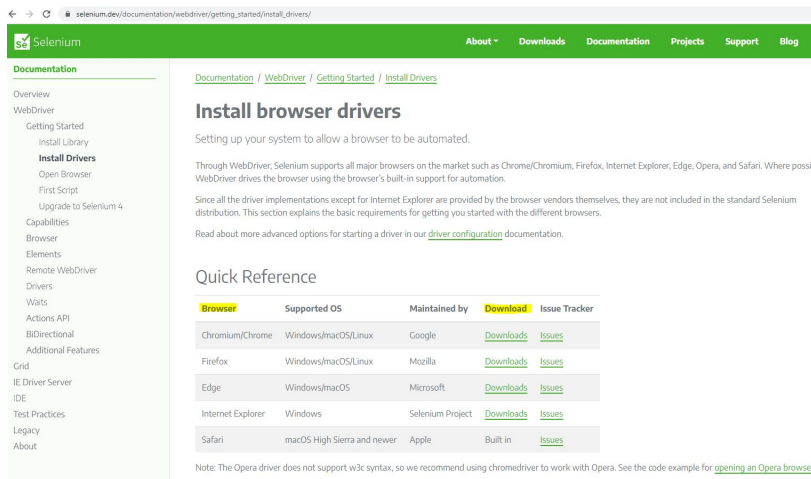


Figure 16.2: WebDriver installation page of the Selenium website.

- b) Find the WebDriver option that has the same major version as your browser, and download it.
 - c) It will be downloaded as a .zip file, so finally extract the webdriver.
5. Now, you are ready to work with Selenium and try the exercises on the lab manual.

16.2 Usage

In this section, it is assumed that Google Chrome is the choice of browser. Therefore, Chrome WebDriver should be installed. After installing the Selenium library, the browser, and the relevant WebDriver, you can open the Selenium-controlled browser with the following code snippet.

```
1 | ChromeOptions options = new ChromeOptions();
2 | driver = new ChromeDriver(options);
3 |
4 | driver.quit();
```

Listing 16.4: Start a Selenium-controlled browser instance.

By default, Selenium 4 is compatible with Chrome v75 and greater. Note that the version of the Chrome browser and the version of chromedriver must match the major version. In addition to the shared capabilities, there are specific Chrome capabilities that can be used. Let's break down and extend the code.

The following code snippet opens a Selenium-controlled browser instance with default options.

```
1 | WebDriver driver = new ChromeDriver();
```

This code snippet opens the given web page on the browser.

```
1 | driver.get("https://selenium.dev");
```

One can access all of the attributes about an opened web page through the driver variable. In the following code snippet, the title of the web page is accessed.

```
1 | driver.getTitle(); // => "Google"
```

Synchronizing the code with the current state of the browser is one of the biggest challenges with Selenium, and doing it well is an advanced topic. Essentially you want to make sure that the element is on the page before you attempt to locate it and the element is in an interactable state before you attempt to interact with it. An implicit wait is rarely the best solution, but it's the easiest to demonstrate here, so we'll use it as a placeholder.

```
1 | driver.manage().timeouts().implicitlyWait(Duration.ofMillis(500));
```

The majority of commands in most Selenium sessions are element related, and you can't interact with one without first finding an element.

```
1 | WebElement searchBox = driver.findElement(By.name("q"));
2 | WebElement searchButton = driver.findElement(By.name("btnK"));
```

There are only a handful of actions to take on an element, but you will use them frequently.

```
1 | searchBox.sendKeys("Selenium");
2 | searchButton.click();
```

Elements store a lot of information that can be requested. Notice that we need to relocate the search box because the DOM has changed since we first located it.

```
1 | driver.findElement(By.name("q")).getAttribute("value"); // => "
   | Selenium"
```

This ends the driver process, which by default closes the browser as well. No more commands can be sent to this driver instance.

```
1 | driver.quit();
```

Let's combine these previous things into a complete script.

