Atılım University
Department of Software Engineering

System Software Validation and Testing Lab Manual

Tolga Üstünkök Ali Yazıcı

January 2022

Department of Software Engineering
Atılım University

Contents

1 Introduction **1**
- 1.1 Brief Introduction to Java Programming Language . . . 1
- 1.2 Dependency Management with Maven . . . 3
- 1.3 Installing Eclipse and Setting Up a Maven Project . . . 5
- 1.4 Introduction to JUnit Jupiter API . . . 10

2 Unit Testing w/ JUnit Jupiter API **13**
- 2.1 Writing Tests . . . 13
- 2.2 Assert Types . . . 15
- 2.3 Running Tests . . . 15
- 2.4 Exercises . . . 16

3 Static Unit Testing **21**
- 3.1 Code Review . . . 21
- 3.2 Exercises . . . 23

4 McCabe's Cyclomatic Complexity **27**
- 4.1 From Cyclomatic Complexity to Unit Tests . . . 27
- 4.2 Statement and Branch Coverage . . . 29
- 4.3 Exercises . . . 29

5 Data Flow Testing **39**
- 5.1 Data Flow Graph . . . 39
- 5.2 Exercises . . . 40

6 Black-Box Testing Strategies **45**
- 6.1 Equivalence Class Partitioning . . . 45
- 6.2 Boundary Value Analysis . . . 46
- 6.3 Cause-and-Effect Graphing . . . 48
- 6.4 Decision Tables . . . 49
- 6.5 Error Guessing . . . 50
- 6.6 Exercises . . . 50

7 Selenium for Black-Box Testing Strategies **59**
- 7.1 Installation . . . 59
- 7.2 Usage . . . 60
- 7.3 Complete Examples . . . 63

Bibliography **65**

List of Figures

1.1 Welcome screen of Eclipse. 5
1.2 Create a new project window. 6
1.3 Maven project wizard landing page. 7
1.4 Maven project wizard final page. 8
1.5 Java compiler properties for a Java project. 9

3.1 Passenger Service System passenger checks in activity diagram. 25

4.1 Flow graph for the Listing 4.1. 28
4.2 The flow graph of the `calculateTotalBill` function. 30
4.3 The flow graph of the `calculateFee` function. 33
4.4 The flow graph of the `firstCriteria` function. 36

5.1 Data flow graph example. 40

6.1 Cause-and-effect graph for the previously defined rules. 49
6.2 Cause-and-effect graph for the question. 56

7.1 Run configurations dialog window. 63

List of Tables

1.1 Maven directory layout. 4

2.1 Some of the assert methods that are defined in JUnit API. 15

3.1 Code review checklist. 22
3.2 List of found defects. 24

4.1 Independent paths. 28
4.2 Independent paths. 31
4.3 Test cases. 31
4.4 Independent paths. 33
4.5 Test cases. 34
4.6 Independent paths. 36
4.7 Test cases. 37

5.1 Def-Use pairs and related test cases. 42
5.2 Def-Use pairs and related test cases. 43

6.1 Equivalence class reporting table. 47
6.2 Summary of test inputs using equivalence class partitioning and boundary value analysis for sample module. Table taken from [5]. 47
6.3 Decision table for the previously defined cause-and-effect graph. 49
6.4 Test cases for equivalence classes. 50
6.5 Test cases for BVA strategy. 51
6.6 Suggested test cases for equivalence classes. 52
6.7 Suggested test cases for BVA strategy. 52
6.8 Test cases for equivalence classes. 53
6.9 Test cases for BVA strategy. 53
6.10 Suggested test cases for equivalence classes. 54
6.11 Suggested test cases for BVA strategy. 55
6.12 Decision table for the previously defined cause-and-effect graph. 57

1 Introduction

Software testing and test design is very important and inseparable part 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 of 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. However, in some chapters, Python is also used. 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 12-weeks semester schedule. However, some sections can be taught as two-week sections.

1.1 Brief Introduction to Java Programming Language

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.

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.

1. **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.

2. **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.

3. **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.

4. **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.

5. **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.

6. **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.

Java has many features. Some of them are object-oriented, multithreaded, allows sandbox execution, and the list goes on. Curious readers can find the full list on GeeksForGeeks website[1].

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 1.1.

Listing 1.1: 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 */`.

[1] https://www.geeksforgeeks.org/introduction-to-java/?ref=lbp

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 1.1, 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 [1, 2, 3].

1.2 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 can be 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[2]. Let's return to the subject. Each Maven project has the same directory structure shown in Table 1.1.

There can be other directories in the project. A full list can be found in Apache Maven Project website[3]. The `src` directory holds the source code of the software and other resources

[2] `https://maven.apache.org/guides/getting-started/index.html`
[3] `https://bit.ly/34VlPvC`

Table 1.1: Maven directory layout.

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

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.txt` 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 1.2.

Listing 1.2: 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 variaty of elements to use inside the project. Full list can be obtained from here[4].

1.3 Installing Eclipse and Setting Up a Maven Project

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[5] 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 in your computer. Alternatively, you can install Eclipse independently from the distribution via Snap[6] in Linux or via Chocolatey[7] in Windows 10 or above.

After installing Eclipse, a standard welcome screen should be opened as shown in Figure 1.1. In this screen go to File » New » Project....

