

CS-339 Advanced Topics in Computer Science:
Testing

Class Testing

Helen Dodd
322805

December 31, 2006

Abstract

A report on Class Testing. Testing is a key stage in the software development process. Class testing is the first testing an object-oriented program will receive. We consider a class to be an individual unit that we aim to test in isolation from others. Test cases should be designed to give the maximum chance of finding a multitude of errors. To run tests we may use a Test Driver to pass in our inputs and then compare program outputs with expected results. In programs that make use of inheritance, we cannot test units in isolation. In these cases, we must test the base classes first, and then retest inherited attributes in the context of the subclass. The question is, to what extent must inherited attributes be retested?

Contents

Introduction	2
1 Unit Testing	4
1.1 Functional or Structural?	4
1.2 When and Who	5
1.3 Drivers and Stubs	6
2 Selecting Test Cases for Class Testing	8
2.1 Boundary Value Testing	8
2.2 Equivalence Class Testing	9
2.3 Decision Table-Based Testing	9
2.4 Path Testing	10
3 Class Testing: Part One	11
3.1 Methods v Classes as Units	11
3.2 Application of Class testing	12
3.2.1 Discussion of Scenario	12
3.2.2 The Solution	13
3.2.3 StateChart Diagram	18
4 JUnits	20
4.1 A Introduction to JUnits	20
4.2 JUnit Design	21
4.3 JUnit Interface	22
5 Class Testing: Part Two	24
5.1 Flattened Class Testing	24
5.2 Incremental Class Testing	25
5.3 Inheritance Bugs	26
Conclusion	27
A Windshield Wiper Java Implementation	29
B Windshield Wiper Test Driver	34
C JUnit Source Code	46

Introduction

Testing is a very important part of the software development process. There are two main things we aim to achieve when we test a piece of software: we want to prove to the customer that the software meets its requirements. That is, that it has the functionality that the customer requested in the specification. Also, we want to find any problems or errors that are within the system. These errors might cause incorrect results in computations, program crashes, or some strange, unexpected behaviour under certain operation conditions [Som06].

Often, it is not feasible to test every single aspect of a system. Testing costs time and money, but it is important to maximise test coverage – the proportion of the program we have tested. This is especially true for critical systems where failures cannot be tolerated.

In order to maximise test coverage, a series of suitable test cases should be designed. A test case should be designed so that it has a good chance of finding an error, and it should have a purpose as to which individual part of the system it is testing. It must specify the inputs it will use, and the expected output of the program [Som06]. An oracle can be used to compare the expected outputs of test cases to the actual output produced by the program.

There are two key types of testing: *Functional Testing* and *Structural Testing*. These are often referred to as *black-box* and *white-box* testing.

“Functional testing is based on the view that any program can be considered to be a function that maps values from its input domain to values in its output range.”
[Jor06a]

In functional testing, we consider the program to be a black box. Test cases are designed based on the specification of the program, and can be designed before any code is written. We are not concerned with *how* the program works, only whether we get a correct or incorrect result. With functional testing, we can check whether the program has the full functionality as described in the specification, but not any unspecified behaviours that may be included in the program.

Structural testing looks at the system as if it were a white box – we can see the code, and testing is based on the implementation. Test cases for this paradigm are designed to ensure that all lines of code are executed at least once, and that we test conditional statements at their boundaries and within their operational limits [Pre04]. With structural testing, we won't be able to identify functionality that has not been implemented.

Ideally, when testing a piece of software, we should use both functional and structural testing to ensure that all specified features have been implemented, and that the code for these features (and indeed any extra features) is correct and error free.

There are multiple levels of testing that are performed on a program, and these stages can apply to both procedural and object-oriented programs. The levels are as follows:

- Unit/Class Testing
- Integration Testing
- System Testing

Unit testing is the first testing a program will receive, where individual components are tested in isolation. Unit testing for object-oriented systems is called *Class Testing*, and this is the main focus of this document.

Integration testing is the next stage of testing, leading on from unit testing. Just because units work correctly on their own, doesn't mean that they will work together. Integration testing has a number of approaches. The "*Big Bang*" approach is to combine all components together, and the program is tested as a whole. Obviously this is bad idea – if errors are encountered, how are you going to know which part of the program is responsible? The alternative is to use incremental approaches: *Top down* or *Bottom up*. In a top down approach, we start with the upper-most module in the unit hierarchy, and gradually add components to be tested. If we need to make use of units that we don't wish to integrate yet, they can be replaced by *stubs*, which will be discussed in section 1.3. With the bottom up method, we start by integrating the lower level units, and gradually move up the hierarchy. A *test driver* (also described in section 1.3) is required to run the tests on these lower level units [Pre04].

Finally, *system testing* is the stage where the whole program is put together and tested as a whole.

"System testing includes testing for performance, security, accountability, configuration sensitivity, start-up and recovery." [Bei90]

Instead of using test cases written by the developer or tester of the program, the test case data is provided by the customer, so that it can be demonstrated that the program works under normal operating conditions.

The rest of the document is organised as follows: In chapter 1 we will discuss the basics of unit testing for procedural languages. This will then lead in to chapter 2 where we will briefly cover some of the techniques used to design test cases. Chapter 3 then begins the discussion of the main focus of this document – Class testing. In this chapter we will look at the definition of a *unit* in the context of object-oriented unit testing, and then apply the concepts of class testing to a simple example. Chapter 4 is concerned with JUnits, a tool used to aid in the construction of test drivers for Java programs. An example showing the use of JUnits will be given. Finally, in chapter 5 we will discuss the strategies for class testing object-oriented programs that make use of inheritance. Full source code for the examples and test drivers can be found in appendices A, B and C.

Chapter 1

Unit Testing

The simplest way to describe *Class Testing* is that it is *Unit Testing* for object-oriented systems. This then begs the question: “*What is Unit testing?*”. Unit testing is the very first testing a program or piece of software will receive.

“Rather than initially testing the program as a whole, testing is first focused on the smaller building blocks of the program.” [Mye79]

The building blocks, or *units* are generally defined as being:

- Single procedures or functions
- The smallest piece of code that can be compiled by itself
- Something that is small enough to have been written by one programmer. [Jor06b]

1.1 Functional or Structural?

Unit testing can be thought of as being both structural (white-box) testing and functional (black-box) testing. For unit testing it is necessary to have the program specification available so that we can design test cases to ensure that the program meets the customers needs. From the specification we know that we need a procedure to perform a certain task. We don't really care how the code in the procedure does this task, or even if it completes the task in a fast an efficient manner. All we are really concerned about is that the procedure exists, and that it produces the expected output. This is functional testing – testing a unit against the specification.

However, we also have to test the actual implementation of the program, to ensure that the logic is correct. We should aim to execute each line of code at least once. There are a number of errors that structural unit testing looks for, and strategies for finding these. We should check that:

- Variables get set correctly
- The correct mathematical operators are used in calculations
- The right procedures or functions are called at the right time (and the correct parameters are given)

- There is no redundant code (case statements that can't be reached)
- Boundary conditions for loops are correct and loop clauses do not overlap
- Correct and meaningful exceptions are thrown, and at the right time. [Pre04]

To check for these errors we need to design test cases. Strategies for choosing inputs such as *Boundary Value testing* and *Equivalence Class testing* will be described briefly in chapter 2, along with *Decision Table-Based testing* and *Path testing* [Mye79], which ensure the program has good test coverage. Figure 1.1 shows the tests that are conducted during unit testing.

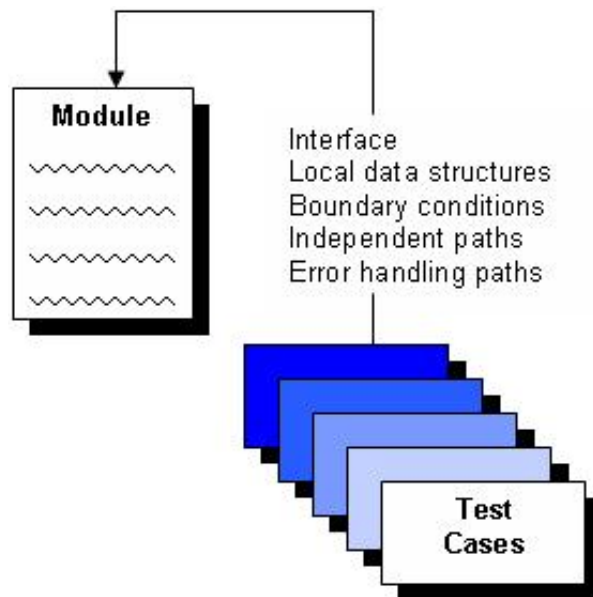


Figure 1.1: Unit Testing [Pre04]

1.2 When and Who

It is desirable to design unit testing test cases before any program code is written. This is often the case when the *Extreme Programming* paradigm is being used to develop a piece of software. Test cases are based on the specification and then the program is built to satisfy these tests [Ola03]. This ensures that the program meets the specification.

However, it is still necessary to design structural tests once the implementation is complete. The danger of writing test cases after implementation is that the programmer knows their own code. They may only write tests they know will succeed, or not test as thoroughly as an independent tester might. Of course, if the code is complex an independent tester may have difficulty choosing adequate tests as they may not understand the code as well as the programmer [Fie89].

1.3 Drivers and Stubs

In order to test a unit it is necessary to have a *driver* program. A driver is basically a “main-program” that runs the procedure or function we are testing, passing in any necessary parameters. We can then compare the output of the procedure with an expected result to find out if the test passed or failed [Pre04]. There may be some debate as to the meaning of *pass* and *fail* when testing. We could consider a failed test to be one that does not find an error, and a test with a pass result to be one that does find an error. For the purposes of this document however, a fail test result will mean that the program did not produce the output expected, and a pass result will mean that it did. A program that compares the program output with expected output may be called an *oracle* [Bei90]. We will see an example of a test driver in section 3.2.2, where it will be applied to object-oriented unit testing (class testing).

