M255 Unit 13
UNDERGRADUATE COMPUTING

Object-oriented programming with Java

Software testing

Unit 13

This publication forms part of an Open University course M255 *Object-oriented programming with Java*. Details of this and other Open University courses can be obtained from the Student Registration and Enquiry Service, The Open University, PO Box 197, Milton Keynes, MK7 6BJ, United Kingdom: tel. +44 (0)870 333 4340, email general-enquiries@open.ac.uk

Alternatively, you may visit the Open University website at http://www.open.ac.uk where you can learn more about the wide range of courses and packs offered at all levels by The Open University.

To purchase a selection of Open University course materials visit http://www.ouw.co.uk, or contact Open University Worldwide, Michael Young Building, Walton Hall, Milton Keynes, MK7 6AA, United Kingdom for a brochure: tel. +44 (0)1908 858785; fax +44 (0)1908 858787; email ouwenq@open.ac.uk

The Open University
Walton Hall
Milton Keynes
MK7 6AA

First published 2006. Second edition 2008.

Edited and designed by The Open University.

Typeset by The Open University.

Printed and bound in the United Kingdom by The Charlesworth Group, Wakefield.

ISBN 978 0 7492 6794 0

2.1

CONTENTS

Introduction 5

1 Testing: its purpose and background 6

- 1.1 Why do we test software? 6
- 1.2 Black box testing and white box testing 6
- 1.3 The input–output model of testing 12
- 1.4 Kinds of testing 13
- 1.5 Test-driven development 14
- 1.6 Running tests 14

2 Introducing JUnit 16

- 2.1 Assertions 16
- 2.2 Enabling JUnit 17
- 2.3 Hands-on with JUnit 18
- 2.4 A method that fails its test 28
- 2.5 Test fixtures 30
- 2.6 Hand coding JUnit tests 34

3 Unit testing the class `CompuFrog` 38

- 3.1 Refactoring the class `CompuFrog` 44

4 More about designing test data 47

- 4.1 Boundary and computation errors 47
- 4.2 Using boundary values and equivalence classes 50
- 4.3 When inputs are not ordered 51

5 Case study: congestion charging 53

- 5.1 Congestion charging for frogs 53
- 5.2 Frogs are already `Observable` 56
- 5.3 Testing the classes `ChargeableFrog`, `CZWarden` and `Account` 57

6 Summary 60

Glossary 63

Reference 65

Index 66

M255 COURSE TEAM

Affiliated to The Open University unless otherwise stated.

Rob Griffiths, Course Chair, Author and Academic Editor

Lindsey Court, Author

Marion Edwards, Author and Software Developer

Philip Gray, External Assessor, University of Glasgow

Simon Holland, Author

Mike Innes, Course Manager

Robin Laney, Author

Sarah Mattingly, Critical Reader

Percy Mett, Academic Editor

Barbara Segal, Author

Rita Tingle, Author

Richard Walker, Author and Critical Reader

Robin Walker, Critical Reader

Julia White, Course Manager

Ian Blackham, Editor

Phillip Howe, Compositor

John O'Dwyer, Media Project Manager

Andy Seddon, Media Project Manager

Andrew Whitehead, Graphic Artist

Thanks are due to the Desktop Publishing Unit, Faculty of Mathematics and Computing.

Introduction

Software should be fit for its purpose and should produce the results it is meant to. If it does not do what it promises, it will not be much use to its users.

Unfortunately, when people write software, it contains errors. Realistically this is unavoidable; software development is an exceedingly complex activity and as human beings we are fallible. Testing is a way of finding as many errors as possible, so that we can put them right. To test a program, it is run with suitable input data to see whether the results are what they should be according to the program specification.

If we could test software with every possible combination of inputs, we could be sure that we had caught all the errors and, when they had been fixed, that the software would be completely correct, but this is seldom if ever achievable. Even in simple cases the number of possible inputs can be enormous. Imagine a program that did no more than multiply two integers. As the values of type `int` range from –2 147 483 648 to 2 147 483 647 in Java, there would be 18 446 744 073 709 551 616 (or approximately 1.8×10^{19}) possible combinations of two integers to test!

So in practice testing involves choosing a small sample from among the immense range of possible inputs, and it is necessary to try very hard to find the right ones – those that have a very high probability of exposing a flaw in the code. A good test is one that throws up a fault that other tests have not found! We can never eliminate all the faults from software, but we can design and execute tests that detect most of them. If there are relatively few remaining faults, and they are of low severity, the software will be acceptable to users.

After a short initial discussion, the unit introduces an example of testing to give a feel for the sorts of question we as testers need to ask. It then goes on to survey some different kinds of testing, in terms of where they fit into the process of software development and what the focus of the testing is. An important idea is **unit testing** – verifying that an individual method functions according to specification. By building up unit tests progressively we arrive at a set of tests for an entire program.

One particular philosophy we discuss is test-first; this is the idea that we should develop software in small stages, testing as we go, and that we should write unit tests for each step first, and only then write the code that will pass the tests.

When we do unit testing, as we develop the classes and methods in a typical application, the number of individual tests we need to run will soon become very large, and setting them up and managing them will be a big and complex task. To support this work, there is a software framework, called JUnit that automates much of the process. JUnit grew from an earlier testing framework for Smalltalk and has become the effective standard for unit testing. Section 2 shows in detail how to set up and run tests in JUnit, and then in Section 3 this knowledge is put into practice.

Section 4 discusses the key question of how to pick a good set of test data, and explains a systematic way of choosing tests.

Finally, in Section 5 all the ideas presented in the unit are brought together and applied to testing a small application which uses an important idea from software design – the Observer pattern. Observer is just one of many software patterns, and this is a first introduction to an interesting topic, which you will meet again if you continue to study computing – as we very much hope you will!

1 Testing: its purpose and background

1.1 Why do we test software?

The commonsense answer is that we test software because we want to check that it does what it is supposed to. We would like know the code is correct. Unfortunately this is not possible!

'Program testing can be used to show the presence of bugs, but never to show their absence!' (Dijkstra, 1972)

We can never prove that software is correct by testing, because we would have to try out the software with every possible combination of inputs and in every conceivable set of circumstances, and to do this with even a small program would require a prohibitively large number of tests. All we can do is try to uncover defects – bugs in other words. If a test produces unexpected results, a fault has been identified. We fix the fault and run the test again; repeating the cycle of test and fix until the software passes all the tests and no more defects are revealed.

If the program is a small one, this strategy may eventually eliminate all the faults, but for medium or large programs that is most unlikely. There is a limit to the time and resources available for testing and moreover some bugs will probably remain undiscovered however much effort we expend. But if our testing is properly designed we will be able locate *most* of them and reduce the level of defects to an acceptable level.

Since we are trying to detect errors, the best kind of test is one that is likely to reveal flaws; we want tests that have a high probability of making something go wrong. Tests that we thought all along would yield the right results will not be very useful, because they add no new information and merely confirm that the software is working as expected.

If we have written the program ourselves we may be tempted to test it in a fairly casual manner, with a limited range of inputs – a bit like designing and building a new aircraft and then testing it with one quick spin round an aerodrome. To avoid this, we need to draw up and carry out a test plan which has been carefully designed to provide adequate coverage and to look actively for as many defects as possible.

Debugging is a whole topic in itself, although you saw some useful debugging techniques in *Unit 8*.

It is important to distinguish **testing** from **debugging**. Testing is the process of uncovering faults in the software. Debugging is the process of identifying the cause of the faults and correcting them. In this unit we are concerned mainly with testing; any faults we discover will be fairly easy to locate just by inspecting the code.

1.2 Black box testing and white box testing

There are two fundamental ways of testing. The first is **black box testing**. This expression probably comes originally from testing of electronic equipment where internal circuitry is completely hidden inside a box. The tester knows what the box is meant to do – there is a written specification – and there are a number of buttons and knobs that can be pressed and turned and the effect observed, e.g. bulbs lighting up. By carrying out a series of experiments the tester hopes to discover if the box is performing to specification.

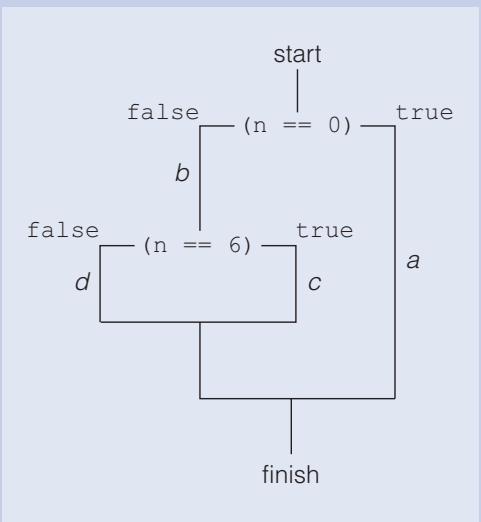
The second approach is **white box testing** (sometimes called glass box or clear box testing). In white box testing the tester knows about the internal circuitry and uses this knowledge to design tests. Now the focus is not on testing against specification, but on making sure that every part of the circuitry has been tested.

In software terms black box testing means we test against a specification – looking at function – without looking at the code at all. White box testing means we examine the code in detail – looking at program structure – and try to make sure that every statement has been tested in action. Of course in both forms of testing we decide if the results the program gives are correct by reference to the specification; we cannot reliably deduce what is 'right' just from looking at the code.

An example of white box testing

We want every statement in the following method to be executed at least once.

```
public String sampleMethod(int n)
{
   if (n == 0)
   {
      return "Sunday"; // a
   }
   else
   {
      // b
      if (n == 6)
      {
         return "Saturday"; // c
      }
      else
      {
         return "Weekday"; // d
      }
   }
}
```



To design the white box tests we look at the control structure of the program. There are two branch points, one at each `if` statement.

If the first condition evaluates to `true`, path *a* is taken.

Otherwise path *b* is taken and the second condition reached. If this evaluates to `true` path *c* is taken, otherwise path *d* is taken.

The three routes *a*, *bc* and *bd* each contain a program statement not found in the others. So to execute every program statement three white box tests will be needed, as follows.

1 With `n` equal to `0`, so *a* is taken.

2 With `n` equal to `6`, so *bc* is taken.

3 With `n` not equal to either `0` or `6`, so *bd* is taken.

White and black box testing are often used in conjunction, because they complement one another. White box testing may fail to cover the full specification but will ensure that all lines of code are executed at least once, while black box guarantees that the specification has been tested fully but may leave parts of the code untested.

However, for most purposes, it is usually adequate for black box testing to be used on its own. It is also less complicated to apply, and more in keeping with the object-oriented idea that the implementation of a method should be hidden from clients that make use of it. In M255 we will use only black box tests.

The following exercise asks you to think about how you would black box test a method from a new amphibian class.

Exercise 1

Recently it has been discovered that some amphibians are gifted at arithmetic. A compufrog is a kind of frog that can add together two numbers and tell us the answer by moving to the corresponding stone. (In case the graphical display is not open, it also returns the result of the addition as a message answer.)

So far compufrogs have learnt to add only single-digit numbers. To be nice to them we have agreed that we will not ask them to do any sums with numbers outside the range 0 to 9, inclusive. This will make testing slightly simpler.

The class `CompuFrog`, which simulates compufrogs, has a method `add(int x, int y)` which adds two numbers and returns the result. Here is the comment for the method.

```
/**
 * The method takes two integer arguments x and y. If these
 * arguments are one-digit positive integers (ie integers
 * between 0 and 9 inclusive) the method sums the two arguments
 * and sets the position of the receiver to that sum and then
 * returns the sum. Otherwise the method simply returns -1.
 */
```

In this exercise we want you, on paper, to draw up a black box test plan for the method `add()`. This will involve deciding on what particular combinations of integer values you need to give as arguments to the message `add()`, and for each message what the expected result should be. We suggest you spend about 15 minutes on this task.

Although `add()` performs only a fairly simple action, devising a test plan for the method will take quite a bit of thought. To help you formulate a plan it is useful to begin with a number of questions.

1 What are you trying to achieve?

   The aim is to design tests which detect as many errors as possible, as you saw in Subsection 1.1.

2 How do you know what the result of a particular test should be?

   You decide what the results should be by referring to the specification given in the method comment.

3 How do you know the testing is adequate and how do you choose the tests?

In practice one can hardly ever get complete coverage, because typically the number of possible combinations of inputs is too large (and may even be infinite); for instance, a program that interacts with a user – a word processor, say – has an open-ended set of inputs. In principle the user can go on entering keyboard characters indefinitely, so we can never cover all the possible cases. In the case of the `add()` method, if we wanted to test it with all possible combinations of integer values as arguments, as noted in the Introduction, there would be 18 446 744 073 709 551 616 possible combinations of two integers to test!

The art of black box testing is to choose a good sample, and what makes a sample good comes back to what testing is trying to achieve. Remember *we are looking for defects.* So any test likely to uncover a fault that other tests cannot detect is a good one to include in the sample.

Conversely if a test is only likely to detect a fault that would be found by some other test, there is no point it including it, since it will not tell us anything new.

Solution

The tests we came up with for the method `add()` are shown in Table 1. The test plan is based on the comment of the method `add()`.

The method should only sum the arguments if they are both between 0 and 9 inclusive, otherwise it should return –1. When a method has to deal in some special way with a range of numbers – in this example 0 to 9 – it is easy for a programmer to make a mistake at the ends of the range. For example, they might use a condition such as `(x > 0)` when it should be `(x >= 0)` and this will give wrong results at 0. (It might make compufrogs think they cannot add 0 to anything!) So you will notice that we have specified several tests with one or both of the arguments equal to or near to 0 and also equal to or near to 9.

Boundary testing is explored in more detail in Section 4.

We also need to test using values around the middle of the range. There is no point in trying lots of these. If the results are correct when we test using 5 plus 7, then we might reasonably expect that, say, 3 plus 6 will also produce a correct addition. In principle it might not – for all we know there may be some very strange effect that means `add()` functions perfectly for all pairs of inputs except 3 plus 6 – but this is unlikely.

The probability is that the code will treat 3 plus 6 in the same way as 5 plus 7, and so either both tests will give correct results or both will give incorrect results. We would expect that if the first test detects a flaw, the second test will only detect the same flaw and reveal nothing new. So the second test is redundant.

We say the two tests are *equivalent*. This is a notion we will explore further in Section 4.

You may have thought of other tests; for example, trying arguments which are not integers, e.g. 2.54 and 3.98, but Java will not let us send such messages. The signature of the method `add(int, int)` dictates that the arguments can only be a pair of integers. However, if we were using another programming language, these tests might well be required.

Table 1 Test cases for the method `add()`

Test case	1st argument	2nd argument	Expected result	Rationale
1	0	0	0	Both numbers on lower boundary
2	–1	–1	–1	Both numbers outside lower boundary
3	0	1	1	First number on lower boundary, second close but inside lower boundary
4	0	–1	–1	First number on lower boundary, second close but outside lower boundary
5	1	0	1	First number close but inside lower boundary, second number on lower boundary
6	–1	0	–1	First number close but outside lower boundary, second number on lower boundary
7	1	–1	–1	First number close but inside lower boundary, second number close but outside lower boundary
8	–1	1	–1	First number close but outside lower boundary, second number close but inside lower boundary
9	1	1	2	Both numbers near but inside lower boundary
10	5	7	12	Typical values near middle of range
11	8	8	16	Both numbers near but inside upper boundary
12	8	9	17	First number close but inside upper boundary, second number on upper boundary
13	10	9	–1	First number close but outside upper boundary, second number on upper boundary
14	9	8	17	First number on upper boundary, second close but inside upper boundary
15	9	10	–1	First number on upper boundary, second close but outside upper boundary
16	9	9	18	Both numbers on upper boundary
17	10	10	–1	Both numbers outside upper boundary
18	8	10	–1	First number close but inside upper boundary, second number close but outside upper boundary
19	10	8	–1	First number close but outside upper boundary, second number close but inside upper boundary

ACTIVITY 1

In this activity you will test the method `add()`. Launch the project Unit13_Project_1.

In the OUWorkspace create an instance of `CompuFrog` and send it the message `add()` with the pairs of arguments shown in Table 1 above. Note the results for each test case and state whether the method has passed or failed each test. This is a black box test, so make sure you do not look at how the class `CompuFrog` is coded!

DISCUSSION OF ACTIVITY 1

You should have obtained the results shown in Table 2.

Table 2 Test results for the method `add()`

Test case	1st argument	2nd argument	Expected result	Actual result	Pass/Fail
1	0	0	0	0	Pass
2	–1	–1	–1	–1	Pass
3	0	1	1	1	Pass
4	0	–1	–1	–1	Pass
5	1	0	1	1	Pass
6	–1	0	–1	–1	Pass
7	1	–1	–1	–1	Pass
8	–1	1	–1	–1	Pass
9	1	1	2	2	Pass
10	5	7	12	12	Pass
11	8	8	16	16	Pass
12	8	9	17	–1	Fail
13	10	9	–1	–1	Pass
14	9	8	17	17	Pass
15	9	10	–1	–1	Pass
16	9	9	18	–1	Fail
17	10	10	–1	–1	Pass
18	8	10	–1	–1	Pass
19	10	8	–1	–1	Pass

The method has failed tests 12 and 16 (but notice it passed 14, so tests 12 and 14 are not equivalent). As you will have guessed, we have deliberately introduced a fault; in fact we have used the wrong comparison operator and so introduced precisely the kind of error that the boundary tests were designed to detect.