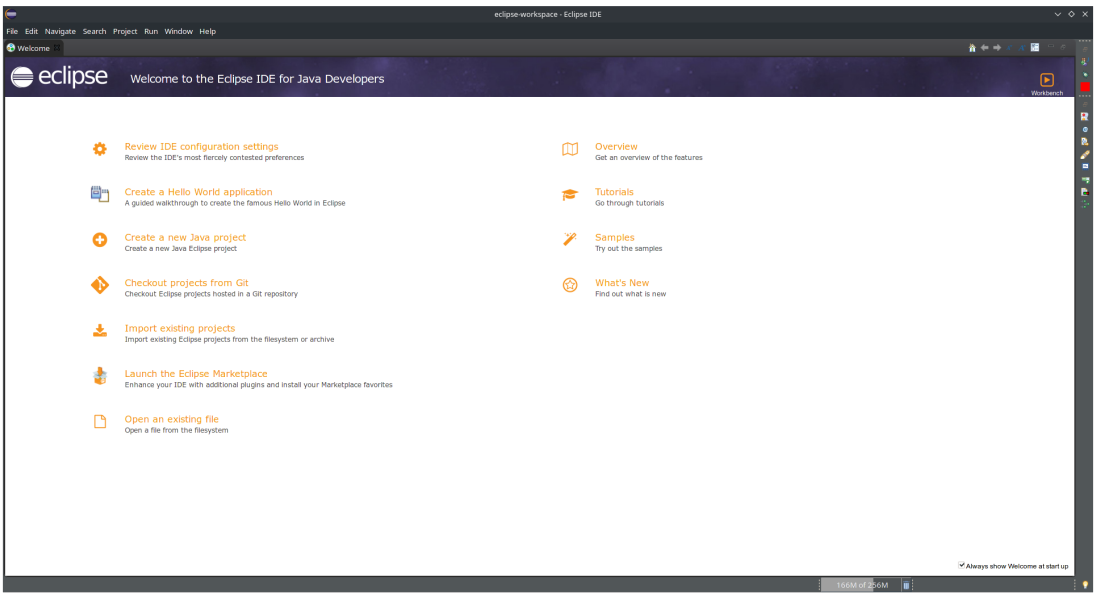

Figure 1.1: Welcome screen of Eclipse.

[4] https://maven.apache.org/pom.html
[5] https://www.eclipse.org/downloads/
[6] https://snapcraft.io/store
[7] https://community.chocolatey.org/packages

A new dialog window should be showing up as in Figure 1.2. Search and choose the *Maven Project*. After that, a wizard (shown in Figure 1.3) asks you a few questions about how you want to configure your new project.

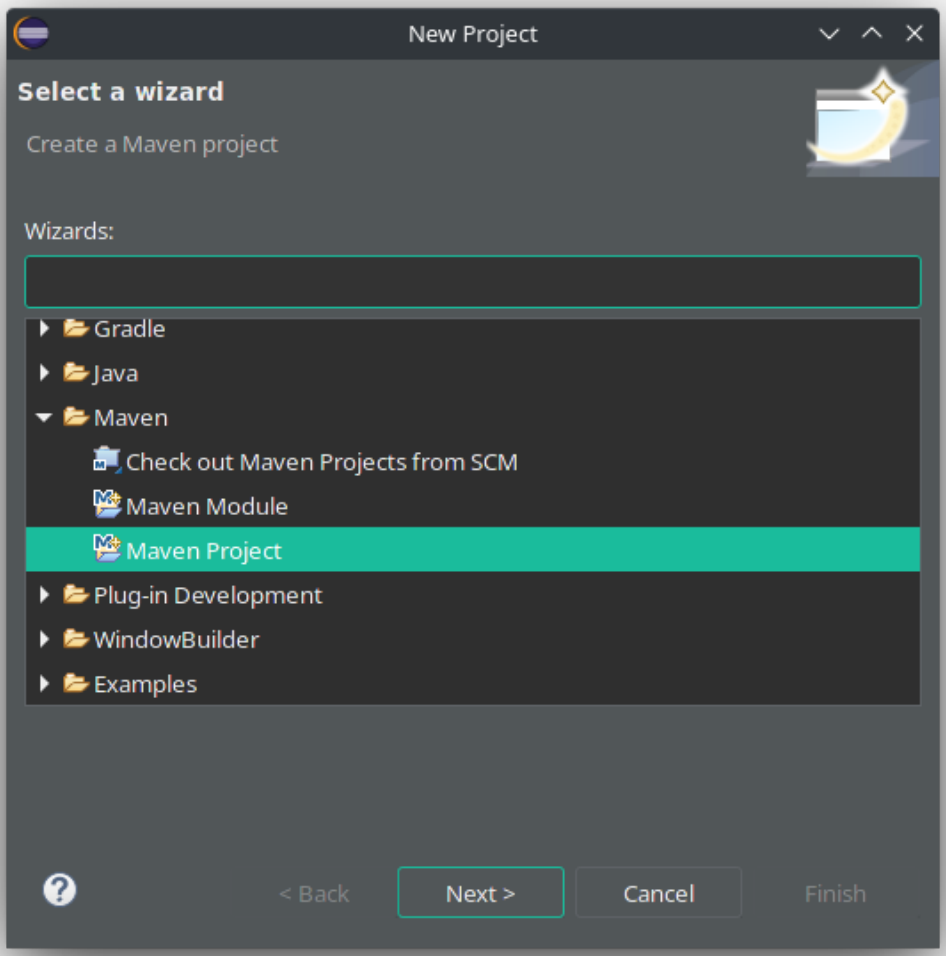

Figure 1.2: Create a new project window.

In Figure 1.3, 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 1.4.