In cases where other procedures are used by the procedure under test, it is necessary to write a *stub*.

“A stub or ‘dummy subprogram’ ... may do minimal data manipulation, provides verification of entry, and returns control to the module undergoing test.” [Pre04]

Ideally, units should be tested in isolation, but as procedures often use other procedures in their operation, it is not always possible to test units in this way. We shall see this more clearly when we discuss the application of class testing, as problems caused by concepts such as inheritance will become clear.

Given the tools of drivers and stubs for use during the unit testing process, we now have an updated model of unit testing, shown in figure 1.2.

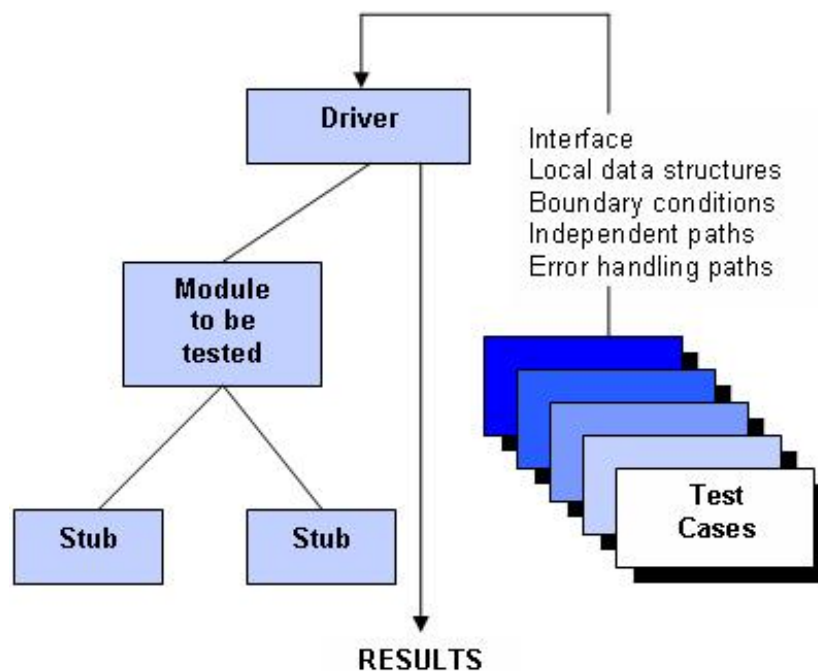


Figure 1.2: Unit Test Environment [Pre04]

Drivers and stubs mean that there is extra work to be done on top of writing the program and designing the actual test cases. However they do have benefits. Drivers automate the process of running tests, and make it easy to repeat test cases should the unit have to be altered because of failed tests. Though it may be tedious to write a test driver with hundreds of test cases, imagine having to re-run these tests manually each time an alteration was made to the program.

Chapter 2

Selecting Test Cases for Class Testing

The tools available to us for class testing an object-oriented program are the same as those we would use to test any procedural program. When testing a program we should ensure that we get a good percentage of *code coverage*, which can tell us how much of the system has been tested.

“In testing theory, coverage is defined by an adequacy criterion, which specifies the elements of an IUT[Implementation Under Test] to be exercised by a test strategy. A test suite is adequate if all the elements to be exercised have been exercised.”
[Bin99]

Having a high percentage of coverage cannot guarantee that there are no bugs in a piece of code, but it can give an indication as to how adequate a test suite is [Bin99].

Test cases should be written so that they have the maximum possibility of finding faults. To do this, there are a number of strategies we can employ. These strategies are described briefly in the following sections.

2.1 Boundary Value Testing

“Experience shows that test cases that explore boundary conditions have a higher payoff than test cases that do not.” [Mye79]

It is often the case that failures occur at the boundaries of conditional statements and data structures. When testing with boundary value analysis, test cases are designed to test at the edges of input and output domains. The following points are suggestions on how to design boundary tests.

- If an input can be a value between a and b , we should test with the values a and b , and also values just above and just below a and b .
- If a program can take a number of inputs (between max and min), we should test it with the max and min number of inputs, and also amounts just above and just below max and min .

- The previous two points should also be applied to output values.
- For data structures of specific sizes we should exercise the structure at its smallest and largest boundaries. [Pre04]

Boundary value testing is a bit of a *common sense* method of testing – programmers intuitively tend to perform this type of testing to some degree when writing programs.

2.2 Equivalence Class Testing

“An equivalence class is a set of input values such that if any value is processed correctly (incorrectly), then it is assumed that all other values will be processed correctly (incorrectly).” [Bin99]

Equivalence class testing is a black-box testing technique. We use the program specification to identify the classes of inputs. The idea is that we group inputs into sets of valid and invalid input values. We may then choose one value from each class to test the program. This minimises the number of tests we need to perform, but at the same time has a good chance of finding errors.

We can use the following guidelines to select equivalence classes:

- If an input can be one of a range of values, we define one valid ($\text{min} < x < \text{max}$) and two invalid ($x < \text{min}$ and $\text{max} > x$) equivalence classes.
- If a program takes a specified number of values as inputs (e.g. between 1 and 3 values), we should have one valid and two invalid (no inputs, and more than 3 inputs) equivalence classes.
- If an input can be a set values, each of which may be treated differently by the program, there should be one valid equivalence class for each member of the set, and one invalid equivalence class – an item that is not in the set.
- If the input must be a specific value, we should have one valid equivalence class where the value is correct, and one invalid equivalence class where the value is not correct.
- If all elements of an equivalence class are not handled in the same way by the program, the class should be split into smaller equivalence classes. [Mye79]

2.3 Decision Table-Based Testing

A decision table is a way of representing the requirements of a program and it associates conditions with actions to perform. A decision table is modelled as shown in table 2.1.

The *actions* are procedures to perform, and the *action entries* state whether the action should be performed for the set of *condition alternatives*. Conditions represent the inputs, and actions represent the outputs.

With decision table-based testing, the aim of the test cases is to ensure that the program performs the right action for each set of conditions. That is, we get the right outputs for all inputs [Bei90].

Conditions	Condition Alternatives (<i>True/False values</i>)
Actions	Action Entries

Table 2.1: Decision table

2.4 Path Testing

“Path testing is a structural testing strategy whose objective is to exercise every independent execution path through a component or program. If every independent path is executed, then all statements in the component must have been executed at least once.” [Som06]

Path testing is used to test each independent path through a program, and each of these paths are executed at least once. This method also tests conditional statements at both their true and false branches. However, path testing does not test all possible combinations of paths through a program, as the number of paths can be infinite in programs that have loops [Som06] – even if they are fairly small programs.

To path test a program, firstly a program flow graph must be constructed. A flow graph is made up of nodes showing decision and conditional statements, and edges connecting the nodes together. Each branch of a conditional statement is represented as a new path from the node [Som06].

To find the number of independent paths in a program, we use the flow graph to calculate the *cyclomatic complexity*. The cyclomatic complexity $V(G)$ for a flow graph G is given by:

$$V(G) = e - n + p,$$

where

e = number of edges in G

n = number of nodes in G

p = number of components in G [Jor06a]

The cyclomatic complexity gives us the number of independent paths through the program. We have one test case for every path, and so the cyclomatic complexity also gives us the number of test cases we need.

Chapter 3

Class Testing: Part One

This chapter will cover unit testing in the context of object-oriented programs, and apply the concepts to an example.

3.1 Methods v Classes as Units

If we are considering class testing to be unit testing for object-oriented programs, we need to redefine the concept of a unit. In procedural languages, we thought of a unit as a procedure or function. In object-oriented languages a *method* is similar to a procedure, and fits in nicely with the original definition of a unit: a method should perform a single task, and would be written by one programmer. The main issue with a method as a unit, is it being compiled by itself. Methods often rely on other methods to complete their task, and it is often necessary to run a sequence of methods *before* we can run the one we wish to test next. However it is perfectly feasible to think of a method as a unit [Jor06b].

Object-oriented programming brings with it the idea of a *class*. A class defines the characteristics of some object. The object can have attributes (variables), and behaviours (methods). Instances of a class are called *objects*. We can also think of a class as being a testable unit. This is best used in situations where there is little inheritance however [Jor06b], as how can subclasses (classes that are derived from other classes), be tested in isolation, and tested adequately?

“A key test design question is the extent to which superclass methods should be tested in a subclass context.” [Bin99]

Strategies for testing classes in a program where inheritance is present will be discussed in chapter 5.

In testing object-oriented systems, it is not necessary to choose either “*method as a unit*” or “*class as a unit*”. Certain situations may deem one methodology more suitable than the other [Jor06b], and in others a combination of these ideas might be used. Even if we think of a class as a unit, we still need to test the individual methods within it:

“Although we test one method at a time, methods cannot exist apart from a class. Class tests must exercise the cooperation of all methods in a class.” [Bin99]

3.2 Application of Class testing

In this section we will apply the concepts of class testing to a simple example.

3.2.1 Discussion of Scenario

Appendix A contains Java source code for an implementation of a Windshield Wiper, described in [Jor06a]. The basic structure of the class is given below.

```
class windshieldWiper

    private wiperSpeed
    private leverPosition
    private dialPosition

    windshieldWiper(wiperSpeed, leverPosition, dialPosition)

    getWiperSpeed()
    setWiperSpeed()

    getLeverPosition()
    setLeverPosition()

    getDialPosition()
    setDialPosition()

    senseLeverUp()
    senseLeverDown()

    senseDialUp()
    senseDialDown()

End class windshieldWiper
```

The variables `wiperSpeed`, `leverPosition`, and `dialPosition` represent the state of the windshield wiper. The lever can be in one of four positions: OFF, INT (intermittent), LOW and HIGH. The speed of the windshield wiper depends on the position of the lever. When the wiper is in the intermittent state, the speed can be varied further by switching a dial between one of three levels. The various windshield wiper speeds are given in table 3.1.

c2.Lever	OFF	INT	INT	INT	LOW	HIGH
c2.Dial	n/a	1	2	3	n/a	n/a
a1.Wiper	0	4	6	12	30	60

Table 3.1: Decision table showing windshield wiper speeds [Jor06a]

The windshield wiper class contains *accessor* (`get`) and *modifier* (`set`) methods that can be used by other classes to alter and check the state of the wiper. The `senseLeverUp`,

`senseLeverDown`, `senseDialUp` and `senseDialDown` methods specify the action when the dial or lever is used. They alter the respective position variables and the `wiperSpeed` variable.