Listing 16.5: The complete example to open Google and search for Selenium.

```
1 | import org.openqa.selenium.By;
2 | import org.openqa.selenium.WebDriver;
3 | import org.openqa.selenium.WebElement;
4 | import org.openqa.selenium.chrome.ChromeDriver;
5 |
6 | public class HelloSelenium {
7 |     public static void main(String[] args) {
8 |         driver = new ChromeDriver();
9 |
10 |        driver.get("https://google.com");
11 |
12 |        driver.getTitle(); // => "Google"
13 |
14 |        driver.manage().timeouts().implicitlyWait(Duration.
15 |            ofMillis(500));
16 |
17 |        WebElement searchBox = driver.findElement(By.name("q"));
18 |        WebElement searchButton = driver.findElement(By.name("btnK
19 |            "));
20 |
21 |        searchBox.sendKeys("Selenium");
22 |        searchButton.click();
23 |
24 |        searchBox = driver.findElement(By.name("q"));
25 |        searchBox.getAttribute("value"); // => "Selenium"
```

```

25     driver.quit();
26 }

```

WebDriver drives a browser natively, as a user would, either locally or on a remote machine using the Selenium server, marks a leap forward in terms of browser automation. More information on the API can be found in the documentation[§].

Several configuration changes need to be made to run the program from the `main(String[])` function. The first one is to add the fully qualified class name of the class that contains the main function. It is marked as `mainClass` variable in the `pom.xml` file. Add the following code snippet inside of the `<properties>` element after the compiler target element:

```
1 <exec.mainClass>the.package.name.HelloSelenium</exec.mainClass>
```

Here, the package name can be nothing or the package name that is indicated at the first line of the main class. Now, we need to tell Eclipse to build and run the program without running the tests when we send the command Maven build. If tests are run after a successful build, make sure that they all pass. Otherwise, the build will fail.

To create a build and run without tests config, right-click to the project. From the context menu, choose **Run As** > **Run Configurations...**. A dialog box should appear. In the dialog, right-click to the Maven Build and choose **New Configuration**. This should create a configuration similar to Figure 16.3. In the Goals textbox, write `clean install exec:java`. Finally, tick the **Skip Tests** checkbox. Click **Apply**, then **Run** buttons. If everything works correctly, a browser window should appear and search for "Selenium" in Google, and close.

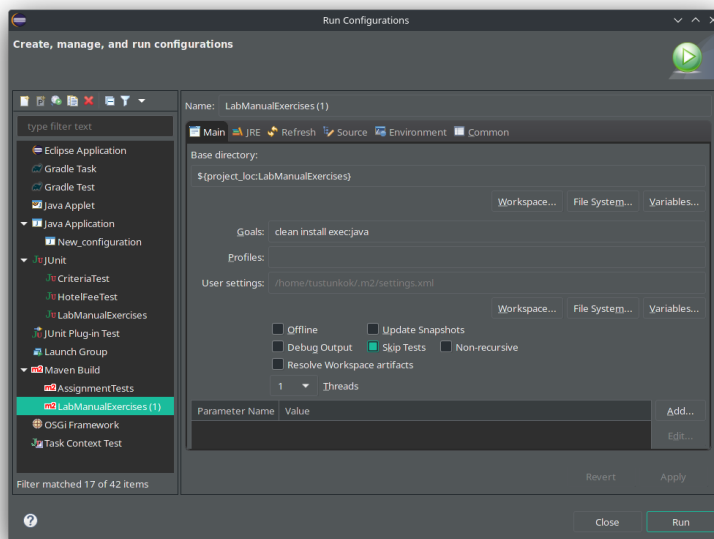


Figure 16.3: Run configurations dialog window.

[§] <https://www.selenium.dev/documentation/webdriver/>

17.1 Examples

The aim here is to write a program that tests another program that the tester has no idea of its code. Selenium's API can be thought of as instructions for a browser that performs the tests. First, the tester must make the test plan. Then, (s)he prepares the test cases. Finally, the test cases are converted into a Java program that is written with Selenium API to instruct the browser to perform the tests.