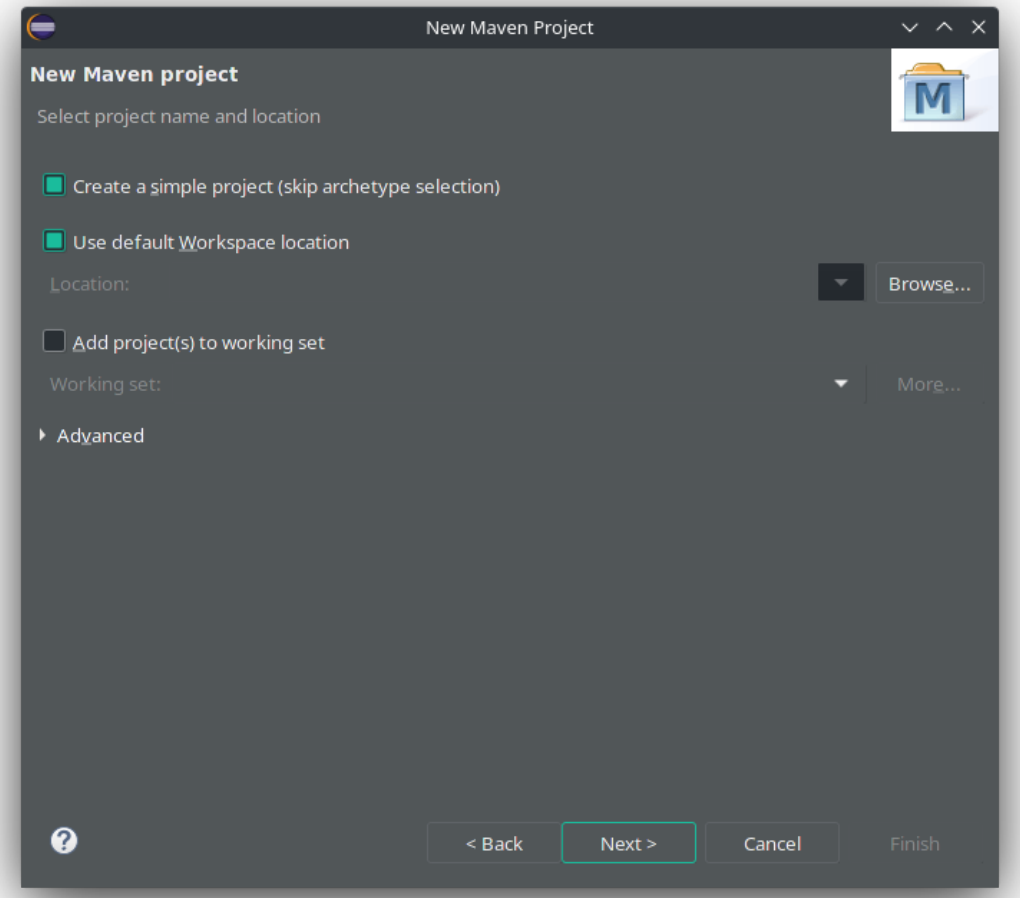

Figure 1.3: Maven project wizard landing page.

In Figure 1.4, 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 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.

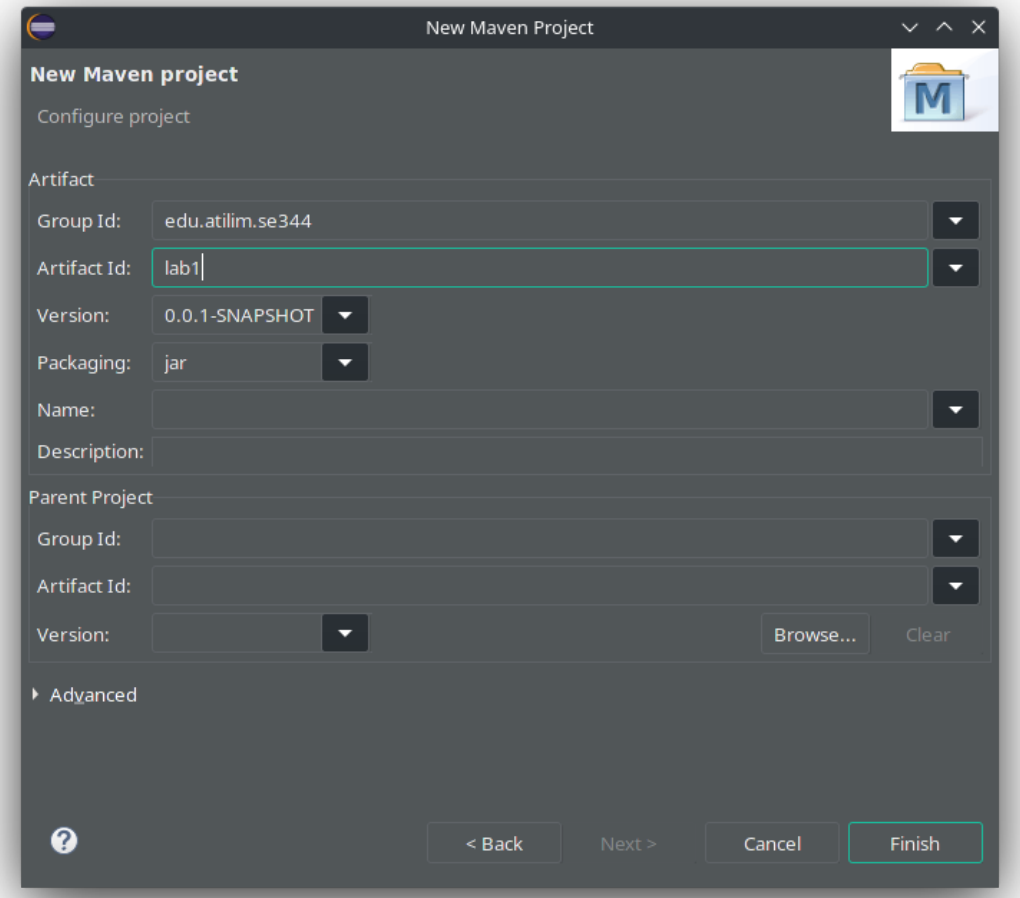

Figure 1.4: Maven project wizard final page.

Before moving on to the next topic, there is a small problem that we need to take care of. In Eclipse 2019-12, 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 1.5 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 & Close* button when you have finished with the project properties.