3.2.2 The Solution

In this very simple example, we can use a bottom-up approach to testing the class [Jor06b]. We can start by testing the *get* and *set* methods, and then the *sense* methods that specify the action when the lever and dial are moved.

In order to test the class we will use a test driver, described in section 1.3. As we shall see in chapter 4, languages have tools that act as test drivers. However, we can write a test driver in the same language we write the implementation. Below is the test driver for the get and set methods of the windshield wiper implementation, written in Java. A screenshot of the test results is give in figure 3.1.

```
public class testGetSet {
    // Variables for storing results.
    private int wiperSpeed;
    private String leverPos;
    private int dialPos;
    private String testResult;

    public testGetSet() {
        // Create a new windshield wiper object, initially
        // set the wiper speed to 0, lever position to OFF
        // and dial position to 1.
        windshieldWiper testCase = new windshieldWiper(0, "OFF", 1);
        System.out.println("\nWindshield wiper initialised
            \nTesting commencing...\n");

        // Set the wiper speed and check result.
        testCase.setWiperSpeed(6);
        wiperSpeed = testCase.getWiperSpeed();
        if(wiperSpeed == 6) {
            testResult = "PASS";
        }
        else {
            testResult = "FAIL";
        }
        System.out.println("1) Set wiper speed to 6: " + testResult);

        // Set the lever position and check result.
        testCase.setLeverPosition("INT");
        leverPos = testCase.getLeverPosition();
        if(leverPos.equals("INT")) {
            testResult = "PASS";
        }
        else {
```

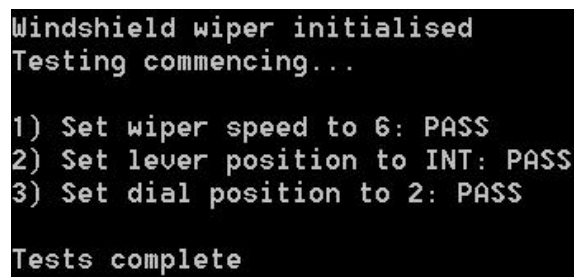
```
        testResult = "FAIL";
    }
    System.out.println("2) Set lever position to INT: " + testResult);

    // Set the dial position and check result.
    testCase.setDialPosition(2);
    dialPos = testCase.getDialPosition();
    if(dialPos == 2) {
        testResult = "PASS";
    }
    else {
        testResult = "FAIL";
    }
    System.out.println("3) Set dial position to 2: " + testResult);

    System.out.println("\nTests complete");
}

// Main method to run the tests.
public static void main(String args[]) {
    testGetSet test = new testGetSet();
}
} // End of class.
```

Ideally, we want to test methods isolation. As we can see from the above example however, and as we shall see in chapter 5, this is not always possible. For us to ensure that the modifier methods in the windshield wiper class are correct, it is necessary to check the state of the variables by using the accessor methods. Therefore, the testing for the get and set methods must be done side-by-side.



```
Windshield wiper initialised
Testing commencing...

1) Set wiper speed to 6: PASS
2) Set lever position to INT: PASS
3) Set dial position to 2: PASS

Tests complete
```

Figure 3.1: Results of the accessor and modifier tests

Appendix B provides the code for a full test suit for the windshield wiper class. Once we are confident that the accessor and modifier methods are correct, we can proceed to test the lever and dial by writing a test driver for each of the *sense* methods. These drivers should aim to test each of the *if* clauses and *case* statements. An example of the test driver for testing the *up* action of the lever is given on the next page.

```
public class testSenseLeverUp {
    // Variables for storing results.
    private int wiperSpeed;
    private String leverPos;
    private int dialPos;
    private String testResult;

    public testSenseLeverUp() {
        windshieldWiper testCase = new windshieldWiper(0, "OFF", 1);
        System.out.println("\nTest 2: Test senseLeverUp\nWindshield
            wiper initialised\nTesting commencing...\n");

        // Simulate the lever moving up.
        try {
            testCase.senseLeverUp();
        } catch (LeverErrorException e) {
            testResult = "FAIL";
        }
        leverPos = testCase.getLeverPosition();
        if(leverPos.equals("INT")) {
            testResult = "PASS";
        }
        else {
            testResult = "FAIL";
        }
        System.out.println("1) Move lever up to INT: " + testResult);

        // Simulate the lever moving up.
        try {
            testCase.senseLeverUp();
        } catch (LeverErrorException e) {
            testResult = "FAIL";
        }
        leverPos = testCase.getLeverPosition();
        if(leverPos.equals("LOW")) {
            testResult = "PASS";
        }
        else {
            testResult = "FAIL";
        }
        System.out.println("2) Move lever up to LOW: " + testResult);

        // Simulate the lever moving up.
        try {
            testCase.senseLeverUp();
        } catch (LeverErrorException e) {
            testResult = "FAIL";
        }
    }
}
```

```

    }
    leverPos = testCase.getLeverPosition();
    if(leverPos.equals("HIGH")) {
        testResult = "PASS";
    }
    else {
        testResult = "FAIL";
    }
    System.out.println("3) Move lever up to HIGH: " + testResult);

    // Attempt to move lever up past HIGH
    try {
        testCase.senseLeverUp();
        testResult = "FAIL";
    } catch (LeverErrorException e) {
        testResult = "PASS";
    }
    System.out.println("4) Attempt to move lever up past HIGH
        (catch DialErrorException): " + testResult);

    System.out.println("\nTests complete");
}

// Main method to run the test.
public static void main(String args[]) {
    testSenseLeverUp test = new testSenseLeverUp();
}
} // End of class.

```

Table 3.2 and table 3.3 show the respective test cases for the lever and dial components of the windshield wiper.

<i>Test Case</i>	<i>Preconditions (Instantiate Statement)</i>	<i>windshieldWiper Event (Method)</i>	<i>Expected Value of leverPos</i>
1	windshieldWiper(0, OFF, 1)	senseLeverUp()	INT
2	windshieldWiper(0, INT, 1)	senseLeverUp()	LOW
3	windshieldWiper(0, LOW, 1)	senseLeverUp()	HIGH
4	windshieldWiper(0, HIGH, 1)	senseLeverDown()	LOW
5	windshieldWiper(0, LOW, 1)	senseLeverDown()	INT
6	windshieldWiper(0, INT, 1)	senseLeverDown()	OFF

Table 3.2: Test cases for the windshield wiper lever [Jor06b]

<i>Test Case</i>	<i>Preconditions (Instantiate Statement)</i>	<i>windshieldWiper Event (Method)</i>	<i>Expected Value of dialPos</i>
1	windshieldWiper(0, INT, 1)	senseDialUp()	2
2	windshieldWiper(0, INT, 2)	senseDialUp()	3
3	windshieldWiper(0, INT, 3)	senseDialDown()	2
4	windshieldWiper(0, INT, 2)	senseDialDown()	1

Table 3.3: Test cases for the windshield wiper dial

Following the tests of the lever and dial, we should test the rest of the windshield wiper class by ensuring that when the lever and dial move up or down to the correct positions, the wiper speed gets set correctly. Again, we have a test driver, part of which is given below.

```
windshieldWiper testCase = new windshieldWiper(0, "OFF", 1);
System.out.println("\nTest 6: Test wiper speeds\nWindshield wiper
    initialised\nTesting commencing...\n");

// Move lever up.
try {
    testCase.senseLeverUp();
} catch (LeverErrorException e) {
    testResult = "FAIL";
}
wiperSpeed = testCase.getWiperSpeed();
if(wiperSpeed == 4) {
    testResult = "PASS";
}
else {
    testResult = "FAIL";
}
System.out.println("1) Wiper speed to 4: " + testResult);

// Simulate the dial moving up.
try {
    testCase.senseDialUp();
} catch (DialErrorException e) {
    testResult = "FAIL";
}
wiperSpeed = testCase.getWiperSpeed();
if(wiperSpeed == 6) {
    testResult = "PASS";
}
else {
    testResult = "FAIL";
}
System.out.println("2) Wiper speed to 6: " + testResult);
```

For this simple windshield wiper example it is feasible to have 100% test coverage. We can test the class by writing a test driver for the use case shown in figure 3.2, that simulates an actual use of the windshield wiper.

<i>UC1</i>	<i>Normal Usage</i>
Description	The windshield wiper is in the OFF position, and the Dial is at the 1 position; the user moves the lever to INT, and then moves the dial first to 2 and then to 3; the user then moves the lever to LOW and then to HIGH. The user moves the lever back to LOW, then INT, then to OFF.
Preconditions	The windshield wiper is in the OFF position, and the Dial is at the 1 position; wiper speed is 0.