For the moment we will leave the bug uncorrected, because we want to use the faulty version in further activities.

1.3 The input–output model of testing

Black box testing, not just of software but of other products, follows an input–output model. We provide the product with some input and check if what comes out the other end, the output, is what it should be according to a specification.

For example, to test an electric kettle, we would fill it with cold water and switch on. The inputs are cold water and electricity and the output should be hot water. If it is not, the kettle has failed the test!

Note that since the behaviour of objects will often be state dependent it is essential to take state into account when devising tests.

To carry out a software test, we create all the objects involved in the code we want to test and set up their initial states. We provide the required message arguments and any form of run-time input that may be required: for example, from a dialogue box or the keyboard. These inputs constitute the **test data**.

Then we execute the code we are testing, and inspect the results, which will be the new state of the objects plus any message answers and output the code has generated.

For any set of test data it is possible to predict what the results should be, given the specification to which the code was written.

We compare these **expected results** with the **actual results** – those produced when we executed the code. If the actual results match the expected results, the code has *passed* the test. Otherwise it has *failed* the test (Figure 1).

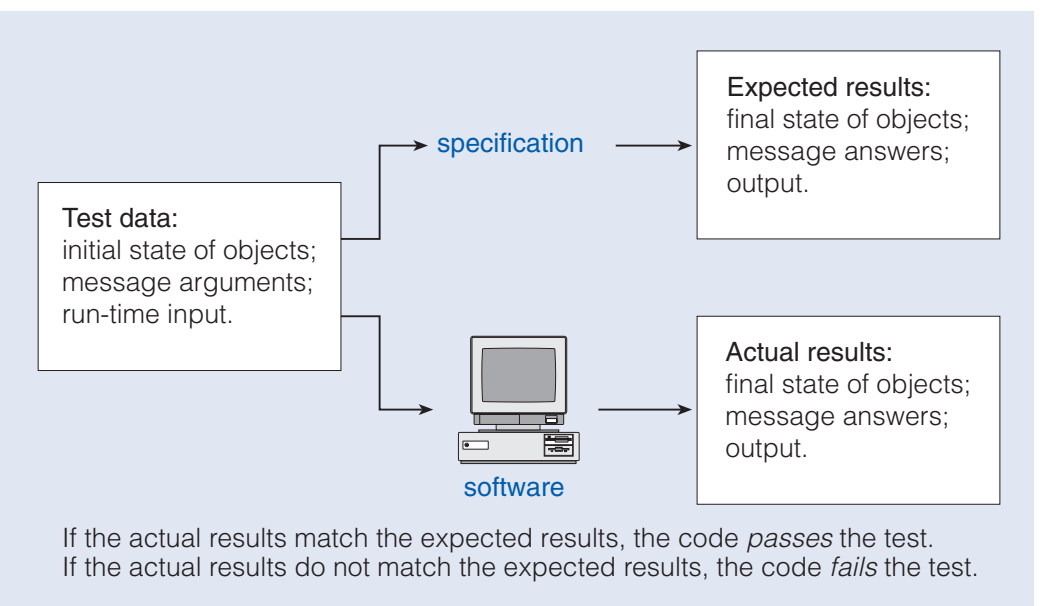


Figure 1 The input–output model of software testing

Exercise 2

Following the input–output model above, outline on paper how you would set about testing the following instance methods of the `Frog` class. Assume you are using the OUWorkspace.

(a) `setColour()`

(b) `getColour()`

(c) `sameColourAs()`

Solution

There are many correct answers. This is what we did.

(a) Begin by creating an instance of `Frog` referenced by `testFrog1`. Inspect its state; `position` should be set to `1` and `colour` should be set to `OUColour.GREEN`.

Send `testFrog1` the message `setColour(OUColour.RED)`. Inspect its state; `colour` should now be `OUColour.RED` and `position` should still be `1` confirming that `setColour()` has only changed the value of the instance variable `colour`.

(b) Send `testFrog1` the message `getColour()`. The message answer should be `OUColour.RED`. Inspect the state of `testFrog1` once more to confirm that `getColour()` has not changed the values of the instance variables.

Send `testFrog1` the message `setColour(OUColour.YELLOW)` and then immediately send it the message `getColour()`. This time `getColour()` should return `OUColour.YELLOW`.

(c) Create a second instance of `Frog` and assign it to a variable called `testFrog2`.

Inspect the state of `testFrog2`, to confirm that `position` is set to `1` and `colour` is set to `OUColour.GREEN`.

Inspect the state of `testFrog1`, to confirm that its `position` is set to `1` and its `colour` is set to `OUColour.YELLOW`.

Send `testFrog2` the message `sameColourAs(testFrog1)`. Then send it the message `getColour()`. The message answer should be `OUColour.YELLOW`. Finally inspect the state of `testFrog2` to check that only its colour has been changed, and then inspect the state of `testFrog1` to confirm that its state is unaltered.

1.4 Kinds of testing

All the testing we will be looking at falls under the umbrella of **alpha testing** – testing which is done in-house before a product is available publicly. You may also have heard of **beta testing**. This is when a trial version of a product is made available to potential users, on the understanding that this is not the final form, so they can try it out if they are interested and report any problems they find.

Commercial software developers often employ specialist testers who are distinct from the programmers who write the code. The advantage of this is that the testers, not having produced the software themselves, tend to be less protective of it and do not make the same assumptions as the programmers. Hence they are more likely to discover its defects. However, it is also common for code to be tested by its authors, or for responsibility to be split, with testers testing the overall program, while programmers test the individual classes they write.

For the purpose of this unit we assume that code is tested and retested by the programmers who wrote it, with the first tests being done as soon as any code has been written and that testing is a continual activity which goes on in parallel with the development of the software. This contrasts with the alternative and more traditional approach in which testing does not begin until all or a large part of the coding is complete.

There are several different levels of testing.

Unit testing (sometimes called component testing) is the smallest test possible. In object-oriented programming a unit test is defined as either a test of a single method, or a test of a single class. We will use the first definition, so our unit tests are tests of single methods.

You can think of a unit test as an 'atom' of testing, with larger-scale tests being built up from atoms.

Integration testing is testing in which software components (groups of classes) or hardware components or both (all having been tested previously in isolation) are combined and tested together to evaluate the interaction between them.

Regression testing checks that modifications or additions to one part of the code have not made any other part stop working properly.

System testing checks that the overall system functions correctly according to its specification.

Acceptance testing checks that the system meets the needs of a customer who has commissioned the software and provided a specification of their requirements.

We shall consider unit testing in detail and we shall also look at regression testing, although more briefly. Integration, system and acceptance testing are beyond our scope and will not be discussed.

1.5 Test-driven development

Traditionally programmers wrote code and then they or someone else tested what they had done. This is a *test-last* approach.

TDD also lays stress on refactoring the code whenever doing so can improve the design. Refactoring was discussed in *Units 6* and *7*.

In **test-driven development (TDD)** code is produced in small steps called **increments**, each of which involves just a few units. All tests are written *before* the corresponding code, so this is a **test-first** approach. Once the code is written it is tested, any bugs are fixed and once the code passes the tests we move on to write the tests for the next increment.

Each time we change the code we do regression testing, which simply consists of rerunning all the earlier unit tests, in case some other part of the code previously written and tested has now stopped working because of the changes.

Because the software is built up in small increments, with the system (built so far) regression tested at the end of every increment, this approach offers considerable assurance that the software will be trustworthy – although we should not assume it will be bug free: TDD is only as good as the design of the tests.

TDD can be traced as far back as NASA's Project Mercury in the 1960s, but it has recently become much more popular, as part of the eXtreme Programming (XP) approach to software development. Some studies have indicated that software developed under TDD may have up to 50 per cent fewer defects than the equivalent software produced using traditional methods.

In M255 we do not use the full TDD approach, but, as you will see, the unit-testing methods we introduce in Section 2 are very heavily influenced by the test-first philosophy.

1.6 Running tests

Once you have designed a set of tests you need a way of running them. In Activity 1 you did this in the workspace, creating an instance of `CompuFrog`, sending it a series of messages, and observing the results.

If we had no workspace – which is the case in most Java development environments – we would have to write a special program to create the instance of `CompuFrog`, send the messages, and output the results using `println` statements. This program would be an example of what is called a **test harness**.

This is fine for a one-off test but hopeless for developing a whole system using test-driven development. Each new unit test would have to be put together by hand, and to do the regression testing we would need a way to manage and reload all the previous unit tests. Since large applications might easily build up to hundreds or thousands (or more) unit tests, this is impossible without computer assistance.

Fortunately there is software to automate the entire process. The **testing framework** JUnit provides all the facilities required to create and manage unit tests. JUnit was originally introduced by Kent Beck and Erich Gamma and derived from Kent Beck's earlier Smalltalk unit-testing framework, SUnit.

Since its inception JUnit has been immensely popular and it is now integrated into most Java IDEs. The next section explains how to use JUnit in BlueJ.

2 Introducing JUnit

JUnit provides a framework which makes it easy to write tests using a standard format, and to reuse tests already written. JUnit tests consist of a series of test methods, which are written in special test classes. Executing the test methods is an automated process; we do not have to write any kind of harness.

Each time the tests are run, JUnit generates a report listing which lists the tests the software passed, and which it failed.

Of course, if the developers are using TDD, each set of tests will be written before the corresponding code!

Working with JUnit, developers write a small chunk of code, then test it, write some more code and test it, and so on, building up a collection of test methods. Each time a new class is added, or an existing class is refactored in any way, the entire set of tests is rerun, so regression testing is done every time the software is changed. If the software fails one or more tests, the developers must fix the fault before they write any more code.

The main advantages of this approach are:

(a) it is easy for developers to write tests, so code is likely to be tested more thoroughly;

(b) errors will be detected early (when they are easier to correct), instead of near the end of the project (when they will much harder to rectify);

(c) regression testing is built in and requires no additional effort;

(d) at each stage there is confidence that the software developed to date has been properly tested.

2.1 Assertions

JUnit has purposely been designed so we can ignore most of what happens behind the scenes. All the background information you need is given below.

Most IDEs that incorporate JUnit provide a way of creating test classes automatically, and of adding incomplete test methods to them. The tests provided are incomplete because JUnit cannot decide what a test should actually be checking or what steps are needed to do it. Only we can determine that, using the specification we have been given for the code.

A test method is built around the input–output model of Subsection 1.3. It begins with an object or objects in given states, executes the code to be tested, and then decides if the results are as expected. If so, the code passes the test, otherwise it fails.

To decide whether the code has passed, JUnit evaluates an **assertion** – a special statement which says what should be true if the code is correct.

As we have noted above, JUnit does not know what should be true, only we do, so only we can write the assertions.

In most cases the assertion simply compares an expect result with the actual one. For example,

```
assertEqual(13, baker.getDozen());
```

Note that the form of this assertion is

```
assertEqual(<expectedValue>, <actualValue>);
```

There are other types of assertion but `assertEqual` is the only one we use in M255.

Once we have written the test methods, we can get the IDE to execute all the test methods for us as a group. Whenever an assertion is not met, the test concerned has failed. At the end, JUnit displays a report of which tests have been passed and which have not.

We can summarise all this as follows. When creating tests using JUnit we:

- use the facilities of the IDE to set up test classes containing outline test methods;
- complete the body of the methods, deciding what the assertions should be;
- use the facilities of the IDE to run the tests and generate a report.

2.2 Enabling JUnit

To use JUnit in BlueJ we must first enable it, since BlueJ's default is that JUnit is disabled.

To enable JUnit choose Preferences from BlueJ's Tools menu and then select the Miscellaneous tab, as shown in Figure 2.

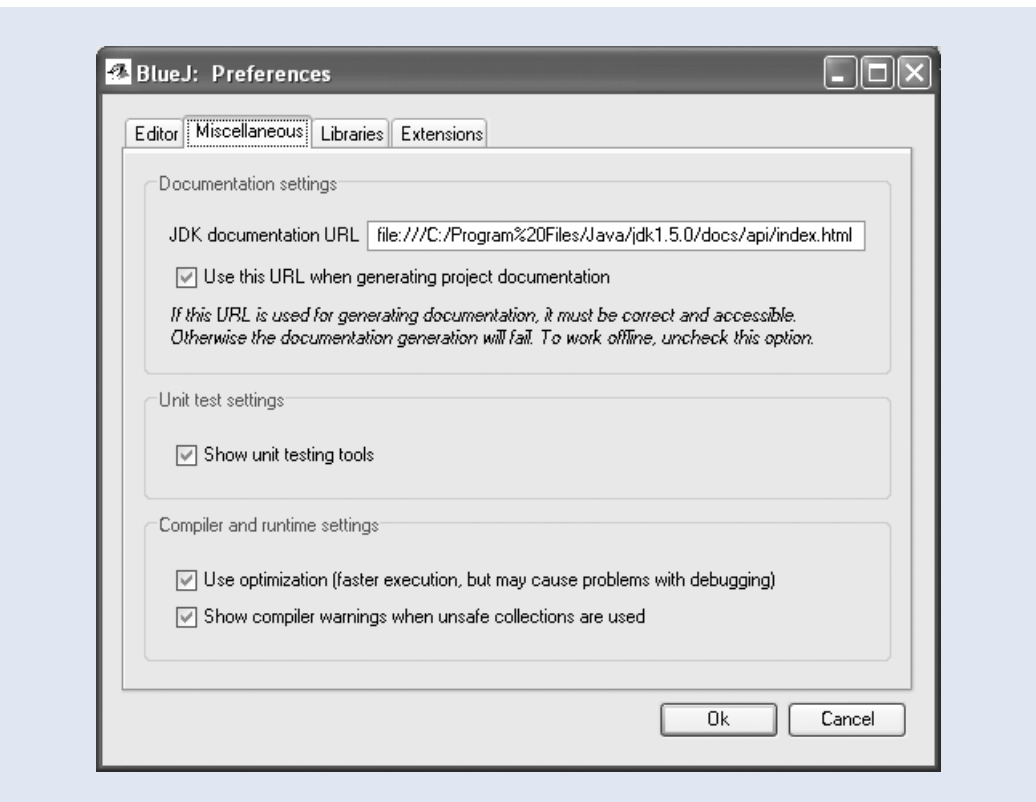


Figure 2 The Preferences window

Ticking the box labelled 'Show unit testing tools' will enable JUnit. Once the Ok button is clicked you will find that some extra buttons have been added to the left-hand pane of the BlueJ window, as shown in Figure 3, which shows a window for Unit13_Project_2.

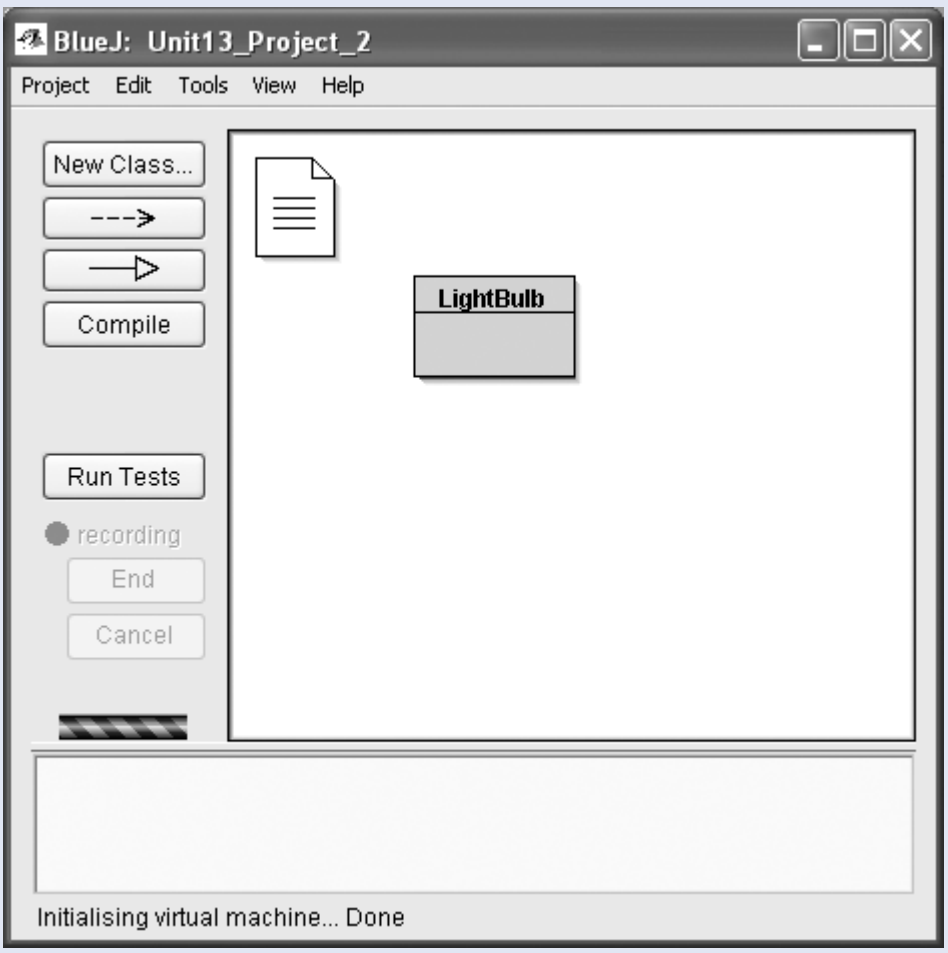


Figure 3 BlueJ window with JUnit enabled

2.3 Hands-on with JUnit

In the activities that follow we demonstrate how JUnit is used in BlueJ, using an application involving two classes `LightBulb` and `Switch` as an example. Activity 2 is much longer than the others, because it establishes the groundwork. There is no discussion of Activity 2, and you will find that not all activities in this unit have discussions.