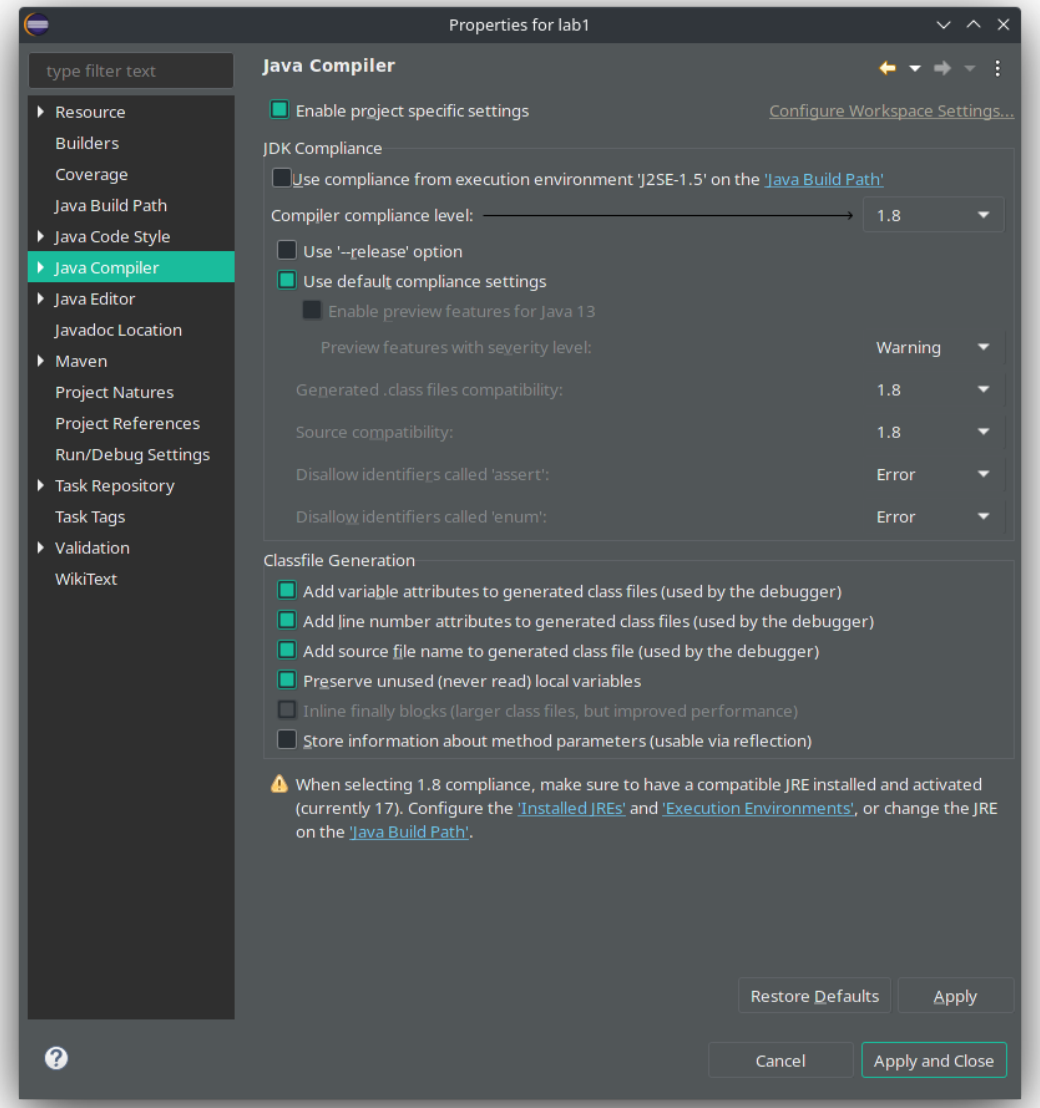

Figure 1.5: Java compiler properties for a Java project.

This concludes our project creation step. Now, we need to add necessary dependencies to our project's `pom.xml` file. In this case, there is only one dependency and it is JUnit Jupiter API. 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.

1.4 Introduction to JUnit Jupiter API

JUnit is the leading unit testing framework 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) 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 Jupiter API 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`[8]. In the middle of the page, you should see a code snippet as shown in Listing 1.3. Copy the code snippet and paste it inside the `<dependencies>` element.

Listing 1.3: JUnit Jupiter API version 5.8.2 dependency element.

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.8.2</version>
    <scope>test</scope>
</dependency>
```

After adding JUnit Jupiter API, we need to indicate a specific version of Maven Compiler Plugin that works with the newest JUnit Jupiter API. In this case, it is the version `3.8.1`. Add the code snippet[9] shown in Listing 1.4 to your `pom.xml` file under the top-level `<project>` element.

Listing 1.4: A compatible Maven Compiler Plugin JUnit Jupiter API v5.8.2..

```
<build>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
            </plugin>
        </plugins>
    </pluginManagement>
</build>
```

In the end, your `pom.xml` file should look like Listing 1.5.

[8] `https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api/5.8.2`
[9] This plugin may not be needed for the recent versions of Eclipse.

Listing 1.5: pom.xml file for future exercises.

```xml
<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">
    <modelVersion>4.0.0</modelVersion>
    <groupId>edu.atilim.se344</groupId>
    <artifactId>lab1</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <!-- Set the compiler version to Java 1.8 for compatibility with JUnit. -->
    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>5.8.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <!-- This part can be omitted in recent versions of Eclipse. -->
    <build>
        <pluginManagement>
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-compiler-plugin</artifactId>
                    <version>3.8.1</version>
                </plugin>
            </plugins>
        </pluginManagement>
    </build>
</project>
```

2 Unit Testing w/ JUnit Jupiter API

JUnit Jupiter API has a very stable and consistent API. There are two essential resources to learn it. The first is the official JUnit 5 User Guide[1] and the second one is the official JUnit 5 Java Docs[2]. In this section, the important assert types of the JUnit 5 is introduced without any meaningful context or formal methods.

2.1 Writing 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 2.1. 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`. 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.

Listing 2.1: A test case for testing the addition functionality 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 addition() {
12         assertEquals(2, calculator.add(1, 1));
13     }
14 }
```

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`

[1] `https://junit.org/junit5/docs/current/user-guide/`
[2] `https://junit.org/junit5/docs/current/api/`

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 after every test case is called. Such a standard test class is proposed by the official JUnit guide. You can see it in Listing 2.2.

Listing 2.2: A standard test class.

```
1  import static org.junit.jupiter.api.Assertions.fail;
2  import static org.junit.jupiter.api.Assumptions.assumeTrue;
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         assumeTrue("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.

2.2 Assert Types

There are many assert types that can be used in various situations. The full list can be found in the documentation[3]. Most of them are different overloads of the same assert function. The widely used ones are given in Table 2.1.

Table 2.1: Some of the assert methods that are defined in JUnit API.

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

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 2.1.

2.3 Running 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:

[3] `https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html`

Listing 2.3: A log output from a Maven test run.

```
[INFO] Scanning for projects...
[INFO]
[INFO] -----------------------< edu.atilim.se344:lab1 >------------------------
[INFO] Building lab1 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ lab1 ---
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.
    e. build is platform dependent!
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ lab1 ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @
    lab1 ---
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.
    e. build is platform dependent!
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:testCompile (default-testCompile) @ lab1
    ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ lab1 ---
[INFO] Surefire report directory: /home/tustunkok/eclipse-workspace/lab1/target/
    surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running lab1.CalculatorTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.002 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.218 s
[INFO] Finished at: 2022-02-03T15:54:36+03:00
[INFO] ------------------------------------------------------------------------
```

2.4 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 2.4: A Calculator class implementation in Java.