<i>Event Sequence</i>	<i>User Action</i>	<i>System Response</i>
1	move lever to INT	Wiper speed is 4
2	move dial to 2	Wiper speed is 6
3	move dial to 3	Wiper speed is 12
4	move lever to LOW	Wiper speed is 30
5	move lever to HIGH	Wiper speed is 60
6	move lever to LOW	Wiper speed is 30
7	move lever to INT	Wiper speed is 12
8	move lever to OFF	Wiper speed is 0

Figure 3.2: Use case [Jor06b]

3.2.3 StateChart Diagram

Drawing a StateChart diagram for the class we are testing can aid in the selection of test cases. Using StateChart based tests encourages good test coverage of the class, as a StateChart shows:

- Every event
- Every state in a component
- Every transition in a component
- All pairs of interacting states (in different components)
- Scenarios corresponding to customer-defined use cases [Jor06b]

If we design test cases to test the items in the above list, we will have good test coverage. Figure 3.3 shows the StateChart for the windshield wiper class.

“... the three devices appear in the orthogonal components. In the Dial and lever components, transitions are caused by events, whereas the transitions in the wiper component are all caused by propositions that refer to what state is ‘active’ in the Dial or Lever orthogonal component.” [Jor06b]

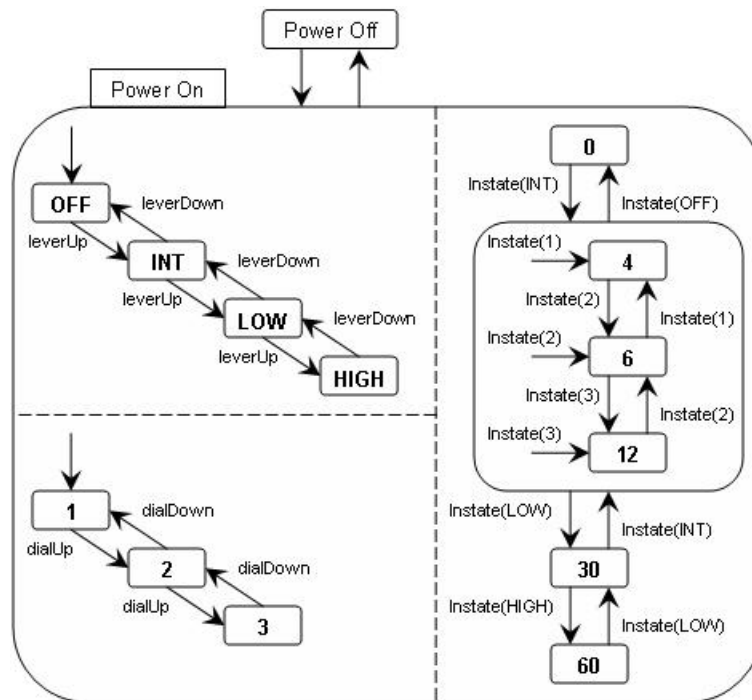


Figure 3.3: StateChart for the windshieldWiper class [Jor06b]

The test suite in appendix B has been designed with this StateChart in mind, testing every state of the lever and dial components, the transitions and events.

Chapter 4

JUnits

4.1 A Introduction to JUnits

A JUnit [JUn06] is a test driver for programs written in Java. JUnits make it much easier to write test cases and test suites. However, JUnits don't fully automate the testing process. Test cases are not generated automatically, and the tester must know what the correct outputs are for any inputs the program under test is given [Ola03].

“JUnit has a number of fundamental concepts. A test case is a Java class that tests some particular usage of the class in question. A test case consists of one or more test methods, which in turn test some component of the class. Multiple test cases can be combined into a test suite.” [DSW05]

To conduct a JUnit test, we use the provided methods such as `assertEquals`, `assertTrue` and `assertNull`. These allow us to compare actual results with the results we expect the program to produce [Ola03]. Below is a small segment of JUnit code, written to test the windshield wiper example, described in section 3.2.1. Full code for the JUnit to test the windshield wiper class is given in appendix C. It is not considered necessary to JUnit test accessor and modifier methods [JUn06].

```
public void testOff() {
    testCase = new windshieldWiper(0, "OFF", 1);
    try {
        testCase.senseLeverUp();
        assertEquals("Lever up error: ", "INT", testCase.getLeverPosition());
        assertEquals("Wiper speed error: ", 4, testCase.getWiperSpeed());
    } catch (LeverErrorException e) {
        fail("LeverErrorException not expected");
    }
}
```

The example shows a single test method from the test class used to test the up action of the windshield wiper lever. The up action of the lever is simulated, and then the `assertEquals` method is used to check that the new lever position is correct, and that the new wiper speed is correct. We can specify text to be displayed in case the test fails. There is also a `fail`

method used inside the `catch` statement, to show that if that statement is reached there has been an error. In this example, we do not expect an exception to be thrown.

4.2 JUnit Design

[Ola03] and [DSW05] give some pointers on how JUnit tests should be designed.

“Test methods must be public, void, take no arguments, and have a name that starts with ‘test’.” [DSW05]

The reason for starting test method names with ‘test’ is so that when test cases are added to test suites, it can easily be determined which methods should be added from each class.

In [Ola03], a paper on teaching JUnits during a programming course, a number of rules for designing JUnits are suggested:

- ***“Derive one TestClass from junit.TestCase for each target class.”***
For each class we want to test, we should have a `TestClass`. For example, for the `windshieldWiper` class we may have `TestwindshieldWiper`.
- ***“Define one nested class inside the TestClass for each collaborator of the target class.”***
As described previously, we want to test classes in isolation. When this is not possible, we may write a stub to take the place of any collaborators of the class under test. The ‘nested classes’ mentioned here refer to writing stubs with hard-coded behaviour.
- ***“Derive one TestClassMethod class from TestClass for each method of each target class.”***
We have a `TestClassMethod` class for each method in the class under test. This helps to modularize the test cases. The same approach was taken in section 3.2.2 when writing test drivers in the Java language. Appendix C has the following classes to test the individual methods in the windshield wiper class:

```
TestwindshieldWipersenseLeverUp
TestwindshieldWipersenseLeverDown
TestwindshieldWipersenseDialUp
TestwindshieldWipersenseDialDown
```

- ***“Implement one testScenario() method in the TestClassMethod class for each possible scenario of the target method.”***
We need to test all possible paths through a method. As we saw in the windshield wiper example, we can have multiple `case` and `if` statements, and so all of these possibilities need to be tested. In the `TestwindshieldWipersenseLeverUp` test class, we have the following `testScenario()` methods that simulate the lever being moved up from each position:

```
testOff
testInt
```

```
testLow
testHigh
```

- “*Derive one TestClassSuite class from junit.TestSuite for each target class.*”
The test suite allows us to organise test cases. To add test cases to a test suite we can either add each test method individually – which can be tedious if we have a large number of tests, or we can add all test methods from a test case [Ola03]. Below is the code for the JUnit test suite for the windshield wiper class.

```
public class TestwindshieldWiperSuite extends TestSuite {

    public static void main (String[] args) {
        junit.textui.TestRunner.run (suite());
    }

    public static Test suite() {
        TestSuite suite = new TestSuite("RUN ALL TESTS");
        suite.addTestSuite(TestwindshieldWipersenseLeverUp.class);
        suite.addTestSuite(TestwindshieldWipersenseLeverDown.class);
        suite.addTestSuite(TestwindshieldWipersenseDialUp.class);
        suite.addTestSuite(TestwindshieldWipersenseDialDown.class);
        return suite;
    }
}
```

4.3 JUnit Interface

Early versions of JUnit (version 3.8 downwards) came with a Swing GUI interface. The interface enables the user to see clearly which tests have been run, which are successful, and which fail or cause errors. Figure 4.1 is a screenshot of the JUnit interface after the 14 tests on the windshield wiper class have been run.

Figure 4.2 also shows the JUnit interface, but this time there are failures in the tests. An exception has been thrown where it was not meant to, an expected exception was not thrown, and one wiper speed is incorrect. The interface shows clearly which tests have failed, and why they have failed.