17.1.1 Additional Exercise with Selenium

In this exercise, you are expected to test the number of CMPE courses available on Moodle using Selenium. You should open the Moodle web page, click on "Moodle Courses", search for "cmpe" and get the number next to the search results text. The location of the number of courses after a search is displayed in Figure 17.1. To reach that number:

1. You should get the xpath of the web element that the number is in.
2. Using that xpath, you should get the text of that web element.
3. Since the text will be something like "Search results: xxx" and we only need the number there, you should extract the digit part of that text.
4. The data type of extracted digits will be string by default. So finally, to be able to store the number in an integer variable and compare with the expected result, you should convert it to integer.

After completing all the steps listed above, create a test class in the src/test/java folder. Write the code below inside the class and run as Maven test.

Kod gelecek.

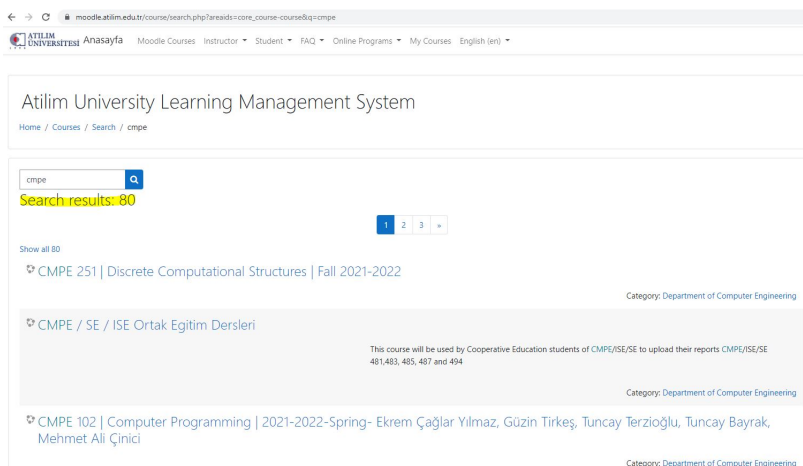


Figure 17.1: Number of search results on the Moodle web page.

17.1.2 Testing Atılım University Moodle Page

1: <https://github.com/moodle/moodle>

Like other universities, Atılım University has a Learning Management System (LMS) called Moodle. Moodle is an open-source LMS system that is supported by more than 600 contributors worldwide¹. That is all by itself guarantees that Moodle is well tested and documented. For demonstration purposes, we are going to test some of the functionality of Moodle with black-box testing strategies.

Let's start with creating a Maven project as described in Chapter 10. This time remember to include Selenium as a dependency as described in the previous section. We will test two functionality as a warm-up practice. The first one is about testing the login link to see if the login page is successfully loaded. The second one is about checking if the SE344 course page can be searched and found through the course searching mechanism. Write the following code snippet and we will discuss the important statements later.

Listing 17.1: Testing Moodle with a few warm-up tests.

```

1 import static org.junit.jupiter.api.Assertions.assertEquals;
2 import static org.junit.jupiter.api.Assertions.assertTrue;
3
4 import java.time.Duration;
5
6 import org.junit.jupiter.api.AfterAll;
7 import org.junit.jupiter.api.BeforeAll;
8 import org.junit.jupiter.api.DisplayName;
9 import org.junit.jupiter.api.Test;
10 import org.openqa.selenium.By;
11 import org.openqa.selenium.Keys;
12 import org.openqa.selenium.WebElement;
13 import org.openqa.selenium.chrome.ChromeDriver;
14 import org.openqa.selenium.support.ui.ExpectedConditions;
15 import org.openqa.selenium.support.ui.WebDriverWait;
16
17 public class TestMoodle {
18
19     private static ChromeDriver driver;
20
21     @BeforeAll
22     public static void initAll() {
23         driver = new ChromeDriver();
24     }
25
26     @Test
27     @DisplayName("Check if login page opening.")
28     public void testLogin() {
29         driver.get("https://moodle.atilim.edu.tr/");
30         WebElement loginLink = new WebDriverWait(driver, Duration.
ofSeconds(10)).until(d -> d.findElement(By.xpath("//a[text()
= 'Log in']")));
31         loginLink.click();
32         WebElement loginButton = new WebDriverWait(driver,
Duration.ofSeconds(10)).until(d -> d.findElement(By.xpath("//
button[@id = 'loginbtn']")));
33
34         assertEquals(loginButton.getText(), "Log in");
35     }