```java
package edu.atilim.se344;

public class Calculator {
    public int add(int n1, int n2) {
        return n1 + n2;
    }

    public int sub(int n1, int n2) {
        return n1 - n2;
    }

    public int div(int n1, int n2) {
        return n1 / n2;
    }

    public int mul(int n1, int n2) {
        return n1 * n2;
    }
}
```

Exercise 1:

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

Solution 1:

```java
import static org.junit.jupiter.api.Assertions.assertEquals;

import edu.atilim.se344.Calculator;
import org.junit.jupiter.api.Test;

class CalculatorTests {

    private final Calculator calculator = new Calculator();

    @Test
    void testAddition() {
        assertEquals(6, calculator.add(1, 5));
    }

    @Test
    void testSubtraction() {
        assertEquals(-5, calculator.sub(2, 7));
    }

    @Test
    void testDivision() {
        assertEquals(0, calculator.div(5, 8));
    }

    @Test
    void testMultiplication() {
```

```
27          assertEquals(16, calculator.mul(8, 2));
28      }
29  }
```

Exercise 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.

Solution 2:

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

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 }
```

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  }
```

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  }
```

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 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.

Listing 2.5: An implementation for multiplication operation with a loop.

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

```
12
13          return result;
14      }
15  }
```

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.

Solution 3:

```
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(0.1, 100000, 1e-8));
16      }
17  }
```

3 Static Unit Testing

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* [4].

- **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.

3.1 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 Table 3.1 as an example.

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*.

Table 3.1: Code review checklist.

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.

3.2 Exercises

Given the following problem;

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

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

```
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 4:

Perform a code review to check problems/defect types in this C Program considering the Table 3.1. Write the line number and the type of the problem/defect to the following table.

Table 3.2: List of found defects.

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

Solution 4:

No solution has been suggested.

Exercise 5:

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.

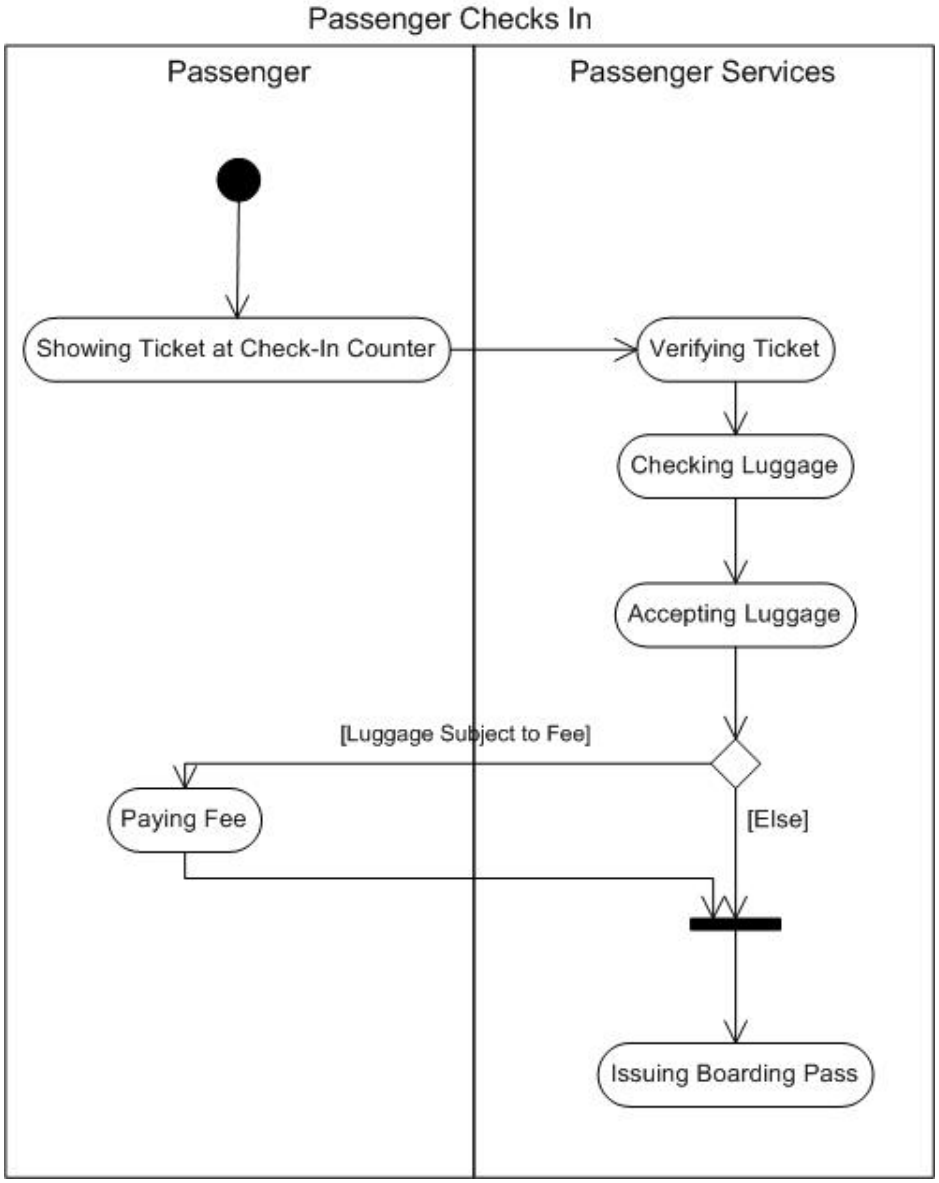

Figure 3.1: Passenger Service System passenger checks in activity diagram.

Solution 5:

No solution has been suggested.

4 McCabe's Cyclomatic Complexity

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.

4.1 From Cyclomatic Complexity to Unit Tests

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 4.1 for the code shown in Listing 4.1.

Listing 4.1: 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.

```
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 }
```

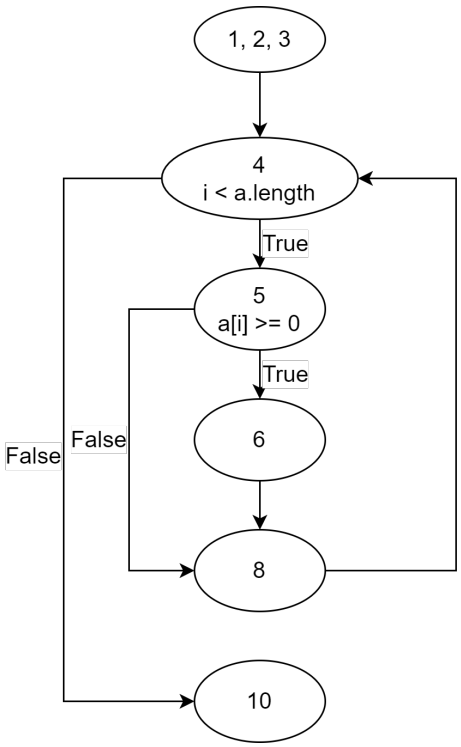

Figure 4.1: Flow graph for the Listing 4.1.

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. 4.1.

$$V(G) = E - N + 2 \tag{4.1}$$

Here, E is the number of edges, N is the number of nodes. Considering the graph in Figure 4.1, 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:

Table 4.1: Independent paths.

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

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.

4.2 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.

4.3 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 6:

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

a) Draw the flow graph of the function.
b) Identify the independent paths.
c) Design the test cases.
d) 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 }
```