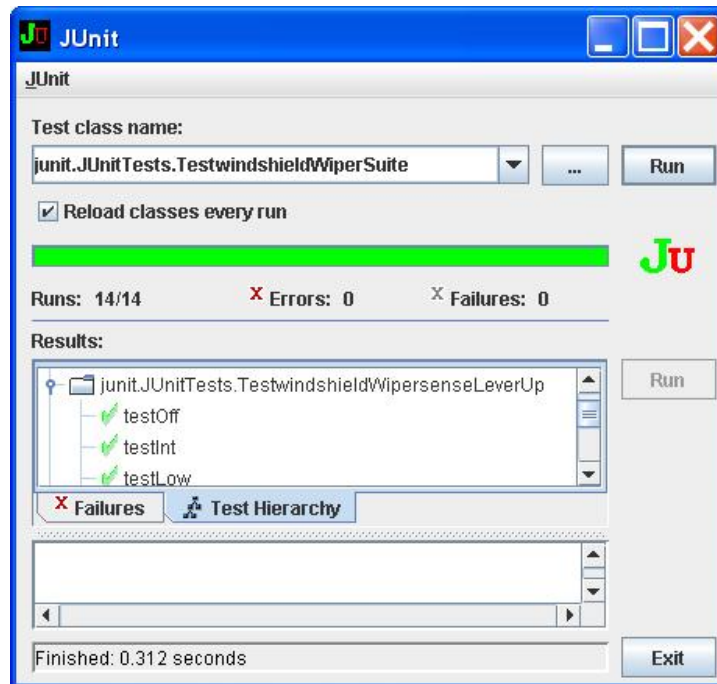


Figure 4.1: JUnit Interface – successful tests

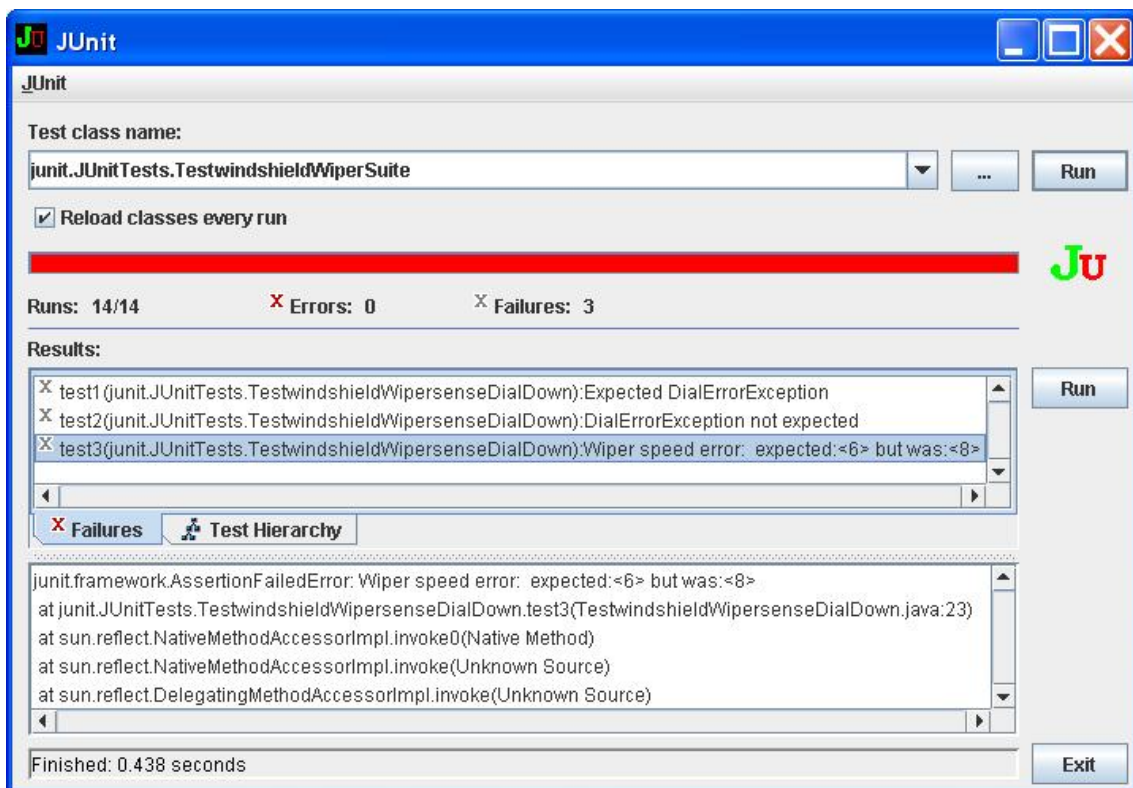


Figure 4.2: JUnit Interface – failed tests

Chapter 5

Class Testing: Part Two

So far we have just examined a simple example to investigate class testing. However, to consider the techniques of class testing more thoroughly, we must consider situations where inheritance is present.

“Classes are used to define new classes, or subclasses, through a relation known as inheritance. Inheritance imposes a hierarchical organisation on the classes and permits a subclass to inherit attributes from its parent classes and either extend, restrict or redefine them. Subclasses may ... contain new attributes not possessed by the parent ...” [MHF92]

Inheritance causes difficulties in class testing. We may initially decide to test a base class using the methods described previously, and then test the derived classes, using stubs where necessary. This however means that subclasses are completely retested, even though their inherited attributes and operations would have been tested in the parent class. This results in extra work that is not really necessary.

“Additionally, completely retesting each class does not exploit opportunities to reuse and share the design, construction and execution of test suites.” [MHF92]

The question is: How much testing should we perform on a subclass? This leads to two more specific questions from [Bin99]:

- Should inherited methods be retested?
- Can we reuse superclass tests for inherited and overridden methods?

There are two main techniques for class testing programs with inheritance that address these questions: *Flattened classes* and *Incremental class testing*.

5.1 Flattened Class Testing

A *flattened class* is a class such that a subclass that is under test and its superclass are merged into a single class. The concept is shown in figure 5.1.

Class A contains three methods, one of which is private and can't be inherited. Class B is a subclass of A, and inherits method A.1. It also has three of its own methods, one of which is

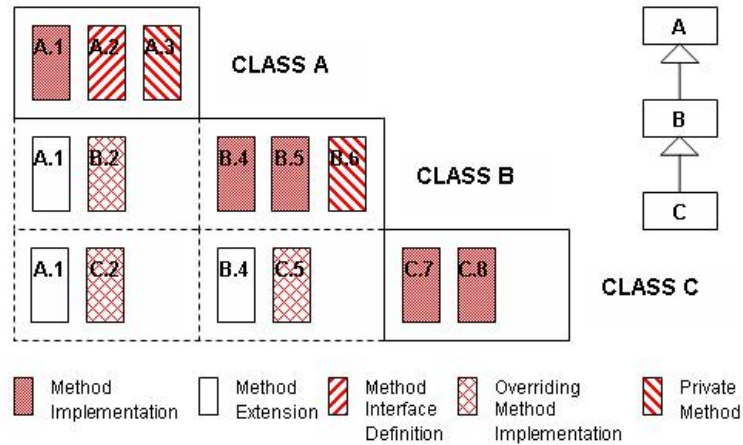


Figure 5.1: Flattened classes [Bin99]

private. Class C is a subclass of B, and inherits A.1 and B.4, has overriding implementations for B.2 and B.5 as well as its own methods.

This flattening technique can become more confusing to apply as the inheritance hierarchy grows. Also, the testing that was performed on the superclass is not used in testing the subclass, so we get duplicate testing [MHF92]. This problem is made worse when a class has multiple subclasses, as show in figure 5.2. We end up with numerous flattened classes that mean more duplicate tests.

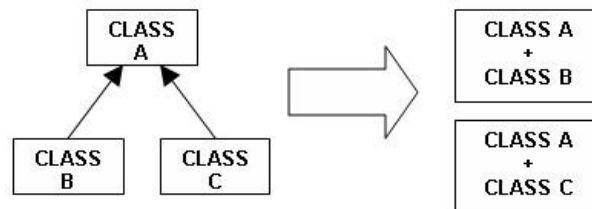


Figure 5.2: Flattened classes

A further problem with using flattened classes, is that the class isn't supposed to be flattened. We are not testing in the way the classes are meant to be used – they will not be flattened in the final implementation. This may mean that errors where the classes interact may not be found at this stage of testing.

5.2 Incremental Class Testing

The technique of *Incremental class testing* is described in [Bin99], [DSW05] and [Jr96]. First of all, the upper most class in a class hierarchy (the base class) is tested thoroughly by testing all attributes and methods. Each attribute that is tested has a *test history* associated with it. When a subclass of a base class is created, as well as inheriting the attributes and methods it also inherits the test history.

The inherited test history can be used as a base to develop a test suite for the derived class. As well as the tests on any inherited methods, the test history must be updated so that it includes tests for the methods that are not present in the superclass.

“We retest inherited attributes in the context of the subclass by identifying and testing their interactions with newly defined attributes in the subclass.” [DSW05]

Although retesting inherited methods seems like duplicate work, it is really only minimal – where appropriate, the test cases from the parent test suite can be used to test the subclass.

The advantage of incremental class testing is that time is saved by using the testing history of the superclass to develop a suitable test suite for the derived class. It is only the new or overwritten methods that new tests must be written for.

5.3 Inheritance Bugs

[Bin99], a book specifically on the subject of testing object-oriented systems suggests a number of inheritance-related bugs that class testing should aim to look for. These are presented as follows:

- ***Incorrect Initialization:*** Superclass initialization is omitted or incorrect.
- ***Missing Override:*** A subclass-specific implementation of a superclass method is omitted. This means that the superclass method may be incorrectly bound to a subclass object, when the behaviour of that method is not entirely suitable.
- ***Naked Access:*** A superclass variable is visible in a subclass and subclass methods can update these variables directly. This is the same problem as having unrestricted access to global data. Changes to the variable by the subclass can have undesirable knock-on effects to the superclass.
- ***Square Peg in a Round Hole:*** A subclass inherits a method from the superclass that is not suitable for it to use.
- ***Spaghetti Inheritance:*** Multiple inheritance (a subclass inheriting from two or more parent classes) or a deep hierarchy of inheritance can cause confusion that may result in errors.
- ***Weird Inheritance:*** Inheritance is used in order to share code with another class. This is called *convenience inheritance* and the practise is severely frowned upon.
- ***Fat Inheritance:*** A subclass inherits methods from the superclass that are inappropriate or irrelevant for it to use. [Bin99]

Conclusion

Testing is a key step in the development of software. Testing allows us to prove to the customer that the software meets the requirements that they have specified. It also helps us to find any bugs that are in the software, so they can be corrected before the product is released. Of course, just because we have performed what we believe to be *adequate* testing on a program, does not mean it is completely error free.

Testing costs time and money, and it is often not feasible to completely test a program. We cannot give it all possible inputs or test all possible combinations of paths through the program. We can only test a subset of these, and we design our test cases so that they have a high chance of finding certain types of errors. Although it should not be completely relied upon, the percentage of *test coverage* a program receives can give a good indication as to how adequately a program has been tested.

The focus of this report has been *Class Testing*. Class testing is a method of testing object-oriented programs. It is the first testing that an object-oriented program will receive. We consider a class to be an individual unit to test, and where possible we test these units in isolation from other units. Where collaboration with another unit is needed we may write a *stub*: a dummy subprogram that simulates the action of the real class. It may simply return some constant value to the class that uses it.

We can write *test drivers* to partially automate the process of class testing. These drivers are main programs that call the methods we are testing, and pass in any necessary parameters. They may contain an *oracle* to determine whether the program operated correctly for each input. Languages such as Java have tools available (JUnits in Java) that make the process of creating a test driver much simpler. Some have interfaces to display the results of tests, and indicate why a test found an error. In this document we have seen examples of manually written test drivers and JUnit test drivers.