```


```

36
37 @Test
38 @DisplayName("Check if SE344 course does exist.")
39 public void testSE344Course() {
40     driver.get("https://moodle.atilim.edu.tr/");
41     WebDriverWait wait = new WebDriverWait(driver, Duration.
ofSeconds(10));
42
43     WebElement moodleCoursesLink = wait.until(d -> d.
findElement(By.xpath("//a[text() = 'Moodle Courses']")));
44     moodleCoursesLink.click();
45
46     WebElement searchMoodleCoursesInput = wait.until(d -> d.
findElement(By.xpath("//input[@name = 'q']")));
47     searchMoodleCoursesInput.sendKeys("SE 344");
48     searchMoodleCoursesInput.sendKeys(Keys.RETURN);
49
50     assertTrue(driver.getPageSource().contains("System
Software Validation and Testing"));
51
52 }
53
54 @AfterAll
55 public static void tearDownAll() {
56     if (driver != null) {
57         driver.close();
58     }
59 }
60 }

```

In the Listing 17.1, we have two tests, one initialization method, and one tear-down method. The initialization method static void `initAll()` opens and prepares a web driver which is in type `ChromeDriver()`. This means that the driver will open a Google Chrome instance. It does that only once and assumes that `WebDriver` executable can be found in `PATH`. After that, the tests are going to run.

The `testLogin()` test method starts with opening up the Moodle page (29). Then, it finds an anchor (link) which has a *Log in* text in it via the anchor's XPath (30). XPath is a query language for querying markup languages such as HTML, XML, etc. More information about XPath can be found in Appendix A. After finding the link, click on it (31) and wait for a button with an id of `loginbtn` to appear (32). When it appears, check if its text is *Log in* (34).

The `testSE344Course()` test method reopens the main page of Moodle (40). In line (41), we create a `WebDriverWait` object `wait` which we will use for explicit waits to a maximum of 10 seconds. After that, we find the *Moodle Courses* link by its XPath (43) and click it (44). In the upcoming page, we choose the input field again by its XPath (46) that allows us to search *SE 344* (47) and press  (48). After getting the desired page, we look for the text *System Software Validation and Testing* on the page (50). If we find it, then the test passes.

Finally, the static void `tearDownAll()` method run after all the tests are finished. The method checks if the driver is initialized (56) and if so, it closes it with its `close()` method (57). This method also closes the opened browser window.

APPENDIX

XML Path Language (XPath)

A

XPath is a query language to search nodes in a markup language like XML. In XPath, we write path expressions to identify certain nodes in the document [16]. Generally, XPath returns either values (from leaf nodes), elements, or attributes. The names that appear in an XPath are the node names, attribute names, or additional qualifier conditions. There are two separators in XPath. A single slash (/) appears before a tag means that the tag must be a direct child of its parent. A double slash (//) means that the tag can appear as a descendant of the previous tag at any level. Several examples are given below:

```
1 /company
2 /company/department
3 //employee[employeeSalary gt 70000]/employeeName
4 /company/employee[employeeSalary gt 70000]/employeeName
5 /company/project/projectWorker[hours ge 20.0]
```

Listing A.1: Several examples of XPath from Elmasri et al..

The expression in line (1) finds the company root node and returns all of its descendants including itself (a.k.a. all of the XML document). XPath can also contain the filename information as well. For example, suppose that the XML file is in a remote location such as `www.company.com/info.xml`. Then, we can get the company root node via XPath `doc(www.company.com/info.xml)/company`.