Solution 6:

Answer for the item a):

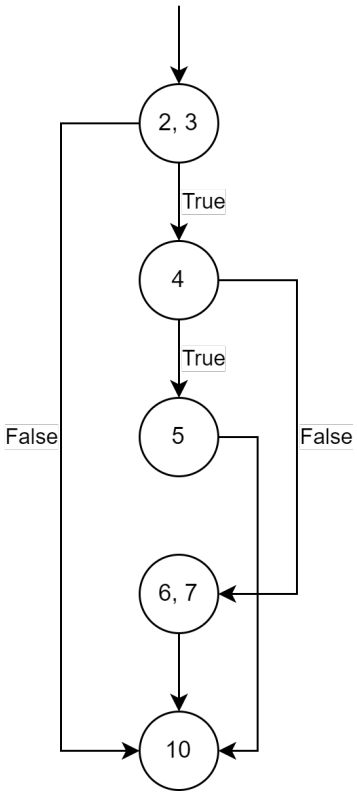

Figure 4.2: The flow graph of the `calculateTotalBill` function.

Answer for the item b):

Table 4.2: Independent paths.

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

Answer for the item c):

Table 4.3: Test cases.

	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	Pass

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 }
```

Exercise 7:

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.

```
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 }
```

a) Draw the flow graph.
b) Calculate cyclomatic complexity and independent paths.
c) Write test cases.
d) Write the test methods and their results for the test cases in JUnit.
e) 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`

Solution 7:

Answer for the item a):

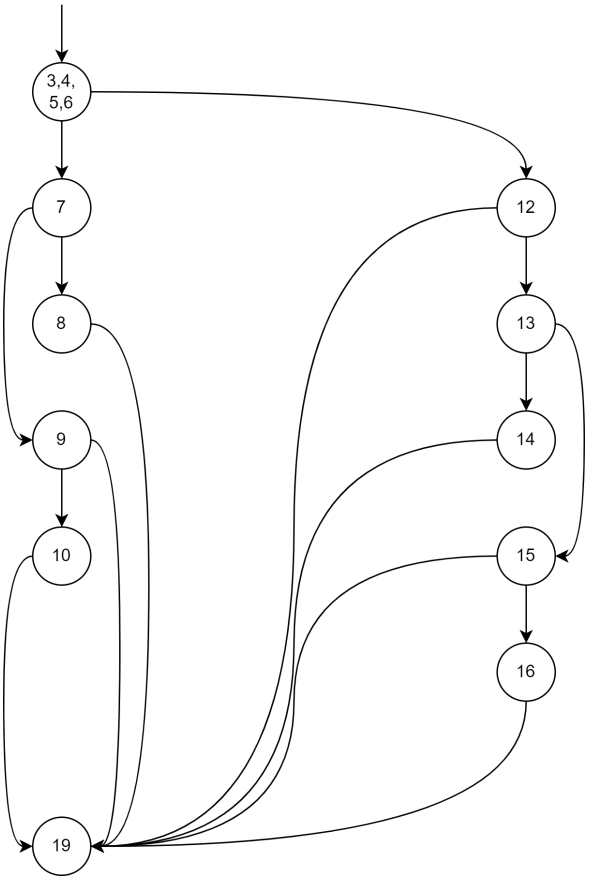

Figure 4.3: The flow graph of the `calculateFee` function.

Answer for the item b):

CC of the graph can be found using the Eq. 4.1; $V(G) = 16 - 11 + 2 = 7$. Therefore, there are seven independent paths in the graph. They are:

Table 4.4: Independent paths.

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 4.5: 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):

```
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, () -> { hotelFee.
                     calculateFee('k', 1); });
28        }
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  }
```

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

Exercise 8:

The Department of Defense identifies soldiers according to some criteria. Only single males whose age is greater than 20 are accepted (martial 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[] gender, int[] age) {
3           int cnt = 1, cntFits = 0;
4           while (cnt < nOfCandid) {
5               if (s[cnt] == 's' && s[cnt] == 'S' )
6                   if(gender[cnt] == 'm' || gender[cnt] == 'M')
7                       if(age[cnt] > 02)
8                           cntFits = cntFits + 1;
9               cnt++;
10          }
11          return cntFits;
12      }
13  }
```

a) Draw the flow graph.
b) Calculate cyclomatic complexity and independent paths.
c) Write test cases.
d) Write the test methods and their results for the test cases in JUnit.
e) Use a coverage tool (Eclemma) to check the coverage percentage.

Solution 8:

Answer for the item a):

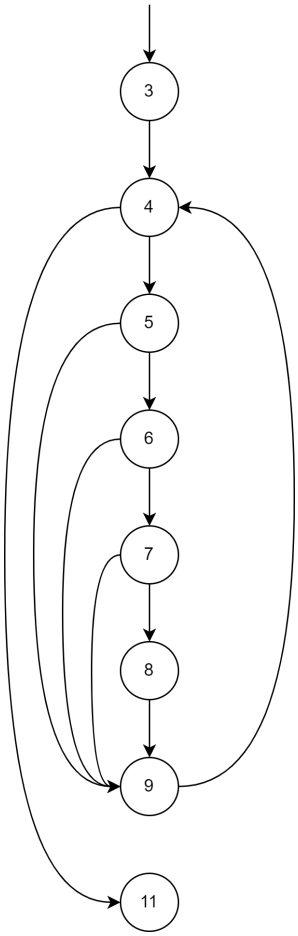

Figure 4.4: The flow graph of the `firstCriteria` function.

Answer for the item b):

CC of the graph can be found using the Eq. 4.1; $V(G) = 11 - 8 + 2 = 5$. Hence, there are five independent paths in the graph. They are:

Table 4.6: Independent paths.