Class testing is an example of both *black-box* and *white-box* testing. We can use the program specification to test that the program has the desired functionality. We know we want a method to perform a certain task, we don't care how the method does it, only that it works. However, we do need to test the actual implementation as well. We need to execute all lines of code at least once, ensure that all branches in conditional statements are entered correctly, and test the conditionals at their input boundaries. We may also want to test all independent paths of execution through the program. We can design test cases using a multitude of strategies, to ensure that we find the largest array of errors. At the same time, we want to minimise the number of test cases we must run, in order to save time, and indeed money.

One of the more complicated issues with class testing is class testing a program where inheritance is present. How can we test units in isolation when they rely on inherited attributes from other classes? This report has discussed two options for dealing with this problem. The

first was to use *flattened classes*, whereby the base class is tested thoroughly first, and then the inheritable attributes are merged with the derived class for them to be tested together. The second option – *incremental class testing* – was to again test the base class first, but this time to keep a *test history* for each attribute. This test history is inherited by the subclass along with the attributes, and is used as a basis for testing the subclass. New tests are written for attributes not in the parent class, but where appropriate the parent class test suite can be used to test the subclass.

Class testing is a field in which research is still being done to improve the process. While it can be partially automated by way of test drivers and oracles, it is not yet possible to completely automate the process of designing suitable test cases. This still requires human interaction.

The next stage of object-oriented software testing is *Integration Testing*. In this, the units we have previously tested are combined into larger parts of the program (this can be done in a number of ways, each with its own merits and demerits), and tested to ensure they interact properly and work together as one.

Appendix A

Windshield Wiper Java Implementation

The Java source code for the windshield wiper class.

```
public class windshieldWiper {
    // State variables.
    private int wiperSpeed;
    private String leverPosition;
    private int dialPosition;

    public windshieldWiper(int wiperSpeed, String leverPosition, int dialPosition) {
        this.wiperSpeed = wiperSpeed;
        this.leverPosition = leverPosition;
        this.dialPosition = dialPosition;
    }

    // Get the current wiper speed.
    public int getWiperSpeed() {
        return wiperSpeed;
    }

    // Set the wiper speed.
    public void setWiperSpeed(int wiperSpeed) {
        this.wiperSpeed = wiperSpeed;
    }

    // Get the current position of the lever.
    public String getLeverPosition() {
        return leverPosition;
    }

    // Set the lever position.
    public void setLeverPosition(String leverPosition) {
        this.leverPosition = leverPosition;
    }
}
```

```
}

// Get the current position of the dial.
public int getDialPosition() {
    return dialPosition;
}

// Set the dial position.
public void setDialPosition(int dialPosition) {
    this.dialPosition = dialPosition;
}

// Move the lever up and set new wiper speed.
public void senseLeverUp() throws LeverErrorException {
    if(leverPosition.equals("OFF")) {
        leverPosition = "INT";

        switch(dialPosition) {
            case 1: {
                wiperSpeed = 4;
                break;
            }
            case 2: {
                wiperSpeed = 6;
                break;
            }
            case 3: {
                wiperSpeed = 12;
                break;
            }
        }
    }
    else if(leverPosition.equals("INT")) {
        leverPosition = "LOW";
        wiperSpeed = 30;
    }
    else if(leverPosition.equals("LOW")) {
        leverPosition = "HIGH";
        wiperSpeed = 60;
    }
    else if(leverPosition.equals("HIGH")) {
        System.out.println("ERROR: leverPosition already at HIGH.");
        throw new LeverErrorException();
    }
}

// Move the lever down and set new wiper speed.
```

```
public void senseLeverDown() throws LeverErrorException {
    if(leverPosition.equals("OFF")) {
        System.out.println("ERROR: leverPosition already at OFF.");
        throw new LeverErrorException();
    }
    else if(leverPosition.equals("INT")) {
        leverPosition = "OFF";
        wiperSpeed = 0;
    }
    else if(leverPosition.equals("LOW")) {
        leverPosition = "INT";
        switch(dialPosition) {
            case 1: {
                wiperSpeed = 4;
                break;
            }
            case 2: {
                wiperSpeed = 6;
                break;
            }
            case 3: {
                wiperSpeed = 12;
                break;
            }
        }
    }
    else if(leverPosition.equals("HIGH")) {
        leverPosition = "LOW";
        wiperSpeed = 30;
    }
}

// Move the dial up and set the wiper speed.
public void senseDialUp() throws DialErrorException {
    if(dialPosition == 3) {
        System.out.println("Already on highest setting");
        throw new DialErrorException();
    }
    else {
        dialPosition++;

        if(leverPosition.equals("INT")) {
            switch(dialPosition) {
                case 1: {
                    wiperSpeed = 4;
                    break;
                }
            }
        }
    }
}
```

```
        case 2: {
            wiperSpeed = 6;
            break;
        }
        case 3: {
            wiperSpeed = 12;
            break;
        }
    }
}
else {
    // No Action
}
}
}

// Move the dial down and set the wiper speed.
public void senseDialDown() throws DialErrorException {
    if(dialPosition == 1) {
        System.out.println("Already on lowest setting");
        throw new DialErrorException();
    }
    else {
        dialPosition--;

        if(leverPosition.equals("INT")) {
            switch(dialPosition) {
                case 1: {
                    wiperSpeed = 4;
                    break;
                }
                case 2: {
                    wiperSpeed = 6;
                    break;
                }
                case 3: {
                    wiperSpeed = 12;
                    break;
                }
            }
        }
        else {
            // No action.
        }
    }
}
} // End of class.
```

The LeverErrorException class:

```
public class LeverErrorException extends Exception {
    public LeverErrorException() {
        super();
    }

    public LeverErrorException(String error) {
        super(error);
    }
}
```

The DialErrorException class:

```
public class DialErrorException extends Exception {
    public DialErrorException() {
        super();
    }

    public DialErrorException(String error) {
        super(error);
    }
}
```

Appendix B

Windshield Wiper Test Driver

Test driver for the accessor and modifier methods:

```
public class testGetSet {
    // Variables for storing results.
    private int wiperSpeed;
    private String leverPos;
    private int dialPos;
    private String testResult;

    public testGetSet() {
        // Create a new windshield wiper object, initially set
        // the wiper speed to 0, lever position to OFF
        // and dial position to 1.
        windshieldWiper testCase = new windshieldWiper(0, "OFF", 1);
        System.out.println("\nTest 1: Test Get/Set Methods
            \nWindshield wiper initialised\nTesting commencing...\n");

        // Set the wiper speed and check result.
        testCase.setWiperSpeed(6);
        wiperSpeed = testCase.getWiperSpeed();
        if(wiperSpeed == 6) {
            testResult = "PASS";
        }
        else {
            testResult = "FAIL";
        }
        System.out.println("1) Set wiper speed to 6: " + testResult);

        // Set the lever position and check result.
        testCase.setLeverPosition("INT");
        leverPos = testCase.getLeverPosition();
        if(leverPos.equals("INT")) {
            testResult = "PASS";
        }
    }
}
```

```
        else {
            testResult = "FAIL";
        }
        System.out.println("2) Set lever position to INT: " + testResult);

        // Set the dial position and check result.
        testCase.setDialPosition(2);
        dialPos = testCase.getDialPosition();
        if(dialPos == 2) {
            testResult = "PASS";
        }
        else {
            testResult = "FAIL";
        }
        System.out.println("3) Set dial position to 2: " + testResult);

        System.out.println("\nTests complete");
    }

    // Main method to run the tests.
    public static void main(String args[]) {
        testGetSet test = new testGetSet();
    }
} // End of class.
```

Test driver for the senseLeverUp method:

```
public class testSenseLeverUp {
    // Variables for storing results.
    private int wiperSpeed;
    private String leverPos;
    private int dialPos;
    private String testResult;

    public testSenseLeverUp() {
        windshieldWiper testCase = new windshieldWiper(0, "OFF", 1);
        System.out.println("\nTest 2: Test senseLeverUp\nWindshield
            wiper initialised\nTesting commencing...\n");

        // Simulate the lever moving up.
        try {
            testCase.senseLeverUp();
        } catch (LeverErrorException e) {
            testResult = "FAIL";
        }
        leverPos = testCase.getLeverPosition();
    }
}
```

```
    if(leverPos.equals("INT")) {
        testResult = "PASS";
    }
    else {
        testResult = "FAIL";
    }
    System.out.println("1) Move lever up to INT: " + testResult);

    // Simulate the lever moving up.
    try {
        testCase.senseLeverUp();
    } catch (LeverErrorException e) {
        testResult = "FAIL";
    }
    leverPos = testCase.getLeverPosition();
    if(leverPos.equals("LOW")) {
        testResult = "PASS";
    }
    else {
        testResult = "FAIL";
    }
    System.out.println("2) Move lever up to LOW: " + testResult);

    // Simulate the lever moving up.
    try {
        testCase.senseLeverUp();
    } catch (LeverErrorException e) {
        testResult = "FAIL";
    }
    leverPos = testCase.getLeverPosition();
    if(leverPos.equals("HIGH")) {
        testResult = "PASS";
    }
    else {
        testResult = "FAIL";
    }
    System.out.println("3) Move lever up to HIGH: " + testResult);

    // Attempt to move lever up past HIGH
    try {
        testCase.senseLeverUp();
        testResult = "FAIL";
    } catch (LeverErrorException e) {
        testResult = "PASS";
    }
    System.out.println("4) Attempt to move lever up past HIGH
        (catch DialErrorException): " + testResult);
```

```
        System.out.println("\nTests complete");
    }

    // Main method to run the test.
    public static void main(String args[]) {
        testSenseLeverUp test = new testSenseLeverUp();
    }
} // End of class.
```