ACTIVITY 2

1 Open Unit13_Project_2 which contains the single class `LightBulb`; instances of this class model a light bulb and have a single instance variable `status` that can be set to one of two values: `"On"` or `"Off"`. Double-click on the class icon to open the editor and look at the code and method comments. The class contains one instance variable named `status`, a constructor that initialises `status` to `"Off"` and three instance methods: the comment for `on()` specifies that it sets the `status` of the receiver to `"On"`; the comment for `off()` specifies that it sets the `status` of the receiver to `"Off"`; and the comment for `getStatus()` specifies that it returns the current value of `status`.

2 Ensure that JUnit is enabled (described in Subsection 2.2).

3 You will now create a test class for the class `LightBulb`.

Right-click on the `LightBulb` icon and choose Create Test Class from the menu (Figure 4).

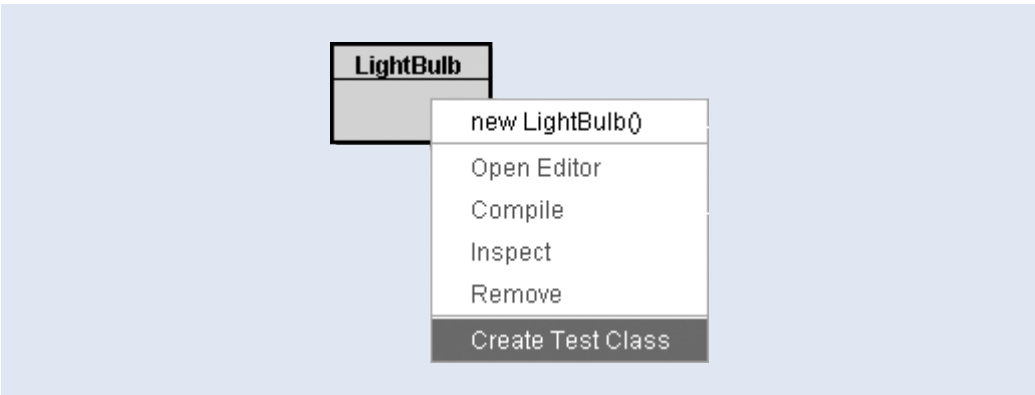


Figure 4 Creating a test class for `LightBulb`

A new green box representing a test class appears behind and slightly to the upper right of `LightBulb` icon (Figure 5). This new 'shadow' class will contain the tests for the class `LightBulb`. If you drag the `LightBulb` icon around the window, you will see that the shadow class follows class `LightBulb` to its new position.

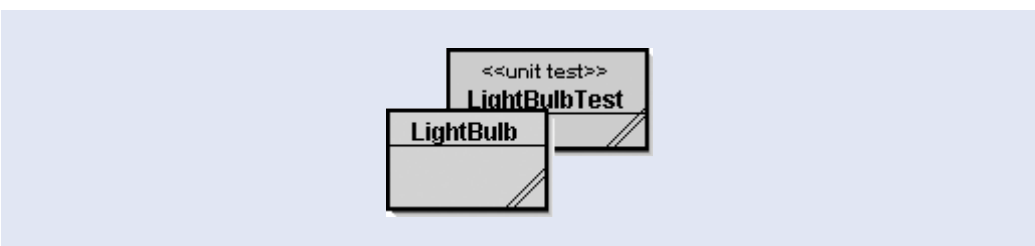


Figure 5 The `LightBulb` class shadowed by the `LightBulbTest` class

Double-click on the `LightBulbTest` icon to open the editor. Note that the class simply contains the default constructor and the method `setUp()` which has an empty code block. Close the editor window.

4 Create a test method.

Currently `LightBulbTest` does not do anything useful, because it contains no useful test methods. The next step is to add a test method which tests the method `on()` of the `LightBulb` class. To help us do this, BlueJ has a useful JUnit recording facility which lets us carry out a series of operations, recording them as we go, and then play them back as a test method, so we will use this recording feature to create a test method for the method `on()`.

Although recording is not generally available in JUnit, it has been added as an extra feature in the BlueJ implementation.

Right-click on the `LightBulbTest` icon and choose Create Test Method... from the menu (Figure 6).

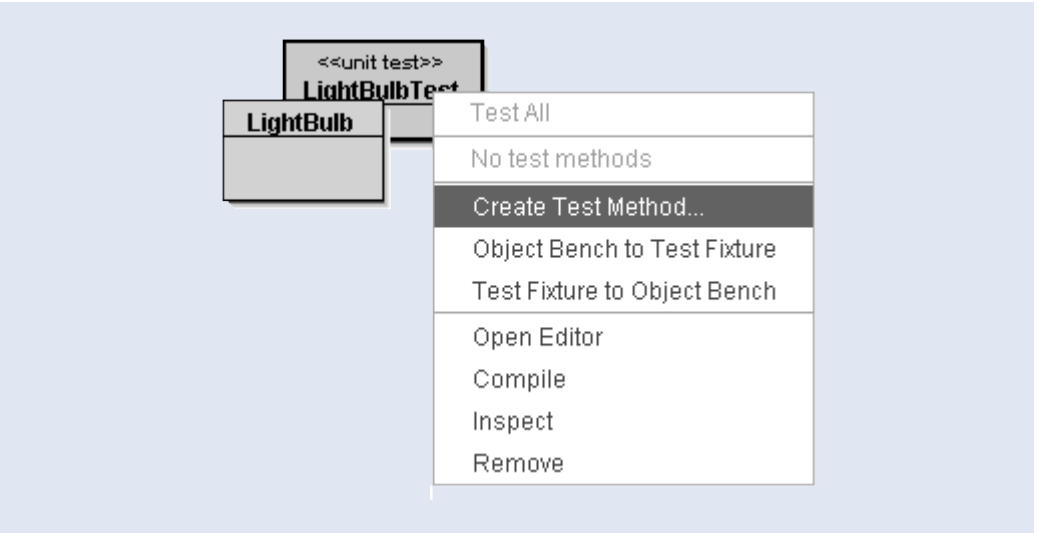


Figure 6 Creating a test method

A New Test Method dialogue box will appear. Give the new test method a sensible name, such as `testOn` (note the lack of parentheses after the method name). Notice that the dialogue box tells you that JUnit will begin recording.

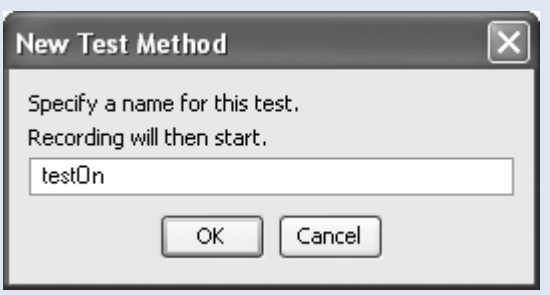


Figure 7 The New Test Method window

Click the OK button to continue.

The New Test Method dialogue box closes, returning you to the main BlueJ window. A red dot and the word recording should now have appeared in the left-hand pane of the BlueJ window. Underneath this is a button labelled End for when you want to finish recording, and a button labelled Cancel in case you want to cancel the recording. A message 'recording LightBulbTest.testOn()' is displayed at the bottom left of the BlueJ window, as shown in Figure 8.

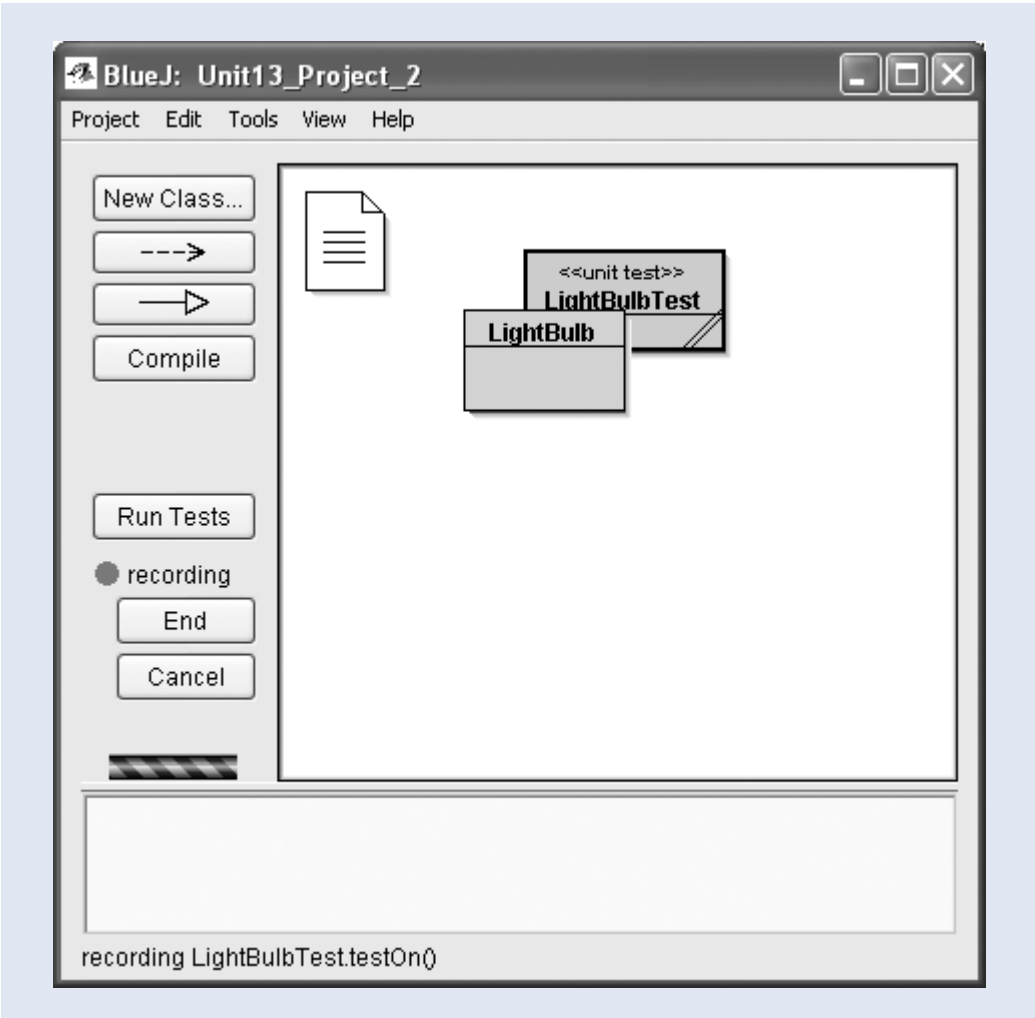


Figure 8 Ready to record a test method

5 Create a test object.

Before you can test the method `on()` of `LightBulb`, you need to create a `LightBulb` object to which you can send the corresponding message. BlueJ allows us to create special test objects for this purpose. Select the `LightBulb` icon (not the shadow class behind it) and right-click to bring up the menu (Figure 9).

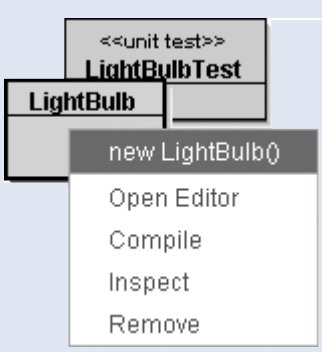


Figure 9 Creating a new test `LightBulb`

Choose new LightBulb() from the menu. A dialogue box will appear, asking you to name the new object. Accept the default name of `lightBul1` and click Ok to create the `LightBulb` object. The new `LightBulb` object will appear near the bottom left of the BlueJ window in a special pane known as the **Object Bench** (Figure 10).

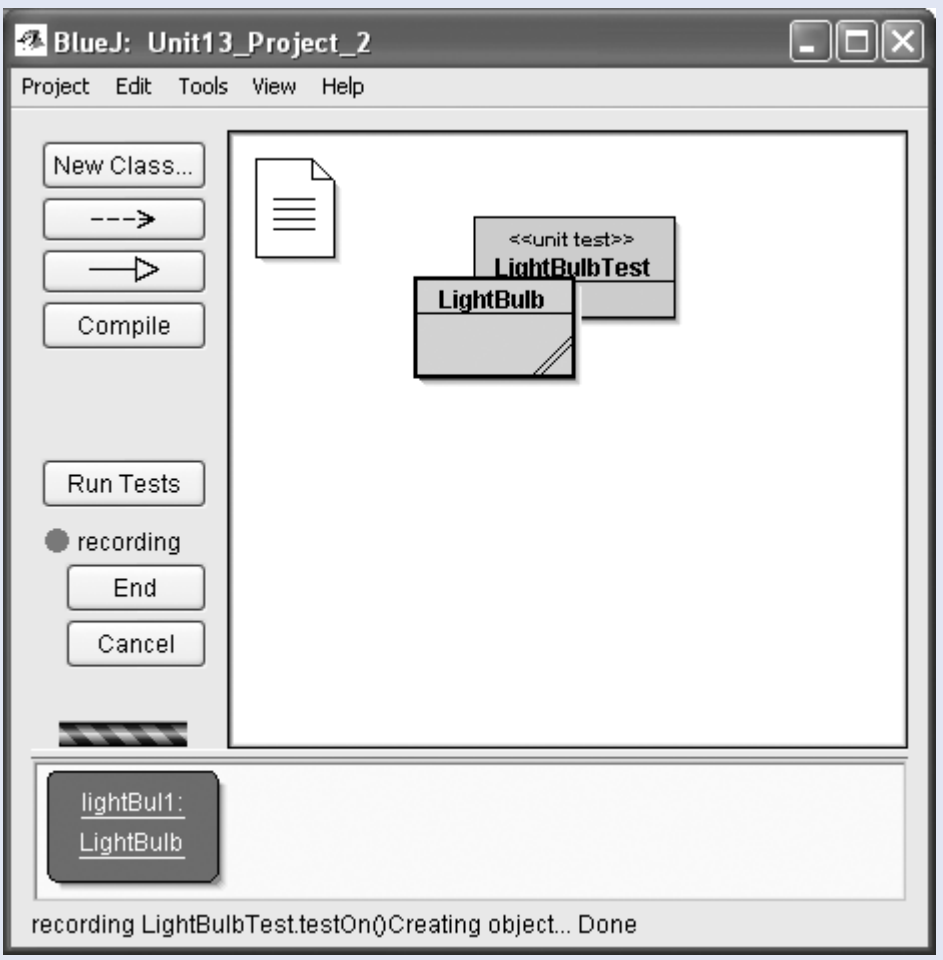


Figure 10 A test object has been added to the Object Bench

Right-clicking on the object `lightBul1` in the Object Bench brings up a menu of actions you can perform (Figure 11).

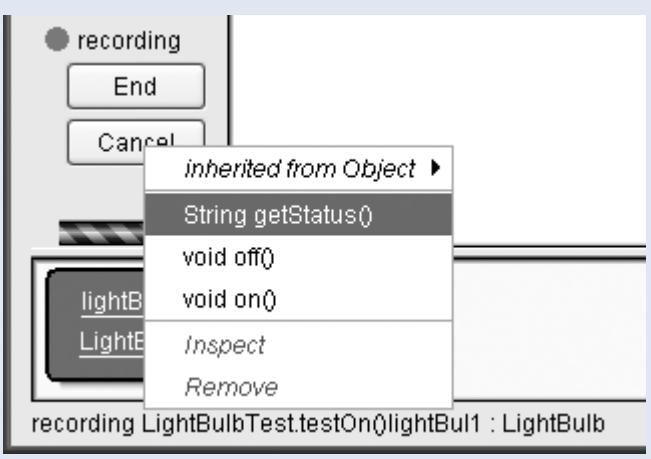


Figure 11 Operations that can be performed on the test object

The first entry, inherited from Object, brings a submenu of methods inherited from class `Object`. Next come entries for the three instance methods defined in the class `LightBulb`. The option Inspect will allow you to examine the current state of `lightBul1`, while Remove will destroy the object. We will be using only the three menu items corresponding to the instance methods in the class `LightBulb`.

6 Record a message-send.

Now that you have created a test object (and recorded this action), you need to send it some messages to test the method. As each message is sent, it will be recorded in the `LightBulbTest` method `testOn()`.

To test the method `on()` you need to record the actions shown in Table 3.

BlueJ's recording mode for JUnit is unlike that of a cassette recorder; if you do not do anything for a while there will not be an embarrassing gap in the actions when you 'playback'. BlueJ patiently waits for you to continue. So you can go at your own speed, and take a break whenever you wish.