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 4.7: 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});`	1	Fail
Path 2	3-4-5-9-4-11	`(2, new char[] {'s', 'm'}, new char[] {'m', 'm'}, new int[] {22, 25});`	1	Fail
Path 3	3-4-5-6-9-4-11	`(2, new char[] {'s', 'm'}, new char[] {'f', 'f'}, new int[] {22, 25});`	0	Pass
Path 4	3-4-5-6-7-9-4-11	`(2, new char[] {'s', 's'}, new char[] {'m', 'm'}, new int[] {18, 19});`	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, 35});`	2	Fail

Answer for the item d):

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

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

5 Data Flow Testing

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)*.

5.1 Data Flow Graph

Data flow graphs (an example is shown in Figure 5.1) 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.

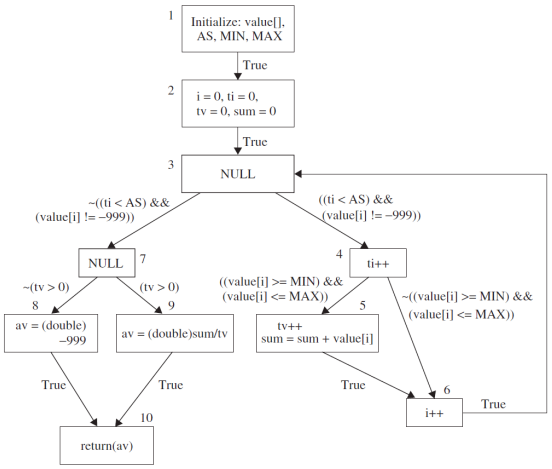

Figure 5.1: Data flow graph example.

5.2 Exercises

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

Exercise 9:

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

Table 5.1: Def-Use pairs and related test cases.

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

Solution 9:

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

Table 5.2: Def-Use pairs and related test cases.

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	✓	everyting
MIN	1	17 (Predicate)	1-17	✓	AS > 0 value[] not empty
MAX	1	17 (Predicate)	1-17	✓	AS > 0 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

6 Black-Box Testing Strategies

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 [5]. 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.

6.1 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 [5] 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.

Algorithm 1 Square root specification [5]

Require: $x, y \in \mathbb{R}$

```
1: function SQUAREROOT(x)
2:     if x ≥ 0.0 then
3:         MESSAGE(x)
4:     end if
5:     if y ≥ 0.0 and y * y ≈ x then
6:         REPLY(y)
7:     else
8:         raise Imaginary square root exception
9:     end if
10:    return y
11: end function
```

6.2 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 [5] 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 6.1.

Table 6.1: 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 6.2: Summary of test inputs using equivalence class partitioning and boundary value analysis for sample module. Table taken from [5].

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.

6.3 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, OR, and NOT. Let's continue with an example from [5]. 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 6.1 shows the corresponding graph.

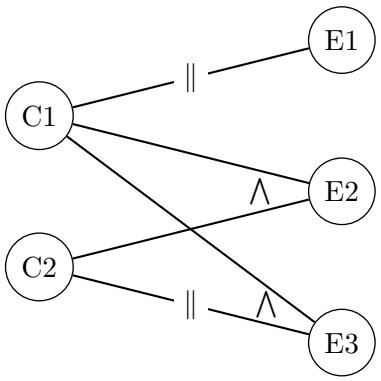

Figure 6.1: Cause-and-effect graph for the previously defined rules.

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.

6.4 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 6.3: 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.

6.5 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.

6.6 Exercises

Exercise 10:

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

a) What are the equivalence classes?
b) Fill the Table 6.4 by finding appropriate test cases for the equivalence classes you found in previous question (a). *Add lines if necessary.*
c) Fill the Table 6.5 by finding appropriate test cases for the boundary testing method. *Add lines if necessary.*

Table 6.4: Test cases for equivalence classes.

Test Case #	**Value**	**Equivalence Classes**	**Result (Valid/Invalid)**
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

Table 6.5: 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		
15		

Solution 10:

Answer for the item a):

- Valid Classes
  - (Special) → [500, 1000]
  - (Ordinary) → [0, 499]
  - (Company) → 2000
- Invalid Classes
  - (Special) → $(-\infty, 499] \cup [1001, \infty)$
  - (Ordinary) → $(-\infty, -1] \cup [500, \infty)$
  - (Company) → $(-\infty, 1999] \cup [2001, \infty)$

Answer for the item b):

Table 6.6: 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

Table 6.7: Suggested test cases for BVA strategy.

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

Exercise 11:

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.

a) What are the equivalence classes? Fill the Table 6.8 by finding appropriate test cases for the equivalence classes.
   - Valid Classes:
   - Invalid Classes:

b) Fill the Table 6.9 by finding appropriate test cases for the boundary testing method. *Add lines if necessary.*

Table 6.8: Test cases for equivalence classes.

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

Table 6.9: Test cases for BVA strategy.

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

```
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              reVal2 += reVal.charAt(index);
45          }
46          return reVal2;
47      }
48  }
```

Solution 11:

Answer for the item a):

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

Table 6.10: Suggested test cases for equivalence classes.

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

Answer for the item b):

Table 6.11: Suggested test cases for BVA strategy.

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

Exercise 12:

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 [6]:

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

Solution 12:

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 a equilateral triangle.
E2: The lengths allow to build a 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.

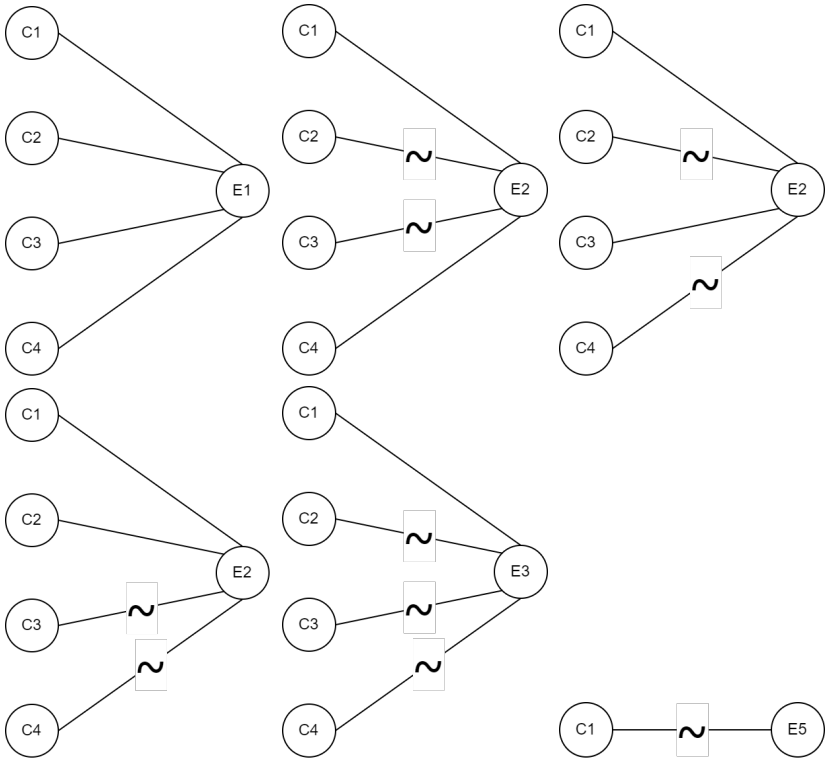

Figure 6.2: Cause-and-effect graph for the question.