Test driver for the senseLeverDown method:

```
public class testSenseLeverDown {
    // Variables for storing results.
    private int wiperSpeed;
    private String leverPos;
    private int dialPos;
    private String testResult;

    public testSenseLeverDown() {
        windshieldWiper testCase = new windshieldWiper(0, "HIGH", 1);
        System.out.println("\nTest 3: Test senseLeverDown\nWindshield
            wiper initialised\nTesting commencing...\n");

        // Simulate the lever moving down.
        try {
            testCase.senseLeverDown();
        } catch (LeverErrorException e) {
            testResult = "FAIL";
        }
        leverPos = testCase.getLeverPosition();
        if(leverPos.equals("LOW")) {
            testResult = "PASS";
        }
        else {
            testResult = "FAIL";
        }
        System.out.println("1) Move lever down to LOW: " + testResult);

        // Simulate the lever moving down.
        try {
            testCase.senseLeverDown();
        } catch (LeverErrorException e) {
            testResult = "FAIL";
        }
        leverPos = testCase.getLeverPosition();
    }
}
```

```
        if(leverPos.equals("INT")) {
            testResult = "PASS";
        }
        else {
            testResult = "FAIL";
        }
        System.out.println("2) Move lever down to INT: " + testResult);

        // Simulate the lever moving down.
        try {
            testCase.senseLeverDown();
        } catch (LeverErrorException e) {
            testResult = "FAIL";
        }
        leverPos = testCase.getLeverPosition();
        if(leverPos.equals("OFF")) {
            testResult = "PASS";
        }
        else {
            testResult = "FAIL";
        }
        System.out.println("3) Move lever down to OFF: " + testResult);

        // Attempt to move the lever down past the OFF position
        try {
            testCase.senseLeverDown();
            testResult = "FAIL";
        } catch (LeverErrorException e) {
            testResult = "PASS";
        }
        System.out.println("4) Attempt to move lever down past OFF
            (catch LeverErrorException): " + testResult);

        System.out.println("\nTest complete");
    }

    // Main method to run the test.
    public static void main(String args[]) {
        testSenseLeverDown test = new testSenseLeverDown();
    }
} // End of class.
```

Test driver for the senseDialUp method:

```
public class testSenseDialUp {
    // Variables for storing results.
```

```
private int wiperSpeed;
private String leverPos;
private int dialPos;
private String testResult;

public testSenseDialUp() {
    windshieldWiper testCase = new windshieldWiper(0, "INT", 1);
    System.out.println("\nTest 4: Test senseDialUp\nWindshield
        wiper initialised\nTesting commencing...\n");

    // Simulate the dial moving up.
    try {
        testCase.senseDialUp();
    } catch (DialErrorException e) {
        testResult = "FAIL";
    }
    dialPos = testCase.getDialPosition();
    if(dialPos == 2) {
        testResult = "PASS";
    }
    else {
        testResult = "FAIL";
    }
    System.out.println("1) Move dial up to 2: " + testResult);

    // Simulate the dial moving up.
    try {
        testCase.senseDialUp();
    } catch (DialErrorException e) {
        testResult = "FAIL";
    }
    dialPos = testCase.getDialPosition();
    if(dialPos == 3) {
        testResult = "PASS";
    }
    else {
        testResult = "FAIL";
    }
    System.out.println("2) Move dial up to 3: " + testResult);

    // Attempt to move dial up past position 3
    try {
        testCase.senseDialUp();
        testResult = "FAIL";
    } catch (DialErrorException e) {
        testResult = "PASS";
    }
}
```

```
        System.out.println("3) Attempt to move dial up past 3
            (catch DialErrorException): " + testResult);

        System.out.println("\nTests complete");
    }

    // Main method to run the test.
    public static void main(String args[]) {
        testSenseDialUp test = new testSenseDialUp();
    }
} // End of class.
```

Test driver for the senseDialDown method:

```
public class testSenseDialDown {
    // Variables for storing results.
    private int wiperSpeed;
    private String leverPos;
    private int dialPos;
    private String testResult;

    public testSenseDialDown() {
        windshieldWiper testCase = new windshieldWiper(0, "INT", 3);
        System.out.println("\nTest 5: Test senseDialDown\nWindshield
            wiper initialised\nTesting commencing...\n");

        // Simulate the dial moving up.
        try {
            testCase.senseDialDown();
        } catch (DialErrorException e) {
            testResult = "FAIL";
        }
        dialPos = testCase.getDialPosition();
        if(dialPos == 2) {
            testResult = "PASS";
        }
        else {
            testResult = "FAIL";
        }
        System.out.println("1) Move dial down to 2: " + testResult);

        // Simulate the dial moving up.
        try {
            testCase.senseDialDown();
        } catch (DialErrorException e) {
            testResult = "FAIL";
        }
    }
}
```

```
    }
    dialPos = testCase.getDialPosition();
    if(dialPos == 1) {
        testResult = "PASS";
    }
    else {
        testResult = "FAIL";
    }
    System.out.println("2) Move dial down to 1: " + testResult);

    // Attempt to move dial down past position 1
    try {
        testCase.senseDialDown();
        testResult = "FAIL";
    } catch (DialErrorException e) {
        testResult = "PASS";
    }
    System.out.println("3) Attempt to move dial down past 1
        (catch DialErrorException): " + testResult);

    System.out.println("\nTests complete");
}

// Main method to run the test.
public static void main(String args[]) {
    testSenseDialDown test = new testSenseDialDown();
}
} // End of class.
```

Test driver for the wiper speeds:

```
public class testWindshieldWiper {
    // Variables for storing results.
    private int wiperSpeed;
    private String leverPos;
    private int dialPos;
    private String testResult;

    public testWindshieldWiper() {
        windshieldWiper testCase = new windshieldWiper(0, "OFF", 1);
        System.out.println("\nTest 6: Test wiper speeds\nWindshield
            wiper initialised\nTesting commencing...\n");

        // Move lever up.
        try {
            testCase.senseLeverUp();
```

```
    } catch (LeverErrorException e) {
        testResult = "FAIL";
    }
    wiperSpeed = testCase.getWiperSpeed();
    if(wiperSpeed == 4) {
        testResult = "PASS";
    }
    else {
        testResult = "FAIL";
    }
    System.out.println("1) Wiper speed to 4: " + testResult);

    // Simulate the dial moving up.
    try {
        testCase.senseDialUp();
    } catch (DialErrorException e) {
        testResult = "FAIL";
    }
    wiperSpeed = testCase.getWiperSpeed();
    if(wiperSpeed == 6) {
        testResult = "PASS";
    }
    else {
        testResult = "FAIL";
    }
    System.out.println("2) Wiper speed to 6: " + testResult);

    // Simulate the dial moving up.
    try {
        testCase.senseDialUp();
    } catch (DialErrorException e) {
        testResult = "FAIL";
    }
    wiperSpeed = testCase.getWiperSpeed();
    if(wiperSpeed == 12) {
        testResult = "PASS";
    }
    else {
        testResult = "FAIL";
    }
    System.out.println("3) Wiper speed to 12: " + testResult);

    // Move lever up.
    try {
        testCase.senseLeverUp();
    } catch (LeverErrorException e) {
        testResult = "FAIL";
    }
}
```

```
    }
    wiperSpeed = testCase.getWiperSpeed();
    if(wiperSpeed == 30) {
        testResult = "PASS";
    }
    else {
        testResult = "FAIL";
    }
    System.out.println("4) Wiper speed to 30: " + testResult);

    // Move lever up.
    try {
        testCase.senseLeverUp();
    } catch (LeverErrorException e) {
        testResult = "FAIL";
    }
    wiperSpeed = testCase.getWiperSpeed();
    if(wiperSpeed == 60) {
        testResult = "PASS";
    }
    else {
        testResult = "FAIL";
    }
    System.out.println("5) Wiper speed to 60: " + testResult);

    // Move lever down.
    try {
        testCase.senseLeverDown();
    } catch (LeverErrorException e) {
        testResult = "FAIL";
    }
    wiperSpeed = testCase.getWiperSpeed();
    if(wiperSpeed == 30) {
        testResult = "PASS";
    }
    else {
        testResult = "FAIL";
    }
    System.out.println("6) Wiper speed to 30: " + testResult);

    // Move lever down.
    try {
        testCase.senseLeverDown();
    } catch (LeverErrorException e) {
        testResult = "FAIL";
    }
    wiperSpeed = testCase.getWiperSpeed();
```

```
        if(wiperSpeed == 12) {
            testResult = "PASS";
        }
        else {
            testResult = "FAIL";
        }
        System.out.println("7) Wiper speed to 12: " + testResult);

        // Move lever down.
        try {
            testCase.senseLeverDown();
        } catch (LeverErrorException e) {
            testResult = "FAIL";
        }
        wiperSpeed = testCase.getWiperSpeed();
        if(wiperSpeed == 0) {
            testResult = "PASS";
        }
        else {
            testResult = "FAIL";
        }
        System.out.println("8) Wiper speed to 0: " + testResult);

        System.out.println("\nTests complete");
    }

    // Main method to run the test.
    public static void main(String args[]) {
        testWindshieldWiper test = new testWindshieldWiper();
    }
} // End of class.
```

A class to run all of the tests:

```
public class testSuiteWindshieldWiper {

    public testSuiteWindshieldWiper() {
        // Run all 6 tests
        testGetSet test1 = new testGetSet();
        testSenseLeverUp test2 = new testSenseLeverUp();
        testSenseLeverDown test3 = new testSenseLeverDown();
        testSenseDialUp test4 = new testSenseDialUp();
        testSenseDialDown test5 = new testSenseDialDown();
        testWindshieldWiper test6 = new testWindshieldWiper();

        System.out.println("\nAll testing complete");
    }
}
```

```
    }  
  
    public static void main(String args[]) {  
        testSuiteWindshieldWiper testsuite = new testSuiteWindshieldWiper();  
    }  
} // End of class.
```