Table 3 Message-sends needed to test `on()` in `LightBulb`

Message-send		Reason
1	`lightBul1.getStatus()`	To check the `status` of `lightBul1` before sending the message `on()`. The `status` of a newly initialised `LightBulb` instance should be `"Off"`.
2	`lightBul1.on()`	This is the method we are testing.
3	`lightBul1.getStatus()`	To check the `status` of `lightBul1` after sending the message `on()` when the value of `status` was `"Off"`. The value of `status` should now be `"On"`.
4	`lightBul1.on()`	This is the method we are testing.
5	`lightBul1.getStatus()`	To check the `status` of `lightBul1` after sending the message `on()` when the value of `status` is already `"On"`. The value of `status` should be `"On"`.

In the Object Bench, right-click on `LightBul1` and select String getStatus() from the menu (as shown in Figure 11).

A Method Result dialogue box will now open (Figure 12). Do not worry if this looks a bit complex at first. We will explain everything.

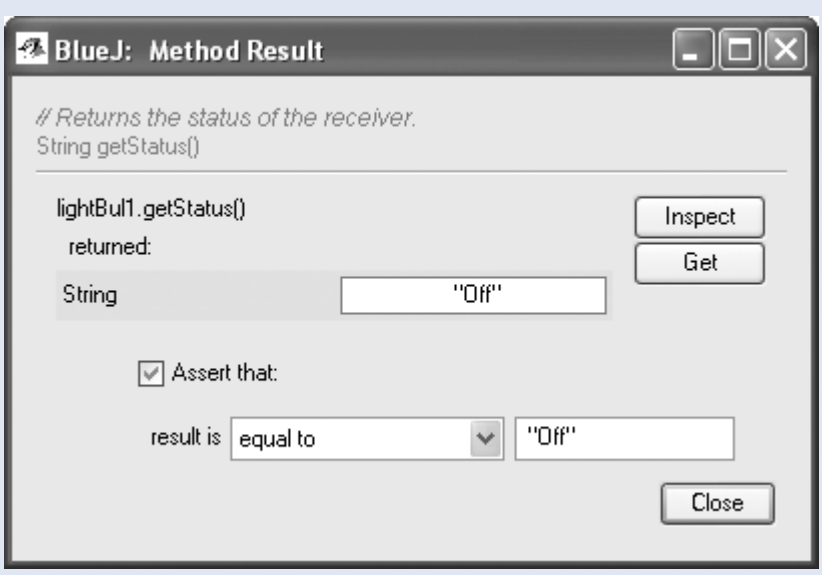


Figure 12 The Method Result window for `getStatus()`

The upper part of the dialogue box shows that the value actually returned from the message-send `lightBul1.getStatus()` was the `String` object `"Off"`.

The bottom part of the window invites us to include an assertion statement in the test method. This assertion will have the form

```
assertEqual(<expectedValue>, lightBul1.getStatus());
```

and it is up to us to say what the expected value should be. However, BlueJ offers a *suggestion*: that the expected value is the same as the actual one it knows about.

In this case, the actual result is indeed the correct one, so we can accept the suggestion and click the Close button. The test statement

```
assertEquals("Off", lightBul1.getStatus());
```

will now be inserted in the test method.

Two wrongs do not make a right!

It is vital that you never accept the assertion that JUnit suggests without making sure that the actual value and the expected one match.

If the actual result is *not* in fact the same as the expected one, something is wrong with the code. You must cancel the recording and rectify the fault before continuing.

Simply clicking Close would be disastrous, because it would build in an assertion that uses an incorrect expected value. As long as the code continued to produce the same erroneous value the assertion would evaluate to `true`. We would then have a situation in which the code was wrong but appeared correct every time we tested it, because the test was wrong as well and the two errors cancelled each other!

The fault would then be virtually undetectable and later we would be faced with a baffling situation in which the program passed all our tests but still functioned incorrectly.

So far you have only recorded the first message-send from Table 3. Now record the second, by right-clicking on the object `LightBul1` and choosing void on(). This time the Method Result window will not appear, because `on()` does not return a value.

If `on()` has worked correctly, the `status` of the `LightBulb` will have changed to `"On"`. Right-click on `LightBul1` again and choose String getStatus() to send the third message of Table 3. Since this returns a value, the Method Result window will open (Figure 13).

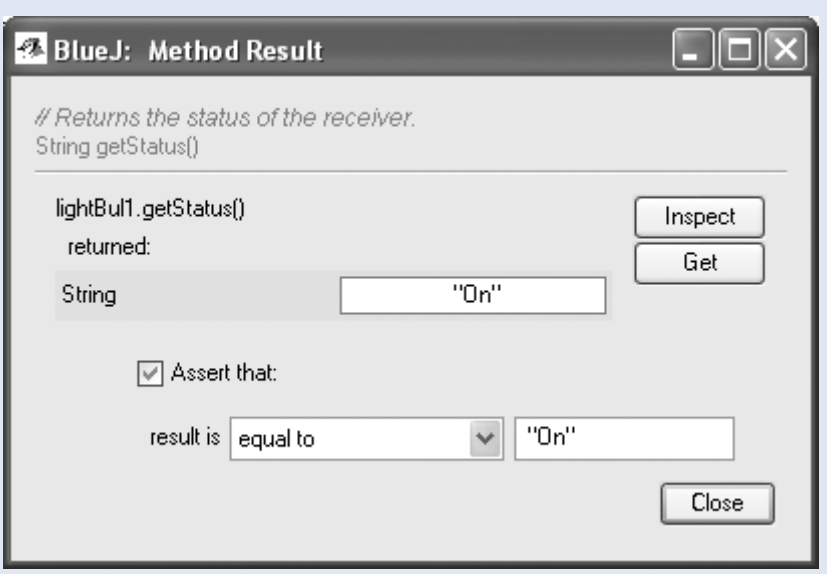


Figure 13 The Method Result window again

The actual value returned is "On", which agrees with the expected value. You can therefore click Close to accept the suggested assertion.

Now record the fourth message-send from Table 3, by right-clicking on the object LightBul1 and again choosing void on().

Finally record the fifth message-send from Table 3, by right-clicking on the object LightBul1 and again choosing String getStatus().The value returned is "On", which is what we would expect so click Close to accept the suggested assertion.

This completes the recording of the testOn() method, so click the End button in the left-hand pane of the BlueJ window. There may be a brief pause as the test class is automatically compiled. Once compilation is complete, the object LightBul1 will disappear from the Object Bench – do not worry, it has simply been incorporated into the code of the method testOn() in LightBulbTest.

7 Look at a test method.

JUnit tests are simply instance methods. Double-click on LightBulbTest to open the editor, then locate the method testOn(), which should be as follows.

```
public void testOn()
{
    LightBulb lightBul1 = new LightBulb();
    assertEquals("Off", lightBul1.getStatus());
    lightBul1.on();
    assertEquals("On", lightBul1.getStatus());
    lightBul1.on();
    assertEquals("On", lightBul1.getStatus());
}
```

When you have finished close the editor window.

8 Run your first JUnit test.

With testOn() successfully recorded, it is time to run the test. In the BlueJ window right-click on the LightBulbTest icon and select Test On to run the test.

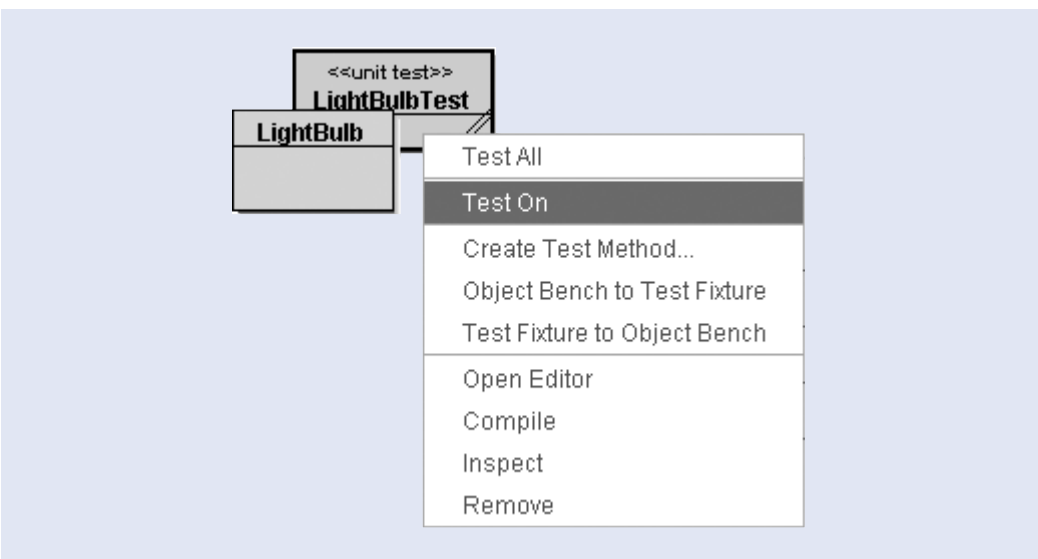


Figure 14 Selecting testOn from LightBulbTest's pop-up menu

9 View the test results.

The result of the test will appear at the bottom left of the BlueJ window. You should see testOn succeeded – good news, your class has passed this test!

For a more detailed test report, go to BlueJ's View menu and choose Show Test Results at the bottom of the menu. The Test Results window will now open.

Failures versus errors

Failures
A grey cross indicates that an assertion failed; in other words, the actual value did not match the expected one. Failures are anticipated, in the sense that we are testing for them with our assertions.

Errors
A red cross indicates that an exception (such as an `ArrayIndexOutOfBoundsException`) occurred.

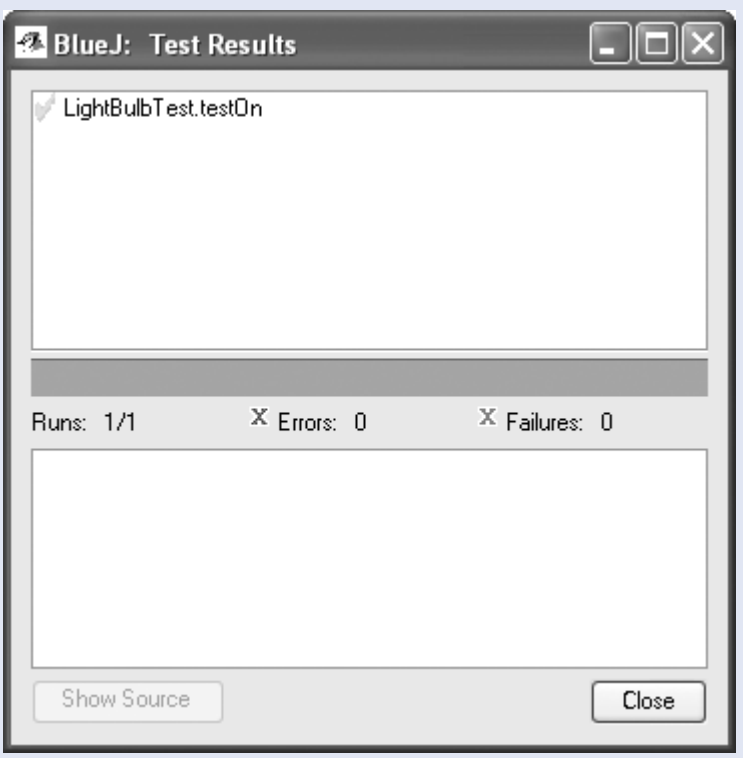


Figure 15 The `testOn()` method has passed

In the Test Results window the top pane contains the list of tests completed. Successful tests are marked with a green tick, unsuccessful tests have a small cross next to their name.

The lower pane provides a summary of the results of testing. A green bar across the middle of the window shows that the tests have been completed successfully. A red bar indicates that an error or failure occurred in the testing process. The number of runs (i.e. test methods executed), errors and failures are also reported.

If any of your tests fail you can click on the name of the test in the top pane to learn more about the failure. Information about the failure appears in the bottom pane. We will look at this in the next activity.

Click Close to exit the Test Results window.

Well done! You have successfully written and run a JUnit test for the method `on()`.

In case you had any difficulties with this activity, the class `LightBulbTest` *as developed so far has been added to Unit13_Project_3.*

The process of Activity 2 may have seemed a little long winded but that was mainly because of the explanations. Once you are familiar with creating tests it is quick and easy to do. Your next task is to create a test for `off()` and you will probably be surprised how simple this is.

SAQ 1

What sequence of messages will you need to send to test the method `off()`?

ANSWER.....

To test `off()` you need the message-sends shown in Table 4.

Table 4 Message-sends needed to test `off()` in `LightBulb`

Message-send		Reason
1	`lightBul1.getStatus()`	To check the `status` of `lightBul1` before sending the message `off()`. The `status` of a newly initialised `LightBulb` instance should be `"Off"`.
2	`lightBul1.off()`	This is the method we are testing.
3	`lightBul1.getStatus()`	To check the `status` of `lightBul1` after sending the message `off()` when the value of `status` was `"Off"`. The value of `status` should still be `"Off"`.
4	`lightBul1.on()`	The `status` of a `lightBul1` should still be `"Off"`, so we must change `status` to `"On"` for another test of `off()`.
5	`lightBul1.getStatus()`	To check the `status` of `lightBul1` is now `"On"`.
6	`lightBul1.off()`	This is the method we are testing.
7	`lightBul1.getStatus()`	To check the `status` of `lightBul1` after sending the message `off()` when the value of `status` is `"On"`. The value of `status` should now be `"Off"`.

ACTIVITY 3

Creating a test for the method `off()` will follow exactly the same steps as creating the test for `on()`, so you should refer back to Activity 2 to remind yourself what to do. Remember you will need to create a test method called `testOff()` and a test object to send messages to. Open Unit13_Project_3. Now record the test for `off()`, following the same process as for `on()` and using the message-sends listed in Table 4.

When you have finished recording the test, run it as before. Choose Show Test Results in the View menu to open the Test Results window. You should find that the method passes the test without any errors or failures.

DISCUSSION OF ACTIVITY 3

The code for the `testOff()` method should look like the following.

```
public void testOff()
{
    LightBulb lightBul1 = new LightBulb();
    assertEquals("Off", lightBul1.getStatus());
    lightBul1.off();
    assertEquals("Off", lightBul1.getStatus());
    lightBul1.on();
    assertEquals("On", lightBul1.getStatus());
    lightBul1.off();
    assertEquals("Off", lightBul1.getStatus());
}
```

In case you had any difficulties with this activity, the class `LightBulbTest` *as developed so far as been added to Unit13_Project_4.*

2.4 A method that fails its test

Now that you have successfully tested `testOn()` and `testOff()`, you have written a complete unit test for the class `LightBulb`. The next activity will extend your knowledge of working with JUnit, as we continue to use the class `LightBulb`.

ACTIVITY 4

Open Unit13_Project_5. You will recognise `LightBulb` and `LightBulbTest` from Activities 2 and 3, but we have made a slight change to the code for the method `off()` of the class `LightBulb`!

Open the class `LightBulb` and examine the code for the method `off()`. We have introduced a deliberate error. The comment for the method `off()` specifies that it should set the `status` of the light bulb to `"Off"` but in our intentionally faulty version `status` is set to `"Not working"` instead. You will see how testing with JUnit enables us to discover the fault and helps pinpoint which part of the code is responsible.

The tests in `LightBulbTest` are unchanged from those you created in Activities 2 and 3. You may recall that the method `testOff()` contains an assertion that, following the message `off()`, the `status` of the `LightBulb` object should be `"Off"`. So when we test our (faulty) class `LightBulb`, JUnit should report that it fails `testOff()`.

1 Run the tests.

Note that if the Test Results window is not visible and the Show Test Results option in the View menu is already ticked, then you will need to restore the minimised Test Results window from your taskbar.

Right-click on the `LightBulbTest` icon and choose Test All. The Test Results window will open. The red bar across the middle of the window shows that at least one of the tests has failed. In the top pane a green tick indicates that `LightBulbTest.testOn()` was completed successfully, but the grey cross next to `LightBulbTest.testOff` tells us that this test failed. JUnit reports 2 Runs, 0 Errors and 1 Failure (the result of our deliberate mistake!)

For more information about the failure, click on LightBulbTest.testOff in the top pane of the window. Detailed information will appear in the lower pane.