The expression in line (2) returns all department elements and their descendent subtrees. The order of the returned elements follows the exact order that they appear on the XML document. If we do not know the full path to a specific element, // is the convenient way to find it. The expression in line (3) is such an example. It searches and finds an element called `employee` and checks if the value of its child element `employeeSalary` is greater than 70000. If so, it returns its child called `employeeName`. The following expression in line (4) is the same one as the previous one except in line (4) the full path is specified. In the last example, at line (5), the XPath returns all `projectWorker` elements and their children which has an `hours` child element whose value is greater than or equal to 20.0.

In XPath, we can use wildcards (*). A wildcard means that all of the elements regardless of their types. For example, `/company/*` query returns all the elements under the root node `company`. If we want to address an attribute of a node, we use @ symbol. For instance, the `//title[@lang='en']` expression returns any title element that has a `lang` attribute is set to the string `en`.

Oracle VM VirtualBox Image

B

Oracle Virtual Box is used to create JUnit environment as a virtual machine to run under Windows 10 environment. In order to create a new virtual machine, you need to install Oracle VDI and Virtual Box (<https://bit.ly/3w5oAEQ>) according to your choice of the OS.

A screenshot of the JUnit VB environment in Oracle VB is given in Figure B.1 below:

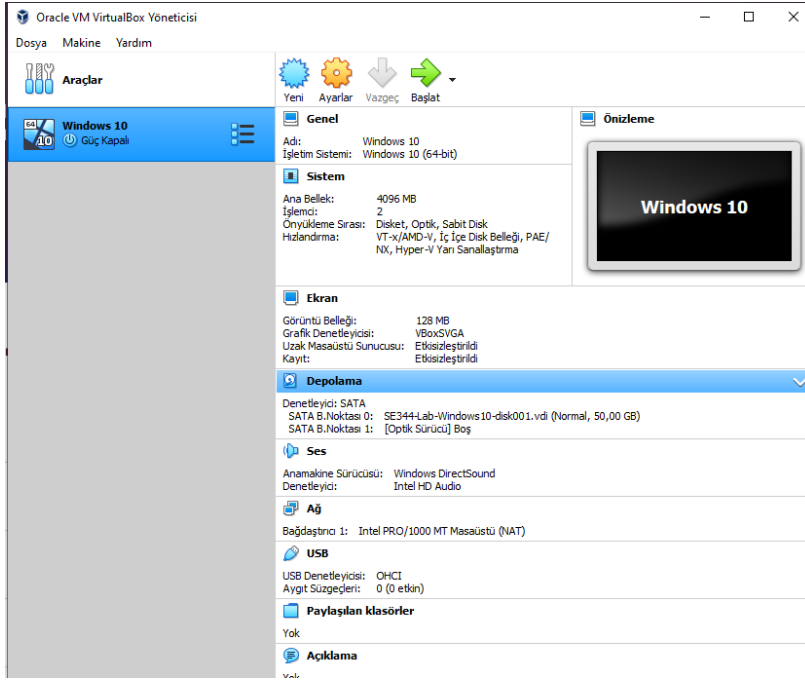


Figure B.1: Oracle Virtual Box Manager screenshot.

Bibliography

Here are the references in citation order.