Appendix C

JUnit Source Code

The JUnit test class for the windshield wiper class `senseLeverUp` method:

```
import junit.framework.*;

public class TestwindshieldWipersenseLeverUp extends TestCase {
    public windshieldWiper testCase;

    public TestwindshieldWipersenseLeverUp(String name) {
        super(name);
    }

    public static void main (String[] args) {
        junit.swingui.TestRunner.run(TestwindshieldWipersenseLeverUp.class);
    }

    public void setUp() {
    }

    // Test lever moving from OFF to INT
    public void testOff() {
        testCase = new windshieldWiper(0, "OFF", 1);
        try {
            testCase.senseLeverUp();
            assertEquals("Lever up error: ", "INT", testCase.getLeverPosition());
            assertEquals("Wiper speed error: ", 4, testCase.getWiperSpeed());
        } catch (LeverErrorException e) {
            fail("LeverErrorException not expected");
        }
    }

    // Test lever moving from INT to LOW
    public void testInt() {
        testCase = new windshieldWiper(4, "INT", 1);
        try {
```

```
        testCase.senseLeverUp();
        assertEquals("Lever up error: ", "LOW", testCase.getLeverPosition());
        assertEquals("Wiper speed error: ", 30, testCase.getWiperSpeed());
    } catch (LeverErrorException e) {
        fail("LeverErrorException not expected");
    }
}

// Test lever moving from LOW to HIGH
public void testLow() {
    testCase = new windshieldWiper(30, "LOW", 1);
    try {
        testCase.senseLeverUp();
        assertEquals("Lever up error: ", "HIGH", testCase.getLeverPosition());
        assertEquals("Wiper speed error: ", 60, testCase.getWiperSpeed());
    } catch (LeverErrorException e) {
        fail("LeverErrorException not expected");
    }
}

// Attempt to move lever up from the HIGH position
public void testHigh() {
    testCase = new windshieldWiper(60, "HIGH", 1);
    try {
        testCase.senseLeverUp();
        fail("Expected LeverErrorException");
    } catch (LeverErrorException e) {
        // Exception test succeeded
    }
}
} // End of class
```

The JUnit test class for the windshield wiper class `senseLeverDown` method:

```
import junit.framework.*;

public class TestwindshieldWipersenseLeverDown extends TestCase {
    public windshieldWiper testCase;

    public TestwindshieldWipersenseLeverDown(String name) {
        super(name);
    }

    public static void main (String[] args) {
        junit.swingui.TestRunner.run(TestwindshieldWipersenseLeverDown.class);
    }
}
```

```
public void setUp() {
}

// Test lever moving from HIGH to LOW
public void testHigh(){
    testCase = new windshieldWiper(60, "HIGH", 1);
    try {
        testCase.senseLeverDown();
        assertEquals("Lever down error: ", "LOW", testCase.getLeverPosition());
        assertEquals("Wiper speed error: ", 30, testCase.getWiperSpeed());
    } catch (LeverErrorException e) {
        fail("LeverErrorException not expected");
    }
}

// Test lever moving from LOW to INT
public void testLow() {
    testCase = new windshieldWiper(30, "LOW", 1);
    try {
        testCase.senseLeverDown();
        assertEquals("Lever down error: ", "INT", testCase.getLeverPosition());
        assertEquals("Wiper speed error: ", 4, testCase.getWiperSpeed());
    } catch (LeverErrorException e) {
        fail("LeverErrorException not expected");
    }
}

// Test lever moving from INT to OFF
public void testInt() {
    testCase = new windshieldWiper(4, "INT", 1);
    try {
        testCase.senseLeverDown();
        assertEquals("Lever down error: ", "OFF", testCase.getLeverPosition());
        assertEquals("Wiper speed error: ", 0, testCase.getWiperSpeed());
    } catch (LeverErrorException e) {
        fail("LeverErrorException not expected");
    }
}

// Attempt to move lever down past the OFF position
public void testOff() {
    testCase = new windshieldWiper(0, "OFF", 1);
    try {
        testCase.senseLeverDown();
        fail("Expected LeverErrorException");
    } catch (LeverErrorException e) {
```

```
        // Exception test succeeded
    }
}
} // End of class.
```

The JUnit test class for the windshield wiper class `senseDialUp` method:

```
import junit.framework.*;

public class TestwindshieldWipersenseDialUp extends TestCase {
    public windshieldWiper testCase;

    public TestwindshieldWipersenseDialUp(String name) {
        super(name);
    }

    public static void main (String[] args) {
        junit.swingui.TestRunner.run(TestwindshieldWipersenseDialUp.class);
    }

    public void setUp() {
    }

    // Test dial moving up from position 1 to position 2
    public void test1(){
        testCase = new windshieldWiper(4, "INT", 1);
        try {
            testCase.senseDialUp();
            assertEquals("Dial up error: ", 2, testCase.getDialPosition());
            assertEquals("Wiper speed error: ", 6, testCase.getWiperSpeed());
        } catch (DialErrorException e) {
            fail("DialErrorException not expected");
        }
    }

    // Test dial moving up from position 2 to position 3
    public void test2() {
        testCase = new windshieldWiper(6, "INT", 2);
        try {
            testCase.senseDialUp();
            assertEquals("Dial up error: ", 3, testCase.getDialPosition());
            assertEquals("Wiper speed error: ", 12, testCase.getWiperSpeed());
        } catch (DialErrorException e) {
            fail("DialErrorException not expected");
        }
    }
}
```

```
// Attempt to move dial up past position 3
public void test3() {
    testCase = new windshieldWiper(12, "INT", 3);
    try {
        testCase.senseDialUp();
        fail("Expected DialErrorException");
    } catch (DialErrorException e) {
        // Exception test succeeded
    }
}
} // End of class
```

The JUnit test class for the windshield wiper class senseDialDown method:

```
import junit.framework.*;

public class TestwindshieldWipersenseDialDown extends TestCase {
    public windshieldWiper testCase;

    public TestwindshieldWipersenseDialDown(String name) {
        super(name);
    }

    public static void main (String[] args) {
        junit.swingui.TestRunner.run(TestwindshieldWipersenseDialDown.class);
    }

    public void setUp() {
    }

    // Test moving dial from position 3 to position 2
    public void test3(){
        testCase = new windshieldWiper(12, "INT", 3);
        try {
            testCase.senseDialDown();
            assertEquals("Dial down error: ", 2, testCase.getDialPosition());
            assertEquals("Wiper speed error: ", 6, testCase.getWiperSpeed());
        } catch (DialErrorException e) {
            fail("DialErrorException not expected");
        }
    }

    // Test moving dial from position 2 to position 1
    public void test2() {
        testCase = new windshieldWiper(6, "INT", 2);
    }
}
```

```
        try {
            testCase.senseDialDown();
            assertEquals("Dial down error: ", 1, testCase.getDialPosition());
            assertEquals("Wiper speed error: ", 4, testCase.getWiperSpeed());
        } catch (DialErrorException e) {
            fail("DialErrorException not expected");
        }
    }

    // Attempt to move dial down past position 1
    public void test1() {
        testCase = new windshieldWiper(4, "INT", 1);
        try {
            testCase.senseDialDown();
            fail("Expected DialErrorException");
        } catch (DialErrorException e) {
            // Exception test succeeded
        }
    }
} // End of class
```

The JUnit test suite for the windshield wiper class:

```
import junit.framework.*;

public class TestwindshieldWiperSuite extends TestSuite {

    // Main method to run the test suite
    public static void main (String[] args) {
        junit.textui.TestRunner.run (suite());
    }

    // Create the test suite
    public static Test suite() {
        TestSuite suite = new TestSuite("RUN ALL TESTS");
        suite.addTestSuite(TestwindshieldWipersenseLeverUp.class);
        suite.addTestSuite(TestwindshieldWipersenseLeverDown.class);
        suite.addTestSuite(TestwindshieldWipersenseDialUp.class);
        suite.addTestSuite(TestwindshieldWipersenseDialDown.class);
        return suite;
    }
} // End of class.
```

Bibliography

- [Bei90] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 2nd edition, 1990.
- [Bin99] Robert Binder. *Testing Object-Oriented Systems: Models, Patterns and Tools*. Addison Wesley, 1999.
- [CM90] Thomas Cheatham and Lee Mellinger. Testing object-oriented software systems. In *Proceedings of the 1990 ACM annual conference on Cooperation*, pages 161–165. ACM Press, 1990.
- [DGA94] Yin Zhong Dechang Gu and Sarwar Ali. On testing of classes in object-oriented programs. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 22. IBM Press, 1994.
- [DSW05] Michael Wick Daniel Stevenson and Paul Wagner. Unit testing and junit across the curriculum. In *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 236–240. ACM Press, 2005.
- [Fie89] Steven Fiedler. Object-oriented unit testing. *Hewlett-Packard Journal*, 40:69–74, 1989.
- [Jor06a] Paul Jorgensen. *Software Testing: A Craftsman’s Approach*, chapter 1. Auerbach Pub, 2006.
- [Jor06b] Paul Jorgensen. *Software Testing: A Craftsman’s Approach*, chapter 17. Auerbach Pub, 2006.
- [Jr96] Morris Johnson Jr. A survey of testing techniques for object-oriented systems. In *Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1996.
- [JUn06] Junit.org. Website, December 2006. www.junit.org.
- [MHF92] John McGregor Mary Harrold and Kevin Fitzpatrick. Incremental testing of object-oriented class structures. In *Proceedings of the 14th international conference on Software engineering*, pages 68–80. ACM Press, 1992.
- [Mye79] Glenford Myers. *The Art Of Software Testing*. John Wiley & Sons, 1979.
- [Ola03] Michael Olan. Unit testing: Test early, test often. In *Journal of Computing Sciences in Colleges*, pages 319–328. Consortium for Computing Sciences in Colleges, 2003.

- [Pre04] Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw Hill Higher Education, 2004.
- [Som06] Ian Sommerville. *Software Engineering*. Addison Wesley, 8th edition, 2006.