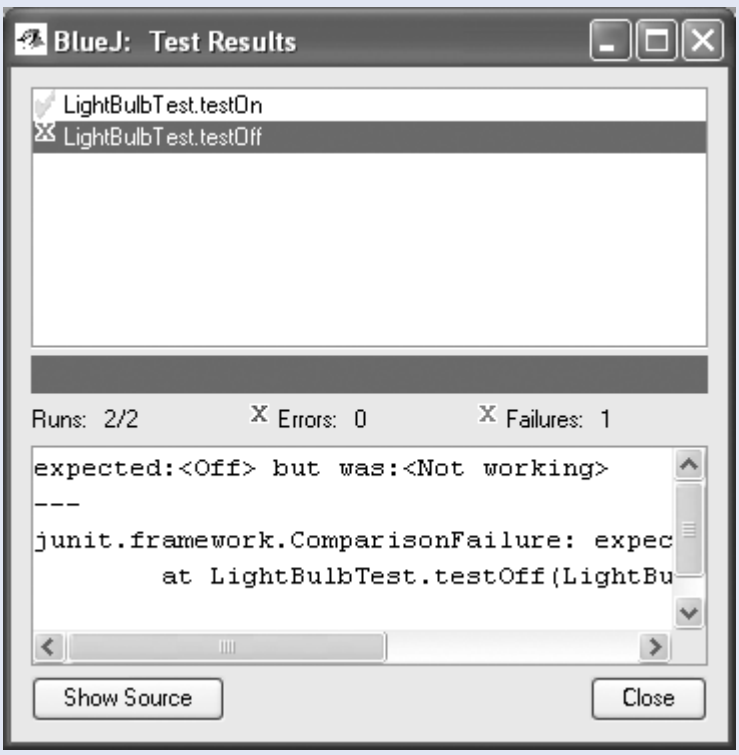


Figure 16 The Test Results window showing that the method `testOff()` failed. Detailed information about the cause of failure is displayed in the lower pane

2 Find the cause of the error.

The first line of the message in the Test Results window (Figure 16) is extremely useful in determining the cause of the error. It explains that it expected to receive the value `"Off"` from a method, but instead received the value `"Not working"`. Since this value differs from that asserted in the test, the method failed the test.

Click on the Show Source button (at the bottom left of the Test Results window). The source code for the `testOff()` method will appear in a new window, with the line of code that caused the failure highlighted in yellow. In this case the line responsible for the failure is

```
assertEquals("Off", lightBul1.getStatus());
```

This statement asserts that the result of sending the message `getStatus()` to `lightBul1` should equal `"Off"`, the value expected from the specification. This was not the case, because `getStatus()` returned `"Not working"` instead. So the assertion failed.

By examining the position of this line in the test program we can determine that the value of `status` was `"Not working"` immediately after `LightBul1` was sent the message `off()`. Therefore the first place to look for a possible error should be in the method `off()`.

3 Correct the code.

In the BlueJ window double-click on the `LightBulb` icon to open the editor. Find the method `off()` and edit it so that it becomes:

```
public void off()
{
    this.status = "Off";
}
```

Recompile the class and close the editor. Right-click on `LightBulbTest` and choose Test All. This time all the tests should be passed.

2.5 Test fixtures

When we created tests for methods of class `LightBulb` we created individual instances of `LightBulb` for each test method in the class `LightBulbTest`. If we had many test methods this would lead to lots and lots of duplicated coded. To avoid this we can use a **test fixture**.

A test fixture is a set of objects that are created by a special set-up method. Using a test fixture means that individual test methods no longer need to create test objects. Every time a test method is executed, the set-up method is automatically called first, and the test method simply uses the objects in the fixture.

Note that new test objects are created afresh for each test method, so different test methods do not share the same objects.

A test fixture is used in the next activity where we introduce a new class `Switch`.

ACTIVITY 5

Open Unit13_Project_6 which contains the classes `LightBulb`, `LightBulbTest` and `Switch`. The class `Switch` is new and models a switch that controls a light bulb. Each `Switch` object has a `LightBulb` object which it is responsible for turning on or off.

If you inspect the code for the `Switch` class you will find that it has an instance variable which is declared as `private LightBulb bulb`. This makes `Switch` dependent on `LightBulb`. So if you change any of the code in the class `LightBulb` then the classes `LightBulb` and `Switch` must both be recompiled.

In this activity you will create a test fixture for a new test class, `SwitchTest`. Right-click on `Switch` and choose Create Test Class from the menu. The icon of a new test class called `SwitchTest` will appear in the BlueJ window.

Before we can proceed further, we must create some test objects. We shall need a switch on which to perform our tests, and since each `Switch` object controls a `LightBulb` object, we will need a light bulb as well.

First create an instance of `LightBulb`. Right-click on the `LightBulb` icon and choose new LightBulb(), accepting the default name `lightBul1`. The new `LightBulb` object will be displayed on the Object Bench.

Now create an instance of `Switch`. Right-click on the `Switch` icon and choose new Switch(LightBulb bulb). A Create Object dialogue box will appear, as in Figure 17.

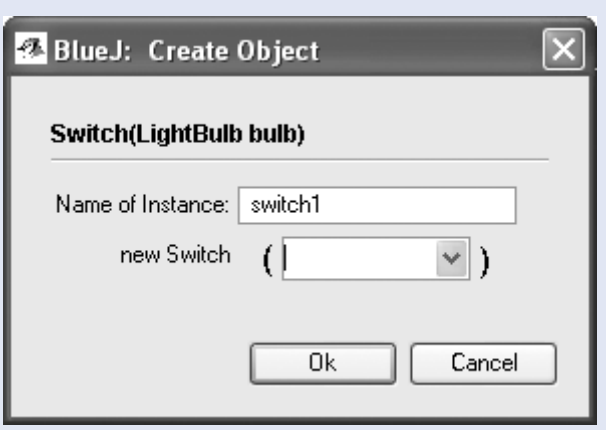


Figure 17 The Create Object dialogue box that appears when a new instance of `Switch` is created, requesting the name of an instance of `LightBulb` that will be passed to the `Switch` constructor

The new `Switch` object (in this case `switch1`) must be linked to a particular `LightBulb` object, so we need to pass a `LightBulb` argument to the `Switch` constructor. We have already created `lightBul1` for this purpose. Type `lightBul1` into the new Switch field and click the Ok button. Now `switch1` object will join `lightBul1` on the Object Bench.

Make sure you type lightBul**1**, not lightBul**l**.

Once we have our two test objects we are ready to create a test fixture. Right-click on the icon for the `SwitchTest` class and choose Object Bench to Test Fixture. The two objects on the Object Bench disappear and become part of `SwitchTest`! If you open the editor on `SwitchTest` you will see the following code.

```
public class SwitchTest extends junit.framework.TestCase
{
    private LightBulb lightBul1;
    private Switch switch1;

    /**
     * Default constructor for test class SwitchTest
     */
    public SwitchTest()
    {
        super();
    }

    /**
     * Sets up the test fixture. Automatically
     * called before every test case method.
     */
    protected void setUp()
    {
        lightBul1 = new LightBulb();
        switch1 = new Switch(lightBul1);
    }

}
```

If you wish to amend or extend the test fixture it is possible to put the objects back on the bench. Right-click on the icon for the `SwitchTest` class and choose Test Fixture to Object Bench. The objects will reappear. If you then try to reuse these test objects as a test fixture (by right-clicking on `SwitchTest` and selecting Object Bench to Test Fixture) a dialogue box, like the one in Figure 18, will appear. This warns you that the unit test class already has a test fixture declared in it, and that proceeding will cause that text fixture to be replaced. The dialogue box gives you the opportunity to cancel the operation or to replace the test fixture with the objects currently on the object bench.

Now you are ready to create tests for the class `Switch`.

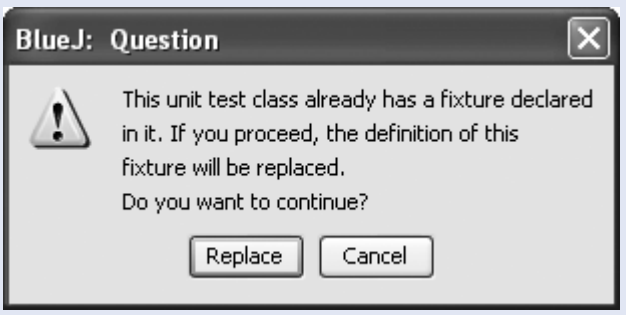


Figure 18 The dialogue box that appears when an attempt is made to add a test fixture to a class that already contains a test fixture

In case you had any problems with this activity, the class `SwitchTest` *and its test fixture has been added to Unit13_Project_7.*

Unit tests for the `Switch` class

The class `Switch` has two methods: `switchOn()` and `switchOff()` which relay the messages `On()` and `Off()`, respectively, to a `LightBulb` object, causing the latter's `status` to change. Therefore, once the `Switch` class has been tested, we will no longer need to send `On()` and `Off()` messages to `LightBulb` objects directly. Instead we will be able to send the messages `switchOn()` or `switchOff()` to an instance of `Switch` instead.

The messages we need to send to test `switchOn()` are shown in Table 5. Notice that more than one receiver is now involved.

Table 5 Message-sends needed to test `switchOn()` in `Switch`

Message-send		Reason
1	`lightBul1.getStatus()`	To check the `status` of `lightBul1` before sending the message `switchOn()`. The `status` of a newly initialised `LightBulb` instance should be `"Off"`.
2	`switch1.switchOn()`	This is the method we are testing.
3	`lightBul1.getStatus()`	To check the `status` of `lightBul1` after sending the message `switchOn()` to `switch1`. The `status` should now be `"On"`.
4	`switch1.switchOn()`	This is the method we are testing.
5	`lightBul1.getStatus()`	To check the `status` of `lightBul1` after sending the message `switchOn()` to `switch1` when the value of the `status` of `lightBul1` is already `"On"`. The value of `status` should still be `"On"`.

SAQ 2

What sequence of messages will you need to send to test the method `switchOff()`?

ANSWER.....

To test `switchOff()` you need the message-sends shown in Table 6.

Table 6 Message-sends needed to test `switchOff()` in `Switch`

Message-send		Reason
1	`lightBul1.getStatus()`	To check the `status` of `lightBul1` before sending the message `switchOff()`. The `status` of a newly initialised `LightBulb` instance should be `"Off"`.
2	`switch1.switchOff()`	This is the method we are testing.
3	`lightBul1.getStatus()`	To check the `status` of `lightBul1` after sending the message `switchOff()` to `switch1` when the value of the `status` of `lightBul1` is already `"Off"`. The value of `status` should still be `"Off"`.
4	`lightBul1.on()`	The `status` of a `lightBul1` should still be `"Off"`, so we must change `status` to `"On"` for another test of `switchOff()`.
5	`lightBul1.getStatus()`	To check the `status` of `lightBul1` is now `"On"`.
6	`switch1.switchOff()`	This is the method we are testing.
7	`lightBul1.getStatus()`	To check the `status` of `lightBul1` after sending the message `switchOff()` to `switch1` when the value of the `status` of `lightBul1` is `"On"`. The value of `status` should now be `"Off"`.

Notice that in step 4 we send an `on()` message to the `LightBulb` object directly rather than sending a `switchOn()` message to the `Switch` object. We did this to make the tests of `switchOff()` and `switchOn()` completely independent of each other. It is debatable whether the message-send in step 5 is needed as it could be argued that the method `on()` of `LightBulb` has already been tested.

ACTIVITY 6

Open Unit13_Project_7 to which the class `SwitchTest` has been added.

1 Right-click on the `SwitchTest` icon and choose Create Test Method. Give the method a suitable name, `testSwitchOn`, say. The objects `switch1` and `lightBul1` will reappear on the Object Bench, allowing you to send them messages.

Record the message-sends given in Table 5. When the Method Results window appears in steps 1, 3 and 5, check that the actual results match the expected ones before clicking Close to accept the assertion.

When you have finished, click End to finish recording (you will notice that `switch1` and `lightBul1` will disappear from the Object Bench).

Run your new test by right-clicking on the `SwitchTest` icon and choosing Test SwitchOn. The test should run without any errors or failures.

2 Now record the corresponding test for the method `switchOff()` (using the message-sends given in Table 6). Again, when the Method Results window appears in steps 1, 3, 5 and 7, check that the actual results match the expected ones.

Run your new test by right-clicking on `SwitchTest` and choosing Test SwitchOff. The test should run without any errors or failures.

3 Finally check the code that has been generated in `SwitchTest` for the methods `switchOn()` and `switchOff()`.

DISCUSSION OF ACTIVITY 6

The code that has been generated in `SwitchTest` for the methods `switchOn()` and `switchOff()` should be as follows.

```
public void testSwitchOn()
{
    assertEquals("Off", lightBul1.getStatus());
    switch1.switchOn();
    assertEquals("On", lightBul1.getStatus());
    switch1.switchOn();
    assertEquals("On", lightBul1.getStatus());
}
public void testSwitchOff()
{
    assertEquals("Off", lightBul1.getStatus());
    switch1.switchOff();
    assertEquals("Off", lightBul1.getStatus());
    lightBul1.on();
    assertEquals("On", lightBul1.getStatus());
    switch1.switchOff();
    assertEquals("Off", lightBul1.getStatus());
}
```

In case you had any problems with this activity, the test methods have been added to `SwitchTest` *in Unit13_Project_7_Completed.*

2.6 Hand coding JUnit tests

Relatively few IDEs currently support the recording of test methods; hand coding is still the most common way of writing test methods. Therefore to prepare you for writing JUnit tests in other IDEs, you will hand code an extremely trivial test class, a test method and a test fixture for the class `Account`. The purpose of the test method is to test the method `setHolder()` of `Account`.

Here is the initial comment for the method `setHolder()`.

```
/**
 * Set the holder of the receiver to the value of the argument accountHolder
 */
```

SAQ 3

What sequence of messages do you think you would need to send to an instance of `Account` referenced by a variable `acc` to test the method `setHolder()`?

ANSWER.....

The message-sends shown in Table 7 are needed.

Table 7 Message-sends needed to carry out the unit testing of `setHolder()`

Message-send		Reason
1	`acc.getHolder()`	To check the `holder` of `acc` before sending a `setHolder()` message. The `holder` of a newly initialised `Account` instance should be `""`.
2	`acc.setHolder("Fred")`	Attempt to set the `holder` to some string.
3	`acc.getHolder()`	To check the `holder` of `acc` after sending the message `setHolder()`. The value of `holder` should be `"Fred"`.

ACTIVITY 7

Open Unit13_Project_8 which contains the class `Account`. Click the New Class... button on the left-hand side of the BlueJ window. The familiar Create New Class dialogue box will appear. Give the new class a suitable name (such as `AccountTest`), then select the Unit Test radio button so that the class created will be a test class (Figure 19).

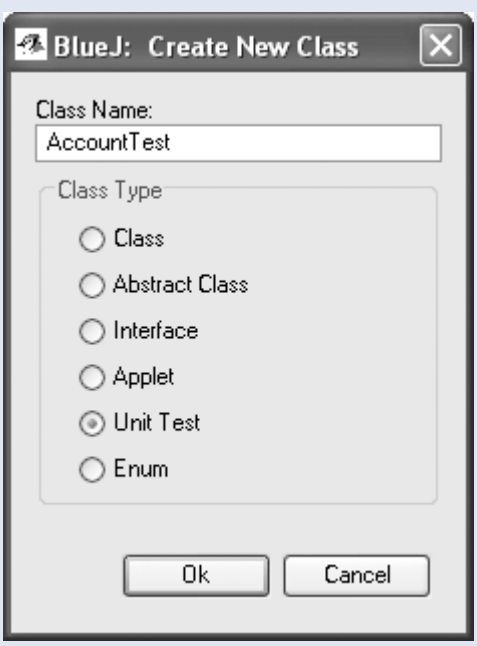


Figure 19 Creating a test class

1 Click the Ok button to close the dialogue box. The icon for the new test class will appear in the main BlueJ window. Note that rather than being attached to the class `Account`, the class `AccountTest` is represented by a totally separate icon that can be dragged around the BlueJ window independently of the `Account` icon. This will make no difference to the efficacy of the test class, it is simply an idiosyncracy of BlueJ's implementation of JUnit.

2 Create the test fixture for `AccountTest`.

Before you can proceed further, you need to create the test fixture for the class `AccountTest`. Double-click on the icon for the class `AccountTest` to open the editor.

Declare a private instance variable called `acc` of type `Account`.

Write the code in the method `setUp()` to assign a new `Account` object to the instance variable `acc`.

3 Write the test method.

Write a method with the header:

```
public void testSetHolder()
```

The code should carry out the steps outlined in Table 7. For message-sends 1 and 3, rather than simply writing `acc.getHolder()`, you will need to write an assert statement that tells JUnit what value you expect `getHolder()` to return. For example, if in message-send 2 you sent the message `setHolder("Fred")` to `acc`, message-send 3 should be:

```
assertEquals("Fred", acc.getHolder());
```

Once you have written the method, attempt to compile the class. Once the class compiles you are ready to run your test.

4 Run the test.

Right-click `AccountTest` and choose Test Holder or Test All from the pop-up menu. The unit test should succeed, with no errors or failures.

An alternative way to run the test is to click the Run Tests button in the left-hand margin on the BlueJ window. This will run all the test methods in all the test classes in the project (although there is only one test class in this case, of course).

Congratulations! You have successfully hand coded a test class. You are now ready to use JUnit.

DISCUSSION OF ACTIVITY 7

The code for the class `AccountTest` should be as follows.

```
public class AccountTest extends junit.framework.TestCase
{
    private Account acc;

    /**
     * Default constructor for test class AccountTest
     */
    public AccountTest()
    {
        super();
    }

    /**
     * Sets up the test fixture. Automatically
     * called before every test case method.
     */
    protected void setUp()
    {
        acc = new Account();
    }
    public void testSetHolder()
    {
        assertEquals("", acc.getHolder());
        acc.setHolder("Fred");
        assertEquals("Fred", acc.getHolder());
    }
}
```

This activity was, of course, extremely simple, but it demonstrated how you might hand code a test class and a test fixture. Indeed you may at times find it quicker and easier to hand code test fixtures and test methods once you become more confident with JUnit.