- [1] Michael A Friedman and Jeffrey M Voas. *Software assessment: reliability, safety, testability*. John Wiley & Sons, Inc., 1995 (cited on page 3).
- [2] Kshirasagar Naik and Priyadarshi Tripathy. *Software testing and quality assurance: theory and practice*. John Wiley & Sons, 2011 (cited on pages 6, 7, 11, 26, 81).
- [3] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. ‘What test oracle should I use for effective GUI testing?’ In: *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings*. IEEE, 2003, pp. 164–173 (cited on page 7).
- [4] Antonia Bertolino and Lorenzo Strigini. ‘On the use of testability measures for dependability assessment’. In: *IEEE Transactions on Software Engineering* 22.2 (1996), pp. 97–108 (cited on page 7).
- [5] Michael E Fagan. ‘Design and code inspections to reduce errors in program development’. In: *IBM Systems Journal* 38.2.3 (1999), pp. 258–287 (cited on page 11).
- [6] Edward Yourdon. *Structured walkthroughs*. Yourdon Press, 1989 (cited on page 11).
- [7] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. ‘Hints on test data selection: Help for the practicing programmer’. In: *Computer* 11.4 (1978), pp. 34–41 (cited on page 14).
- [8] Thomas J McCabe. ‘A complexity measure’. In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320 (cited on page 23).
- [9] Ali Mili and Fairouz Tchier. *Software testing: Concepts and operations*. John Wiley & Sons, 2015 (cited on page 32).
- [10] Herbert Schildt. *Java: the complete reference*. Mc Graw Hill, 2007 (cited on page 49).
- [11] Herbert Schildt. *Java 7: A Beginner’s Guide*. McGraw-Hill, Inc., 2010 (cited on page 49).
- [12] Cay S. Horstmann. *Core java*. Addison-Wesley Professional, 2021 (cited on page 49).
- [13] Andreas Spillner and Tilo Linz. *Software Testing Foundations: A Study Guide for the Certified Tester Exam-Foundation Level-ISTQB® Compliant*. dpunkt. verlag, 2021 (cited on page 65).
- [14] Ilene Burnstein. *Practical software testing: a process-oriented approach*. Springer Science & Business Media, 2006 (cited on pages 85–87).
- [15] Paul C Jorgensen. *Software testing: a craftsman’s approach*. Auerbach Publications, 2013 (cited on page 94).
- [16] Ramez Elmasri and Shamkant Navathe. *Fundamentals of database systems*. 6th ed. Pearson Education Limited, 2014 (cited on page 109).

Notation

The next list describes several symbols that will be later used within the body of the document.

c Speed of light in a vacuum inertial frame

h Planck constant

Greek Letters with Pronunciations

Character	Name	Character	Name
α	alpha <i>AL-fuh</i>	ν	nu <i>NEW</i>
β	beta <i>BAY-tuh</i>	ξ, Ξ	xi <i>KSIGH</i>
γ, Γ	gamma <i>GAM-muh</i>	\omicron	omicron <i>OM-uh-CRON</i>
δ, Δ	delta <i>DEL-tuh</i>	π, Π	pi <i>PIE</i>
ϵ	epsilon <i>EP-suh-lon</i>	ρ	rho <i>ROW</i>
ζ	zeta <i>ZAY-tuh</i>	σ, Σ	sigma <i>SIG-muh</i>
η	eta <i>AY-tuh</i>	τ	tau <i>TOW (as in cow)</i>
θ, Θ	theta <i>THAY-tuh</i>	υ, Υ	upsilon <i>OOP-suh-LON</i>
ι	iota <i>eye-OH-tuh</i>	ϕ, Φ	phi <i>FEE, or FI (as in hi)</i>
κ	kappa <i>KAP-uh</i>	χ	chi <i>KI (as in hi)</i>
λ, Λ	lambda <i>LAM-duh</i>	ψ, Ψ	psi <i>SIGH, or PSIGH</i>
μ	mu <i>MEW</i>	ω, Ω	omega <i>oh-MAY-guh</i>

Capitals shown are the ones that differ from Roman capitals.

Index

A		dynamic analysis		3	S	
acceptance testing	6	dynamic unit testing	14		static analysis	3
assertEquals()	7				static checking	4
B		E			static unit testing	10
black-box testing	6	error	4		stubs	14
C		F			system level testing	6
c-use	30	failure	4		T	
c-use path	33	fault (bug)	4		test case	7
code reviews	4				test driver	14
control flow graph (CFG)	22	I			test oracle	7
D		inspection	10		test suite	7
data anomalies	30	inspections	4		U	
data flow graph	30	integration testing	4, 5		unit testing	4
dd	30	M			ur	30
debugging	18	mutation testing	14		V	
def-clear path	32	P			V-model	5
defect	4	p-use	30		validation	3
definition-clear path	32	p-use path	33		verification	3
definition-use path	32	preface	v		W	
DFG	30	R			walkthrough	4, 11
du	30	regression testing	6		white-box testing	6
du-paths	32					