Answer for the item b):

Table 6.12: Decision table for the previously defined cause-and-effect graph.

	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

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 6.12 are failed tests.

7 Selenium for Black-Box Testing Strategies

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 by their definition in the official website[1]. 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 7.1 with Java programming language.

Listing 7.1: A Java program that opens the official Selenium website through a Chrome-based browser.

```
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 }
```

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.

7.1 Installation

There are two ways to work with Selenium. With the first one, you have to install the language bindings 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[2]. Selenium IDE provides a record-and-replay style of workflow. It is easier to work with and it is not required to write code.

[1] https://www.selenium.dev/documentation/
[2] https://www.selenium.dev/selenium-ide/

It is easy to install the Selenium library for Java with Maven. Just add the Selenium dependency which is shown in the code snippet in Listing 7.2.

Listing 7.2: The Selenium dependency for Maven.

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-java</artifactId>
  <version>4.0.0</version>
</dependency>
```

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[3]. The installed WebDriver should be visible on PATH. Each operating system has a different way to maintain PATH. In Linux, 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 7.3.

Listing 7.3: Using the driver from a hard-coded path.

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

7.2 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.

Listing 7.4: Start a Selenium-controlled browser instance.

```
ChromeOptions options = new ChromeOptions();
driver = new ChromeDriver(options);

driver.quit();
```

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.

[3] https://www.selenium.dev/documentation/webdriver/getting_started/install_drivers/

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

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

```
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.

```
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.

```
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.

```
WebElement searchBox = driver.findElement(By.name("q"));
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.

```
searchBox.sendKeys("Selenium");
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.

```
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.

```
driver.quit();
```

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

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

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.ofMillis(500));
15
16        WebElement searchBox = driver.findElement(By.name("q"));
17        WebElement searchButton = driver.findElement(By.name("btnK"));
18
19        searchBox.sendKeys("Selenium");
20        searchButton.click();
21
22        searchBox = driver.findElement(By.name("q"));
23        searchBox.getAttribute("value"); // => "Selenium"
24
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[4].

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 7.1. 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.

[4] `https://www.selenium.dev/documentation/webdriver/`

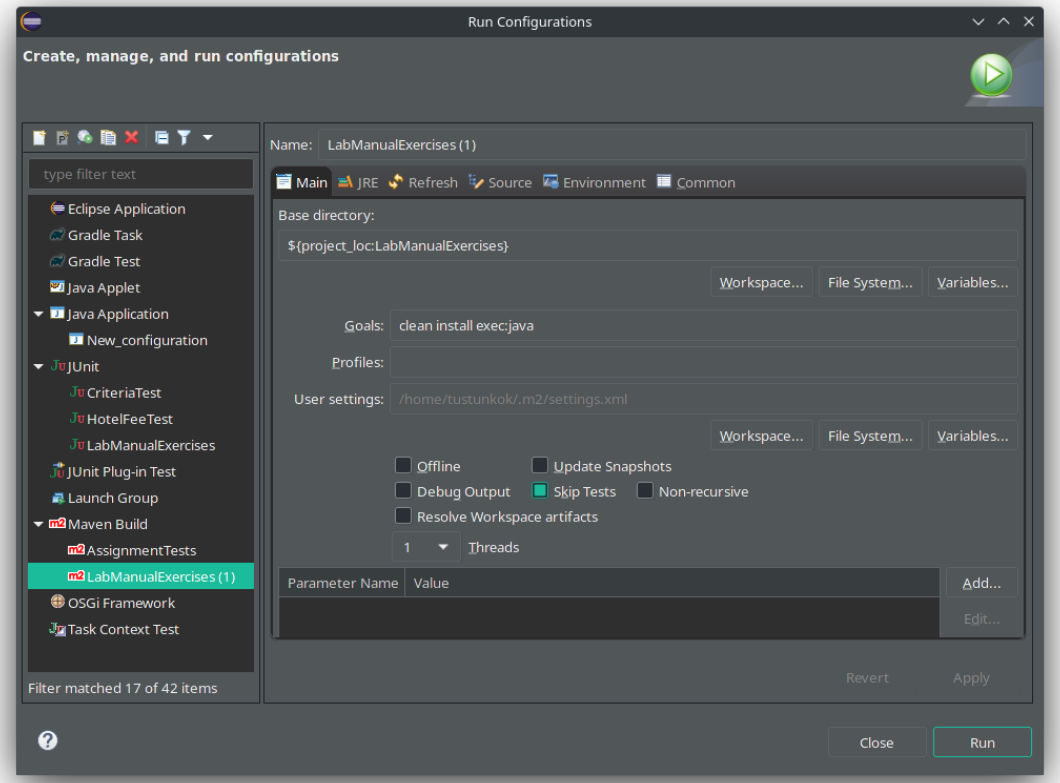

Figure 7.1: Run configurations dialog window.

7.3 Complete 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.

Bibliography

[1] Herbert Schildt. *Java: the complete reference*. Mc Graw Hill, 2007.

[2] Herbert Schildt. *Java 7: A Beginner's Guide*. McGraw-Hill, Inc., 2010.

[3] Cay S. Horstmann. *Core java*. Addison-Wesley Professional, 2021.

[4] Kshirasagar Naik and Priyadarshi Tripathy. *Software testing and quality assurance: theory and practice*. John Wiley & Sons, 2011.

[5] Ilene Burnstein. *Practical software testing: a process-oriented approach*. Springer Science & Business Media, 2006.

[6] Paul C Jorgensen. *Software testing: a craftsman's approach*. Auerbach Publications, 2013.