In case you had any problems with this activity, the class `AccountTest` *has been added to Unit13_Project_8_Completed.*

3 Unit testing the class CompuFrog

We now return to the class CompuFrog you encountered in Subsection 1.2. First you will retest its method add() using JUnit. As before, the method fails the test. Once you have fixed the fault and the method passes the test, we show you how to extend the arithmetical capabilities of compufrogs. We will work incrementally, writing and running unit tests as we go.

ACTIVITY 8

Open the project Unit13_Project_9 which contains the class CompuFrog. Your task in this activity is to create a test class for CompuFrog called CompuFrogTest with a test method called testAdd() to test the method add() of CompuFrog.

1 Right-click on the CompuFrog icon and choose Create Test Class. The icon for CompuFrogTest will appear in the main BlueJ window.

2 Create a test fixture (which will be an instance of CompuFrog) for the class CompuFrogTest. You can do this in two ways:

- by hand coding the variable declaration and the code for the setUp() method, as you did in Activity 7, or
- by right-clicking on the CompuFrog icon, selecting new CompuFrog from the menu, accepting the name compuFro1 and pressing Ok; then right-clicking on the CompuFrogTest icon and choosing Object Bench to Test Fixture.

3 Table 8 shows the test plan that we devised for the method add() of CompuFrog in Section 1.

Table 8 Test cases for the method add() of CompuFrog

Test case	1st argument	2nd argument	Expected result	Rationale
1	0	0	0	Both numbers on lower boundary
2	–1	–1	–1	Both numbers outside lower boundary
3	0	1	1	First number on lower boundary, second close but inside lower boundary
4	0	–1	–1	First number on lower boundary, second close but outside lower boundary
5	1	0	1	First number close but inside lower boundary, second number on lower boundary
6	–1	0	–1	First number close but outside lower boundary, second number on lower boundary

7	1	–1	–1	First number close but inside lower boundary, second number close but outside lower boundary
8	–1	1	–1	First number close but outside lower boundary, second number close but inside lower boundary
9	1	1	2	Both numbers near but inside lower boundary
10	5	7	12	Typical values near middle of range
11	8	8	16	Both numbers near but inside upper boundary
12	8	9	17	First number close but inside upper boundary, second number on upper boundary
13	10	9	–1	First number close but outside upper boundary, second number on upper boundary
14	9	8	17	First number on upper boundary, second close but inside upper boundary
15	9	10	–1	First number on upper boundary, second close but outside upper boundary
16	9	9	18	Both numbers on upper boundary
17	10	10	–1	Both numbers outside upper boundary
18	8	10	–1	First number close but inside upper boundary, second number close but outside upper boundary
19	10	8	–1	First number close but outside upper boundary, second number close but inside upper boundary

Using the above table, we want you to create a method called `testAdd()` for `CompuFrogTest` that will test the method `add()` of `CompuFrog`. You can do this by using BlueJ's recording facility or by hand coding.

4 Once you have written the test method and the class `CompuFrogTest` has recompiled successfully, run the unit test. The test should show that the method `add()` fails! Note that although both tests 12 and 16 should fail, the execution of a test method ceases as soon as an assertion fails, hence JUnit only reports the failure of test 12.

You will now fix the bug that caused the test to fail.

5 Open the class `CompuFrog` and inspect the method `add()`.

The error is in line //1, where the programmer has used the condition `(y < 9)` when it should be `(y <= 9)`. Correct this. Close the editor and run the test again. This time it should pass with flying colours! (If it does not, go back and check the code.)

DISCUSSION OF ACTIVITY 8

Here is our code for the class `CompuFrogTest`.

```
public class CompuFrogTest extends junit.framework.TestCase
{
    private CompuFrog compuFro1;

    /**
     * Default constructor for test class CompuFrogTest
     */
    public CompuFrogTest()
    {
        super();
    }

    /**
     * Sets up the test fixture. Automatically
     * called before every test case method.
     */
    protected void setUp()
    {
        compuFro1 = new CompuFrog();
    }

    public void testAdd()
    {
        assertEquals(0, compuFro1.add(0, 0));
        assertEquals(-1, compuFro1.add(-1, -1));
        assertEquals(1, compuFro1.add(0, 1));
        assertEquals(-1, compuFro1.add(0, -1));
        assertEquals(1, compuFro1.add(1, 0));
        assertEquals(-1, compuFro1.add(-1, 0));
        assertEquals(-1, compuFro1.add(1, -1);
        assertEquals(-1, compuFro1.add(-1, 1);
        assertEquals(2, compuFro1.add(1, 1));
        assertEquals(12, compuFro1.add(5, 7));
        assertEquals(16, compuFro1.add(8, 8));
        assertEquals(17, compuFro1.add(8, 9));
        assertEquals(-1, compuFro1.add(10, 9));
        assertEquals(17, compuFro1.add(9, 8));
        assertEquals(-1, compuFro1.add(9, 10));
        assertEquals(18, compuFro1.add(9, 9));
        assertEquals(-1, compuFro1.add(10, 10));
        assertEquals(-1, compuFro1.add(8, 10));
        assertEquals(-1, compuFro1.add(10, 8));
    }
}
```

In case you had any problems with this activity, the class `CompuFrogTest` *and the corrected method* `add()` *have been added to Unit13_Project_10.*

In the next activity you will give compufrogs the ability to multiply.

ACTIVITY 9

Open the project Unit13_Project_10 which contains the class `CompuFrog` and the test class `CompuFrogTest` as so far developed.

The class `CompuFrog` is to have a new method with the following header and initial comment:

```
/**
 * The method takes two integer arguments x and y. If these
 * arguments are one-digit positive integers (ie integers
 * between 0 and 9 inclusive) the method sets the position to
 * the product (multiplication) of the two arguments and then
 * returns that product. Otherwise the method simply returns -1.
 */
public int times(int x, int y)
```

1 Begin by writing (on paper) a suitable set of black box tests for the method. The test cases can be similar to those for `add()` shown in Table 8, except of course the expected values will be different (remember to replace `add()` by `times()`!).

2 Once you have sketched out the tests, write the method, taking `add()` as a guide, including the code that checks that the inputs are both single digits.

3 Add a new test method called `testTimes()` to `CompuFrogTest`.

4 Recompile and press the Run button. If all goes well, you will see confirmation that the method has passed both tests. If `testTimes()` fails, go back and check the coding. You can find our version of the method in the discussion below.

DISCUSSION OF ACTIVITY 9

1 Table 9 shows our test plan.

Table 9 Test cases for the method `times()` of `CompuFrog`

Test case	1st argument	2nd argument	Expected result	Rationale
1	0	0	0	Both numbers on lower boundary
2	–1	–1	–1	Both numbers outside lower boundary
3	0	1	0	First number on lower boundary, second close but inside lower boundary
4	0	–1	–1	First number on lower boundary, second close but outside lower boundary
5	1	0	0	First number close but inside lower boundary, second number on lower boundary
6	–1	0	–1	First number close but outside lower boundary, second number on lower boundary

7	1	–1	–1	First number close but inside lower boundary, second number close but outside lower boundary
8	–1	1	–1	First number close but outside lower boundary, second number close but inside lower boundary
9	1	1	1	Both numbers near but inside lower boundary
10	5	7	35	Typical values near middle of range
11	8	8	64	Both numbers near but inside upper boundary
12	8	9	72	First number close but inside upper boundary, second number on upper boundary
13	10	9	–1	First number close but outside upper boundary, second number on upper boundary
14	9	8	72	First number on upper boundary, second close but inside upper boundary
15	9	10	–1	First number on upper boundary, second close but outside upper boundary
16	9	9	81	Both numbers on upper boundary
17	10	10	–1	Both numbers outside upper boundary
18	8	10	–1	First number close but inside upper boundary, second number close but outside upper boundary
19	10	8	–1	First number close but outside upper boundary, second number close but inside upper boundary

2 Our version of the `times()` method for the `CompuFrog` class is as follows.

```
/**
 * The method takes two integer arguments x and y. If these
 * arguments are one-digit positive integers (ie integers
 * between 0 and 9 inclusive) the method sets the position to
 * the product (multiplication) of the two arguments and then
 * returns that product. Otherwise the method simply returns -1.
 */
public int times(int x, int y)
{
    if (((x >= 0) && (x <= 9)) && ((y >= 0) && (y <= 9)))
    {
        int product = x * y;
        this.setPosition(product);
        return product;
    }
    else
    {
        return -1; // one or more inputs not single-digit
    }
}
```

3 Our version of the method `testTimes()` for the `CompuFrogTest` class is as follows.

```
public void testTimes()
{
    assertEquals(0, compuFro1.times(0, 0));
    assertEquals(-1, compuFro1.times(-1, -1));
    assertEquals(0, compuFro1.times(0, 1));
    assertEquals(-1, compuFro1.times(0, -1));
    assertEquals(0, compuFro1.times(1, 0));
    assertEquals(-1, compuFro1.times(-1, 0));
    assertEquals(-1, compuFro1.times(1, -1));
    assertEquals(-1, compuFro1.times(-1, 1));
    assertEquals(1, compuFro1.times(1, 1));
    assertEquals(35, compuFro1.times(5, 7));
    assertEquals(64, compuFro1.times(8, 8));
    assertEquals(72, compuFro1.times(8, 9));
    assertEquals(-1, compuFro1.times(10, 9));
    assertEquals(72, compuFro1.times(9, 8));
    assertEquals(-1, compuFro1.times(9, 10));
    assertEquals(81, compuFro1.times(9, 9));
    assertEquals(-1, compuFro1.times(10, 10));
    assertEquals(-1, compuFro1.times(8, 10));
    assertEquals(-1, compuFro1.times(10, 8));
}
```

In case you had any problems with this activity, the class `CompuFrogTest` *and the method* `times()` *have been added to Unit13_Project_11.*

3.1 Refactoring the class CompuFrog

As you have been working on the methods `add()` and `times()` of `CompuFrog`, it might have struck you that these methods exhibit code duplication – both methods use the same condition `(((x >= 0) && (x <= 9)) && ((y >= 0) && (y <= 9)))` to check that the arguments are positive single-digit integers. This is not very elegant and a much better solution would be to factor out the redundant code into a helper method which checks that an integer is a positive single digit. This is precisely what you will do in the next activity.

ACTIVITY 10

In this activity you will improve the design of `CompuFrog` by refactoring. Open the project Unit13_Project_11 which contains the class `CompuFrog` as so far developed, including the test class and the tests for `add()` and `times()`.

Here is the method header and initial comment for a helper method that will enable you to get rid of the code duplication in the methods `add()` and `times()` of `CompuFrog`.

```
/**
 * The method takes a single argument aNumber. If aNumber
 * is a one-digit positive integer (ie an integer
 * between 0 and 9 inclusive) the method returns true.
 * Otherwise the method returns false.
 */
public boolean isInRange(int aNumber)
```

Note that the method has been declared as `public` although we have previously stated that helper methods should usually be declared as `private`. The reason for this is to allow testing – if `isInRange()` were to be declared as `private` we would not be able to test it from the class `CompuFrogTest`. The method can be re-declared as `private` once all testing is complete.

1 Open the editor on the `CompuFrog` class and write the method `isInRange(int aNumber)`.

2 Once you have written the method and got the `CompuFrog` class to compile, right-click on the `CompuFrogTest` icon, choose Create Test method... and record a new test testIsInRange. Send the message `isInRange()` seven times, with the successive arguments `-1`, `0`, `1`, `5`, `8`, `9`, `10`, accepting the Method Results `false`, `true`, `true`, `true`, `true`, `true`, `false`, respectively. (If you get any different results you will need to check the coding of your method.) Press End to finish recording.

   Now open `CompuFrogTest` and check that the method `testIsInRange()` contains all seven tests. If any are missing, add them manually.

3 Making sure that `CompuFrogTest` is compiled, right-click on the `CompuFrogTest` icon and choose Test IsInRange. If all is well you should find that `isInRange()` passes the test. If not, go back and look for the error.

4 Change the methods `add()` and `times()` so they now check the arguments `x` and `y` using `isInRange()` instead of the complicated condition `(((x >= 0) && (x <= 9)) && ((y >= 0) && (y <= 9)))`.

5 Rerun *all* the tests you have developed so far for `CompuFrog` by pressing Run Tests. If you have made the modifications correctly all the tests should be passed. If any test is failed, re-examine the code, correct it and retest.

Note you have not needed to change `testAdd()` or `testTimes()` in any way even though you have made changes to both `add()` and `times()`. This is because the changes you made did not change the specifications of the methods; they both still take the same types of argument and return values of the same type as before. Once test methods are written, they can be run again and again without alteration, unless the *specifications* of the methods under test are changed.

Our version of `isInRange()` and the changes to `add()` and `times()` appear in the discussion below.

DISCUSSION OF ACTIVITY 10

Our version of `isInRange()` is as follows (the comment is left out for brevity).

```
public boolean isInRange(int aNumber)
{
    return ((aNumber >= 0) && (aNumber <= 9));
}
```

In the methods `add()` and `times()` we replaced the condition

```
if (((x >= 0) && (x <= 9)) && ((y >= 0) && (y <= 9)))
```

with

```
if (this.isInRange(x) && this.isInRange(y))
```

In case you had any problems with this activity, the class `CompuFrogTest` *and the corrected method* `isInRange()` *have been added to Unit13_Project_11_Completed.*

Now that you have refactored the class `CompuFrog`, any other arithmetic methods you might write in the future can just use `isInRange()` instead of doing the checking themselves. As well as leading to better factored code, this also means that if compufrogs one day learn to calculate with two-digit numbers, only `isInRange()` would need to be altered, instead of having to change all the arithmetic methods. Of course you would have to alter the test methods as well.

SAQ 4

Is there any further refactoring that could be done for the class `CompuFrog`?

ANSWER.....

Yes, both `add()` and `times()` have the condition

```
(this.isInRange(x) && (this.isInRange(y))
```

so we could refactor further and introduce another helper method as follows:

```
public bool bothInRange(int num1, int num2)
{
    return (this.isInRange(num1) && (this.isInRange(num2));
}
```

The condition in both `add()` and `times()` would then be simplified to:

```
if (this.bothInRange(x, y))
```

Exercise 3

If compufrogs learnt to handle numbers up to 99, what test cases would you then use for the method `add()`? Your answer should be a description which is similar to Table 8. You are not expected to carry out any tests.

Solution

Table 10 shows our test cases.

Table 10 Test cases for an updated method `add()` of `CompuFrog`

Test case	1st argument	2nd argument	Expected result	Rationale
1	0	0	0	Both numbers on lower boundary
2	–1	–1	–1	Both numbers outside lower boundary
3	0	1	1	First number on lower boundary, second close but inside lower boundary
4	0	–1	–1	First number on lower boundary, second close but outside lower boundary
5	1	0	1	First number close but inside lower boundary, second number on lower boundary
6	–1	0	–1	First number close but outside lower boundary, second number on lower boundary
7	1	–1	–1	First number close but inside lower boundary, second number close but outside lower boundary
8	–1	1	–1	First number close but outside lower boundary, second number close but inside lower boundary
9	1	1	2	Both numbers near but inside lower boundary
10	47	53	100	Typical values near middle of range
11	98	98	196	Both numbers near but inside upper boundary
12	98	99	197	First number close but inside upper boundary, second number on upper boundary
13	100	99	–1	First number close but outside upper boundary, second number on upper boundary
14	99	98	197	First number on upper boundary, second close but inside upper boundary
15	99	100	–1	First number on upper boundary, second close but outside upper boundary
16	99	99	198	Both numbers on upper boundary
17	100	100	–1	Both numbers outside upper boundary
18	98	100	–1	First number close but inside upper boundary, second number close but outside upper boundary
19	100	98	–1	First number close but outside upper boundary, second number close but inside upper boundary

4 More about designing test data

In this section we take a more systematic look at the question of how to choose an adequate set of test cases

4.1 Boundary and computation errors

Imagine that an assignment has a total of 80 marks available and consider a very simple program which takes an assignment score out of 80 and scales it to a percentage (i.e. a raw score of 80 would scale to 100 per cent). Scores not in the range 0 to 80 are rejected as invalid, with a warning to the user.

Imagine that this is an existing product which we need to test before it is distributed and that we do not have access to the source code.

What test cases are we going to use? As we discussed previously, the only way to be *completely* sure that the program is correct is to test every possible input. However the number of possibilities is far too large for this to be feasible and we need a way to pick a fairly small set of cases that will nevertheless have a high probability of detecting errors in the program.

To understand how to do this, we think about the possible inputs and the ways in which things could go wrong so that the program would produce the wrong answer. In very general terms, when a program receives an input, a condition is evaluated to decide which branch of the program should be taken. Then, when this branch is followed, some particular action is carried out.

In the example we are considering, the condition would be something like whether the input is valid; if it is, the action should be to convert it to a percentage and return the result; otherwise the program should warn the user that the data was not valid.

Now remember testing is a search for defects, so we try to imagine what sort of things could go wrong when a particular number, such as 40, is input. We can distinguish two basic forms of error: **boundary errors** and **computation errors**.

A boundary error is when the input is classified wrongly and as a result the program takes the wrong action. For example, 79 might be rejected as an invalid mark. We call this a boundary error because when a program has to decide between alternatives this usually involves working out on which side of some boundary an input lies. In our example there are two boundaries, one at 0 and one at 80, and they separate the valid from the invalid cases.

A computation error is when the program classifies the input correctly, but does the wrong thing to it. For instance, the mark-scaling program might correctly classify 40 as a valid input, but then divide it by 1.25, when it should have multiplied by 1.25. The result would then be 32 per cent when it should have been 50 per cent. Or, as another example, if the input were greater than 80, a computation error might lead to the user being given the wrong warning message, even though the input was correctly identified as invalid.

Where does 1.25 come from? Multiplying our raw scores by 1.25 will give us the percentage, because 1.25 is equal to 100/80.

Thus a boundary error is when the program chooses the wrong course of action and a computation error is when it chooses the right course of action but carries it out incorrectly.

Now how does this help us choose test cases? Well we can think of the boundaries as splitting the range of possible inputs into distinct sets of possible values, usually called **equivalence classes**. This term derives from the fact that all the inputs in the same set (equivalence class) will normally be subject to the same computation, so they are all equivalent from the testing point of view, because, if the computation is programmed wrongly, any one of these inputs can be expected to show up the error. It is normally sufficient to choose just one representative input value from each equivalence class.

The word 'class' in the term 'equivalence class' has nothing to do with class in the object-oriented sense. The term predates object-oriented programming. Here the word 'class' is being used to mean 'set of values'.

For instance, Figure 20 shows a situation where there are three equivalence classes and two boundaries. For each equivalence class we pick a typical value, which we have shown as a blob.

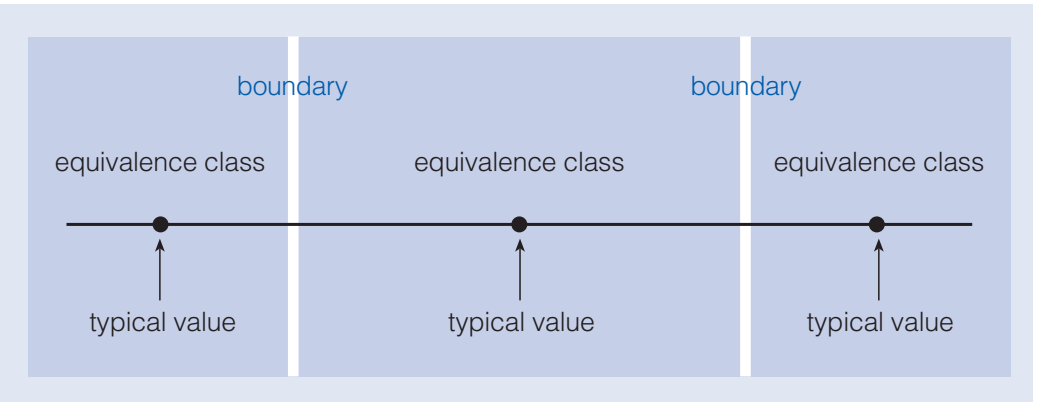


Figure 20 Choosing a typical value from equivalence classes

This approach is good for detecting computation errors, but does not work very well with boundary errors. Although the inputs we choose to represent the equivalence classes may get classified correctly, things may still go wrong closer to the boundaries. This is because programmers frequently make mistakes when programming conditions. For instance, a programmer might very easily write `(x > 0)` instead of `(x >= 0)` and an input of 0 would then be handled incorrectly.

To look for boundary errors we normally do a test on the boundary itself and at two values near it, one on either side of the boundary (Figure 21).

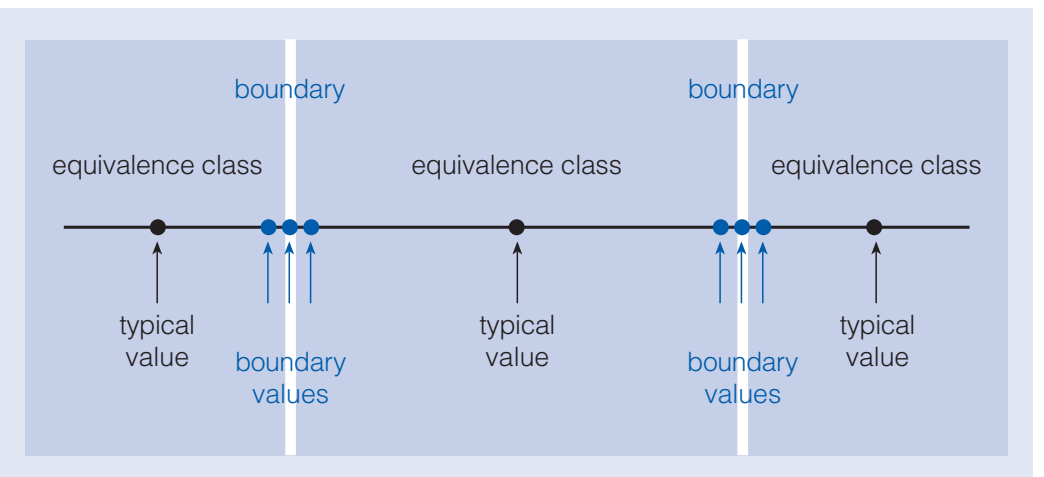


Figure 21 Testing near to the boundary values

In the case of our imaginary program for converting raw marks to percentages we have three equivalence classes as follows (the boundaries are at 0 and 80):

- input values less than 0;
- input values between 0 and 80 inclusive;
- input values more than 80.

Choosing typical values for each equivalence class, and values at each boundary and close by on either side gives the test inputs shown in Table 11.

Table 11 Test cases for the raw scores to percentages program

Test	Input	Rationale	Type of error this is designed to detect
1	–10	Typical value below 0	Computation
2	–1	Just below boundary	Boundary
3	0	Boundary	Boundary
4	1	Just above boundary	Boundary
5	42	Typical value between 0 and 80	Computation
6	79	Just below boundary	Boundary
7	80	Boundary	Boundary
8	81	Just above boundary	Boundary
9	90	Typical value above 80	Computation

SAQ 5

Look back at Activity 10 and consider the `isInRange()` method for instances of `CompuFrog`. What equivalence classes can be determined for possible values of the method's argument?

ANSWER.....

The possible values for the `isInRange()` method's argument fall into three equivalence classes:

- values less than 0;
- values between 0 and 9 inclusive;
- values more than 9.

SAQ 6

In Activity 10 we stated that the method `isInRange()` should be tested with successive arguments of `-1`, `0`, `1`, `5`, `8`, `9`, `10`. In the light of our discussion on equivalence classes can you think of any other values that the method should have been tested with?

ANSWER.....

There are three equivalence classes and two boundaries. In testing `isInRange()` in Activity 10, we correctly tested with values just below (`-1`), on (`0`) and just above (`1`) the lower boundary; with values just below (`8`), on (`9`) and just above (`10`) the upper boundary; and with a typical value from the equivalence class 0 to 9 inclusive (`5`).

However, we omitted to test with *typical* values from the other two equivalence classes (values less than 0 and values more than 9). Therefore to fully test the method we should

have tested with a value some way below the lower boundary, such as `-8`, and with a value some way above the upper boundary, such as `17`.

Similarly, we omitted such tests for the `add()` and `times()` methods in order to keep down the number of tests and to avoid overwhelming you as we introduced the idea of testing around boundaries.

4.2 Using boundary values and equivalence classes

Summarising the ideas of Subsection 4.1 gives the following general strategy for choosing a small number of test cases that are likely to be effective at detecting errors.

1 Identify the boundary values and use them to divide the set of possible inputs into equivalence classes.

2 For each equivalence class choose a typical value to represent that class.

For greater security we may decide to test at more than one interior point.

3 Include a test at each boundary value and at two nearby values on either side of that boundary.

By using this approach we can choose a set of test cases which is small enough to be manageable but still retains a high chance of detecting the likely sorts of program error.

The next exercise asks you to put these ideas into practice.

Exercise 4

For both situations, (a) and (b), below you are asked to say what test cases you would use for the methods described, using the strategy of boundary and equivalence class testing.

For each test case, you should give the arguments, the state of the receiver object if the method is state dependent, and the expected results, i.e. the return value and the resulting state of any objects that have been changed. You can just list the tests; there is no need to use a table format unless you wish to.

(a) The Amphibian Transport Company operates a shuttle railway which runs between Olmby and Sirencester (a distance of 45 km), stopping at many places along the way. Tickets cost £2 for a journey of up to 12 km, £4 for a journey over 12 km but not more than 30km, and £6 to travel over 30 km.

A new method `ticketPrice()` is to be added to `CompuFrog` which takes as its argument an integer representing the distance a customer wishes to travel and returns the correct price for the ticket. If the value of the argument is less than 1 or greater than 45 (a non-valid distance) the method returns –1.

(b) A subclass of `Account` has been created, called `InterestAccount`, which pays interest on positive balances, at a sliding rate which depends on the current balance as follows.

Balances of at least 0.00 but less than 100.00 earn an interest rate of 0.05.

Balances of at least 100.00 but less than 500.00 earn an interest rate of 0.07.

Balances of 500.00 or over earn an interest rate of 0.10.

Overdrawn accounts (i.e. negative balances) earn no interest.

`InterestAccount` has an attribute `interestRate` and methods `getInterestRate()` and `computeInterestRate()`. When executed, `computeInterestRate()` sets the interest rate to the appropriate value for the receiver's current balance. The return value is `void`.

When picking a value of `balance` close to a boundary it should be as near as possible. When dealing with money, this means a difference of `0.01`.

Solution

(a) There are five equivalence classes:

- less than 1;
- at least 1 and not more than 12;
- more than 12 but not more than 30;
- more than 30 but not more than 45;
- greater than 45.

The method is not state dependent, so the state of the `compuFrog` object does not matter. We need to test the method with arguments `-5`, `0`, `1`, `2`, `6`, `11`, `12`, `13`, `21`, `29`, `30`, `31`, `37`, `44`, `45`, `46` and `50`. (You may have used different values for the equivalence class tests at `-5`, `6`, `21`, `37` and `50`.) The expected results are the return values, which should be, respectively, `-1`, `-1`, `2`, `2`, `2`, `2`, `2`, `4`, `4`, `4`, `4`, `6`, `6`, `6`, `6`, `-1` and `-1`.

(b) There are four equivalence classes:

- less than 0.00, i.e. overdrawn;
- at least 0.00 but less than 100.00;
- at least 100.00 but less than 500.00;
- more than 500.00.

The method is state dependent but takes no argument. There is no return value but the receiver changes state.

We need to test the method with the following balances of the account: `-50.00`, `-0.01`, `0.00`, `0.01`, `50.00`, `99.99`, `100.00`, `100.01`, `350.00`, `499.99`, `500.00`, `500.01` and `750.00`. (As before, you may have picked different equivalence class tests but your boundary tests should agree with ours.)

After the method has executed, the `interestRate` of the account (which we can find out by using `getInterestRate()`) should be, respectively, `0.00`, `0.00`, `0.05`, `0.05`, `0.05`, `0.05`, `0.07`, `0.07`, `0.07`, `0.07`, `0.10`, `0.10`, `0.10`.

4.3 When inputs are not ordered

The strategy discussed above assumes the inputs have some order to them – for example, numbers are ordered by size, strings are ordered alphabetically, and many other data such as days of the week or months of the year have their conventional order. But what if the input data are without order? For example, suppose they are just members of an unordered set. In such a case, the notion of boundaries, and testing at and close to them, does not mean anything. But it may still be possible to form equivalence classes and choose a representative test case from each one.

What happens, for example, if a subclass of `Frog` overrides the methods `right()` and `left()` so that instances of this subclass move right and left by two stones when their colours are red or brown but otherwise behave normally? In this case, there are two equivalence classes, red and brown, and all the other colours but there is no boundary between them. In an example like this, we can choose representative test cases from the equivalence classes as we did before but there will be no boundary tests.

5 Case study: congestion charging

In this section we reprise all you have learnt about testing by looking at a small case study. This case study will also introduce you to an important and interesting programming pattern called the Observer pattern. This pattern addresses the question of how objects can be made to keep in step with other objects.

5.1 Congestion charging for frogs

Recently the amphibian world has become concerned about growing traffic congestion. Certain stones in the central zone are heavily clogged with frogs at busy times and in order to reduce the congestion a system of charging is to be introduced.

Here is a description of the classes involved.

`ChargeableFrog`

Instances of `ChargeableFrog` are frogs which have bank accounts (with an initial balance of £250) that they use to pay for travel in the amphibian congestion charging zone. Each chargeablefrog is watched by a czwarden which charges £10 by direct debit each time a chargeablefrog makes a change of position to a stone in the outer charging zone, and £20 if the change of position takes the frog to a stone in the inner charging zone.

The outer charging zone is from stone 5 to stone 15 inclusive, and from stone 25 to stone 35 inclusive.

The inner charging zone is from stone 16 to stone 24 inclusive.

Stones 4 and below, and stones 36 and above, are outside the charging zones.

The stone that the chargeablefrog starts on is of no importance in the charging scheme.

`CZWarden`

An instance of `CZWarden` is an observer of a number of instances of `chargeableFrog`. Whenever one of the chargeablefrogs that the czwarden is observing changes the value of the instance variable `position`, the czwarden checks if the chargeablefrog has moved to a stone in a charging zone and, if so, charges £10 or £20 as appropriate, by direct debit from the chargeablefrog's bank account.

`Account`

This class should be familiar from earlier units.

Exercise 5

This exercise asks you to devise a series of test cases for the above scenario.

First you need to pick a suitable set of test cases. To test the charging scheme, you need to move a chargeablefrog to selected stones and check if its bank account is debited by the correct amount. As you will have realised, the charging zones are natural equivalence classes.

On paper, applying what you learnt in Section 4 about equivalence class and boundary testing, draw up a table of test cases for `ChargeableFrog`. All the tests should be carried out on a single instance of `ChargeableFrog` that has a bank account with an initial balance of £250 and which is sent the message `setPosition()`.

For each case, say which stone the frog should be moved to, what the charge will be, the new balance, and the rationale for the test. Note that chargeablefrogs must pay a charge each time they move to a stone in a charging zone. Even if they start from a stone in a charging zone already they must still pay again! Moving from either of the two charging zones to the free zones does not incur a charge.

Solution

Table 12 shows the tests we came up with.

Table 12 Test cases for `ChargeableFrog`

Test case	Destination stone	Charge	New balance	Rationale
1	2	none	250	Well outside all charging zones
2	4	none	250	Just outside outer zone lower boundary
3	5	–10	240	Outer zone lower boundary
4	6	–10	230	Just inside outer zone lower boundary
5	10	–10	220	Typical value near middle of range
6	15	–10	210	Just outside inner zone lower boundary
7	16	–20	190	Inner zone lower boundary
8	17	–20	170	Just inside inner zone lower boundary
9	20	–20	150	Typical value near middle of inner zone
10	23	–20	130	Just inside inner zone upper boundary
11	24	–20	110	Inner zone upper boundary
12	25	–10	100	Just outside inner zone upper boundary
13	30	–10	90	Typical value near middle of range
14	34	–10	80	Just inside outer zone upper boundary
15	35	–10	70	Outer zone upper boundary
16	36	none	70	Just outside outer zone upper boundary
17	40	none	70	Well outside all charging zones

In the above table we have chosen the boundaries of:

- 5 for the outer zone lower boundary;
- 16 for the inner zone lower boundary;
- 24 for the inner zone upper boundary;
- 35 for the outer zone upper boundary.

However, you may have chosen slightly different boundaries and indeed, when dealing with equivalence classes of adjacent discrete values, it is often debatable about where the boundaries should fall. For example you might have chosen:

- 5 for the outer zone lower boundary,
- 15 for the outer zone upper boundary (to the left of the inner zone),
- 25 for the outer zone lower boundary (to the right of the inner zone),
- 35 for the outer zone upper boundary.

Either set of boundary values could be used to formulate adequate test cases for `ChargeableFrog`.

The Observer pattern

Before we start testing the program we need to explain a little of how it works. How does the czwarden watch what the chargeablefrogs are doing?

The answer to this will also explain something else you may have wondered about at various times during the course – how does the graphical display manage to respond to changes in an amphibian's state – how does it 'know' there has been a change and what that change is?

This magic comes about because of something called the **Observer pattern**.

In many programs there is a need for one object (known as an observer) to be kept informed every time some other object (known as a subject) changes its state. The observer must first register itself with the subject. Then every time the subject changes its state, it sends a message `update()` to the observer (Figure 22).

Observer is an example of a **software pattern**: a tried and tested solution to a programming problem that comes up again and again. In this case, the problem solved is how one object can be kept informed when another object changes state.

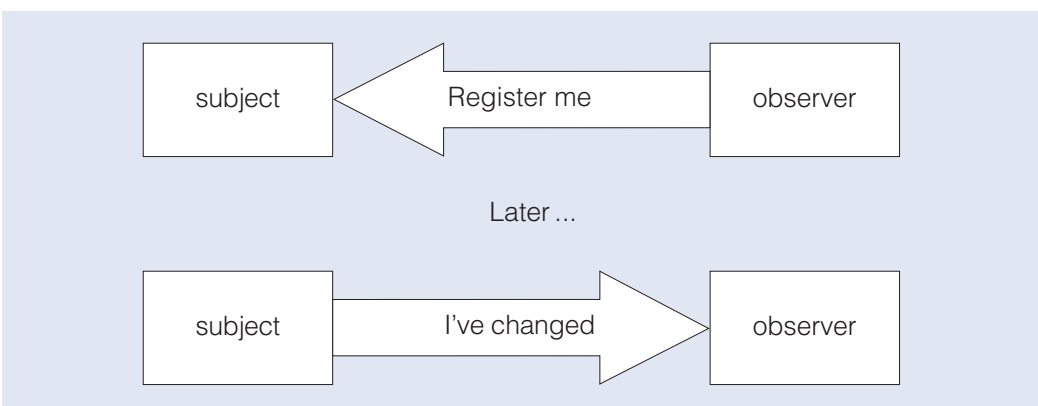


Figure 22 A subject and an observer

The basic scheme is very simple. However there are a number of more subtle points you need to know.

First, more than one observer may observe the same subject. The observers do not know about one another. When a subject issues a message `update()`, it sends the message to all observers registered with it.

Second, all a subject knows about observers is that they are observers – they implement the interface `Observer`, which simply involves implementing a method `update()`. It does not know what class they belong to. This means any observer can be registered with a subject without changing the coding of the subject.

Third, an observer may be registered with lots of different subjects; for example, an instance of `CZWarden` needs to keep watch over many instances of `ChargeableFrog`. Therefore, when an observer object receives a message `update()`, it needs to know

exactly who the sender was. So the message `update()` includes an argument giving a reference back to the subject that sent it.

Fourth, the observer also needs to know what aspect of the subject's state has changed, because not all observers will be interested in all aspects of the subject's state. Therefore the `update()` message also includes a second argument (which can be of any class) that informs the observer what instance variable has changed. In our example this second argument is a string denoting the instance variable, i.e. `"position"`. If an observer is not interested in whether that particular instance variable has changed, it can just ignore the message.

Finally, a subject is able to notify an observer each time it changes, but it cannot normally send the observer any other messages. The observer on the other hand can send the subject whatever messages it likes.

In the case of chargeablefrogs, the exchange between the subject and the observer goes something like the following.

chargeablefrog (subject) to czwarden (observer): *My position has changed.*

czwarden to chargeablefrog: *Tell me your new position (message answer is position)*

If the position is in a charging zone:

czwarden to chargeablefrog: *Tell me your bank account*

(message answer is account)

czwarden to account: *Debit(10) (or 20 as appropriate)*

5.2 Frogs are already `Observable`

Not just any old object can be a subject in the Observer pattern described above. It has to be an instance of a class which is a subclass of a special class called `Observable`. So how can we make instances of `ChargeableFrog` subjects for an instance of `CZWarden`?

All amphibians are observable because the abstract class `Amphibian` is a subclass of `OUAnimatedObject` which in turn is a subclass of `Observable`.

Luckily frogs and all other amphibians are already observable! This is the reason why the graphical display is able to respond to changes in amphibian state. It is an observer, with the amphibians as subjects.

When an amphibian changes state, it sends the graphical display a message `update()`. The graphical display then asks the amphibian for details of its new state.

When the graphical display knows the new details, it updates the icons which represent the amphibians, changing the colour or position (or in the case of hoverfrogs, height). In this way the graphical display is kept in step with the state of the amphibians.

This means that all amphibians, including frogs, already issue the message `update()` whenever their state alters. All we have to do to make a czwarden observe a chargeablefrog is for the czwarden to register itself with the chargeablefrog. When the message `update()` is issued it is sent to *all* observers, so as well as being sent to the graphical display it will go the czwarden.

Let us look at the method `setPosition()` as defined in the class `Amphibian`.

```
public void setPosition (int aPosition)
{
    position = aPosition;
    this.update("position");
}
```

Note that in the method the receiver sends `update()` to itself! This may seem a little strange at first, but let us explain. This message `update()` is not the one that is sent to the observers, it is just the first in the chain of messages that occurs:

- `Amphibian` inherits `update()` from `OUAnimatedObject`, and this method sends the message `notifyObservers()` (with the argument `"position"`) to the receiver.
- `OUAnimatedObject` inherits `notifyObservers()` from `Observable`, and it is this method that sends the message `update(this, "position")` to all the observers.

You do not need to remember the details of this, just that the ability of a czwarden to watch what chargeablefrogs are doing can simply be piggy-backed onto the existing arrangements for keeping the graphical display up to date.

The class `CZWarden` has no instance variables and has only one method which is shown below (the initial comment has been omitted for brevity):

```
public void update(Observable subject, Object changedInstVar)
{
    if (changedInstVar.equals("position"))
    {
        ChargeableFrog cf = (ChargeableFrog) subject;
        int pos = cf.getPosition();
        if (((pos >= 5) && (pos <= 15)) || ((pos >= 25) && (pos <= 35))) // outer zone
        {
            cf.getAccount().debit(10);
        }
        if ((pos >= 16) && (pos <= 24)) // inner zone
        {
            cf.getAccount().debit(20);
        }
    }
}
```

Notice that the first thing the method does is check whether the changed instance variable is one that it is interested in, i.e. `position`. If it is, then the new position is checked to see if it falls into one of the two charging zones, and, if it does, one of the two `if` statements is executed and the subject's bank account is appropriately debited. If it does not, no money is debited.

5.3 Testing the classes `ChargeableFrog`, `CZWarden` and `Account`

All the classes have been coded and your task is to test the interaction between chargeablefrogs, czwardens and accounts (integration testing). To save time we have already created unit tests for `Account`. Of course they will be run along with the tests you create, because the policy is to do regression testing every time, retesting all the old classes as well as testing the new.

The only class you need to write a new test class for is `ChargeableFrog`. The reason for this is that instances of `CZWarden` have no state – they have no instances variables. However this does not mean that the class `CZWarden` will not be tested, because testing that messages `setPosition()` appropriately debit a chargeablefrog's bank account will of course test the logic of the only instance method in `CZWarden` which is `update()`.

ACTIVITY 11

Open Unit13_Project_12. You will find it contains the amphibian hierarchy, including the class `ChargeableFrog` and the classes `CZWarden` and `Account`. Begin by making sure all the classes are compiled.

1 `Account` already has a test class `AccountTest`. First right-click on the `AccountTest` icon and choose Test All to confirm that `Account` passes all four of its unit tests.

2 Now create a test class for `ChargeableFrog`. We are going to provide this test class with a test fixture.

Right-click on the `CZWarden` icon and select New CZWarden. Accept the default name cZWarden1 and press Ok. A `CZWarden` object will appear on the Object Bench.

Next right-click on the `ChargeableFrog` icon and choose new ChargeableFrog (CZWarden czw). In the argument field enter cZWarden1 and press Ok. The instance of `ChargeableFrog` will be displayed on the Object Bench.

Finally right-click on the test class `ChargeableFrogTest` and choose Object Bench to Test Fixture.

3 Now you are nearly ready to test that the overall program behaves as specified. You need to move the `ChargeableFrog` object created in the fixture to the positions in the test plan in Table 12 and verify that, where appropriate, the `CZWarden` object debits its bank balance is as expected.

Right-click on `ChargeableFrogTest` and select Create Test Method.... Name the test testSetPosition. You will enter the tests manually into the method `testSetPosition()`, not record them, so press End to stop recording.

The balance of an account is a `float` value and there is an important fact you need to know about floating-point numbers. Without going into technical details, when two numbers of type `float` or `double` are compared, they are not exact numbers in the way integers are. You may have sometimes noticed that on your calculator you can end up with say 2.999999999, when the answer is really 3. This is the same effect.

So when we compare two `float` values we have to specify how near they must be to be considered equal. This tolerance is called **delta**. Since we are dealing with money, within 0.01 seems appropriate.

When we assert that two floating-point values are equal, the statement takes the form

```
assertEquals(expected, actual, delta);
```

So, for example, to check the first test case in Table 12 we move the chargeablefrog to stone 2, then make an assertion about its bank balance (which should not have decreased).

```
chargeab1.setPosition(2);
assertEquals(250, chargeab1.getAccount().getBalance(), 0.01);
```

(Notice the collaboration here; we first ask `chargeab1` for its account and then ask the account for its balance.)

4 Enter the 17 tests from Table 12. If you like you can run the tests from time to time as you go along, to make sure that you have not made any typing errors, but remember these are only partial results; we want the program to pass all 17 tests.

5 When all the tests have been entered, press Run Tests. If all goes well you will be rewarded with the following window (Figure 23). The classes have passed all the tests!

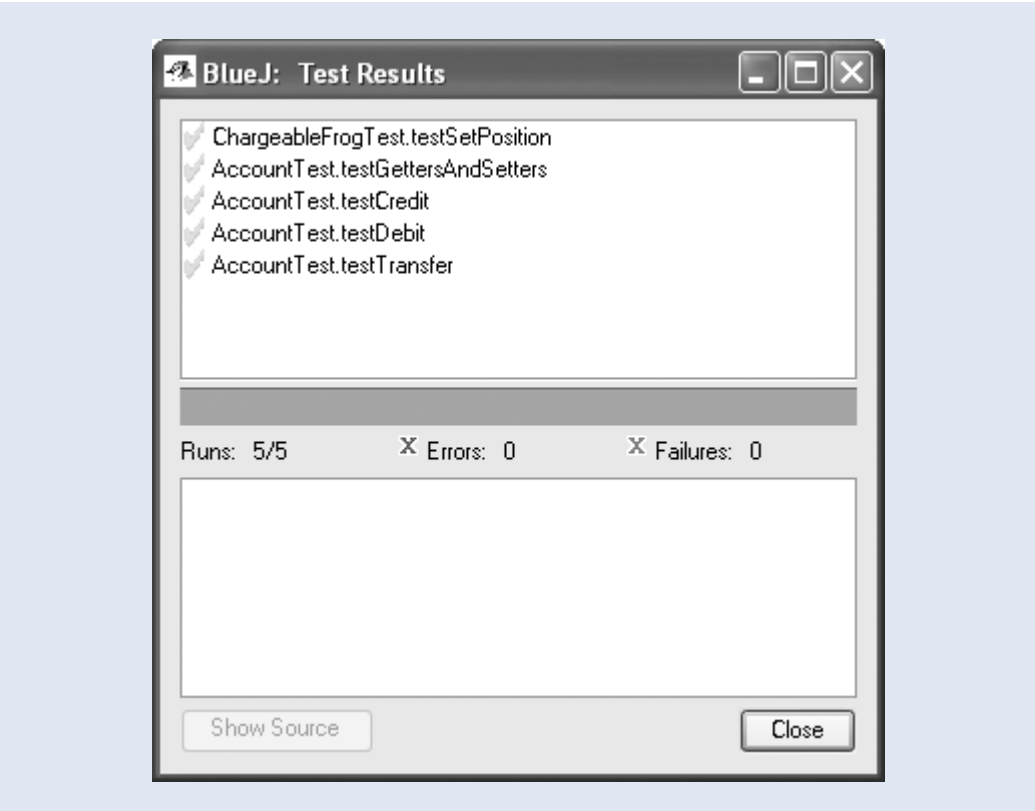


Figure 23 The integration test is passed

In case you had any problems with this activity, the complete tests are available in Unit13_Project_12_Completed.

6 Summary

After studying this unit you should understand the following ideas.

- The purpose of testing is to find the faults in software and an effective test is one which discovers faults other tests have missed.
- White box tests take the internal workings of the code into consideration and try to ensure that every part of the code is executed. Black box tests are designed purely from the specification which the software is intended to meet, without any knowledge of how the specification is actually implemented. In M255 we only consider black box testing.
- The input–output model of testing compares actual results with expected ones. If the actual results match the expected ones, the software passes the test, otherwise it fails.
- A unit test is the smallest possible test – a test of a single method or class. A unit test is an 'atom' of testing.
- Integration testing is when a number of components developed in isolation are brought together and tested as a combination.
- Regression testing is testing an entire program each time any part of it is changed, in case the changes have caused other parts of the program to stop working correctly.
- Test-driven development is an approach in which software is developed in small steps called increments, with the tests for each increment being written before the code.
- Running tests manually is not practical, because regression testing may easily involve hundreds or thousands of classes and methods, and test automation becomes essential. JUnit is a framework which lets us write tests in a standard format and takes care of the practicalities of managing and running tests for us.
- Using JUnit involves writing test classes and test methods. Most Java IDEs provide some support for JUnit, by automating the writing of outline classes and methods, but we must supply the bodies of the test methods, since only we can predict what the result of a given test should be.
- An assertion is a statement which compares an expected value with an actual one. The assertions we use in this unit are all of the form
  `assertEqual(<expectedValue>, <actualValue>).`
- A test fixture is a set of test objects which will be created automatically before each test method is run. Using a test fixture avoids duplicating object-creation code, since the test objects are defined only once, in a special `setUp()` method.
- Using JUnit, suites of tests are built up and regression testing is done by running them all every time the code is changed in any way. If the code fails any of the tests we correct the fault before going any further.
- A boundary error is when faulty code causes an input to take an incorrect path through the program.
- A computation error is when the correct path is taken but faulty code results in the input being processed incorrectly.

- A general strategy for choosing a small number of test cases that are likely to be effective at detecting errors is to:

  identify the boundary cases, at which boundary errors are likely to occur, and test on the boundary and close to it on either side;

  choose a suitable representative of each of the equivalence classes into which the inputs are split by the boundaries, in order to detect any computation error affecting that region.

- The Observer pattern is a general solution to the problem of how to keep one object (or set of objects), called the observer, updated whenever another object, called the subject, changes its state. The Observer pattern underlies the ability of the graphical display to respond to changes in the state of amphibians.

LEARNING OUTCOMES

Having studied this unit (including the associated practical work) you should be able to:

- explain the purpose of software testing;
- explain the difference between black box and white box testing;
- explain the meaning of the terms unit testing, integration testing and regression testing;
- understand the input–output model of testing, including the fact that tests must take into account the state of objects, since behaviour is often state dependent;
- explain the test-first approach to development;
- given a method or class specification design a suitable set of test cases for it;
- use JUnit to set up and run unit tests and regression tests;
- use JUnit to set up and run a simple integration test;
- apply the technique of equivalence-class and boundary testing to the design of test plans;
- appreciate the way in which the Observer pattern is used to keep objects updated when other objects they have an interest in change state.

Glossary

acceptance testing Testing that software fulfils a customer's requirements.

actual results Given a set of test data, the result of executing a method or program.

alpha testing Testing which is done before a product is available publicly.

assertion A statement which tests whether an expected value matches an actual one.

beta testing Testing of a trial version of a product by potential users, who provide feedback to the developers.

black box testing Testing something against specification, without any knowledge of its inner workings.

boundary error A programming error which causes an incorrect path to be taken through code.

computation error A programming error which causes data to be processed incorrectly; for example, the wrong calculation may be performed on the data.

debugging Identifying the cause of faults in software and correcting them.

delta A tolerance used for comparing floating-point numbers, which are not represented exactly in the computer, so that two values which should be equal may differ marginally. We set a small delta and then consider that two floating-point numbers are the same if they differ by less than the delta.

equivalence class All the possible input values to a block of code (for example, a method) can be divided into mutually exclusive sets of values. Each set of values is called an equivalence class. Each value in an equivalence class can be expected to be processed by that block of code in the same manner. The corollary is that values from different equivalence classes will be treated in a different manner (i.e. be processed by a different branch of the code).

expected results Given a set of test data, results that a method or program should produce according to its specification.

integration testing Testing that a group of components, already tested in isolation from one another, work together in the way intended.

Observer pattern A software pattern in which one object (the observer) is automatically kept informed whenever another object (the subject) changes state.

regression testing Testing that modifications or additions to one part of the code have not made any other part stop working properly.

software pattern A tried and tested solution to a programming problem that comes up again and again, presented as a pattern of classes and collaborations.

system testing Testing that the overall system functions correctly.

test data A set of data used as the inputs to a test.

test-driven development An approach to software development in which code is written in increments, with the tests for each increment being written before the code.

test-first A software development philosophy in which tests are written first and the corresponding code follows.

test harness A programme written solely to test other software components.

testing The process of uncovering faults in software.

testing framework An API which provides support for automated software testing, allowing tests to be created in a standardised way and reused.

unit testing A test of a single method.

white box testing Testing which is guided by information about internal structure and which aims to ensure that every part of that structure has been tried.

Reference

Dijkstra, E.W. (1972) 'Notes on structured programming' in Dahl, O.-J., Dijkstra, E.W. and Hoare, C.A.R. (eds), *Structured Programming*, London, Academic Press.

Index

A
acceptance testing 14
actual results 12
alpha testing 13
assertion 16

B
beta testing. 13
black box testing 6
boundary error 47

C
computation error 47

D
debugging 6
delta 58

E
equivalence class 48
expected results 12

I
increment 14
integration testing 14, 57

O
Object Bench 21
Observer pattern 55

R
regression testing 14

S
system testing 14

T
test data 12
test fixture 30
test harness 14
test-driven development (TDD) 14
test-first approach 14
testing 6
  framework 15

U
unit testing 5, 13

W
white box testing 7