Self-Study Module: Test Case Design

This module introduces the technique of functional (blackbox) unit testing to verify the correctness of classes. It shows how to design unit test cases based on a class specification within a contract programming approach. The laboratory exercises then guide students through creating and running tester classes in Java from a test case design, utilising the JUnit unit test framework. It also contains a worked example on how to unit test GUI and event-handling classes.

Learning objectives:

- To develop skill in designing functional unit tests based on pre/post conditions within a contract programming approach.
- To develop skill in constructing tester classes with the JUnit unit test framework.

References:

- John McGregor and David Sykes, "A Practical Guide to Testing Object-Oriented Software", Addison-Wesley, 2001

  Chapters 4 and 5
- BlueJ Unit testing tutorial (downloadable from www.bluej.org)

Source code:

- Wu Chapter 9, Address Book program and Exercise 17, p.478
- *JavaCollections* BlueJ Project (Downloadable from UTSOnline OOD folder Course Documents/ Self-Study Modules)

Contents:

Exercise 1 – Designing unit test cases 2
Exercise 2 – Unit Testing with JUnit in BlueJ...5
Exercise 3 - Worked example: How to unit test GUI and event-handling classes 8

Self-Study Module: Test Case Design

Exercise 1 – Designing unit test cases

This exercise involves the design of unit test cases based on pre/post conditions within a contract programming approach.
A design specification has been provided for an extension to the Address Book program.
It is your task to design the test cases for the getNextPerson method based on pre/post conditions and using functional testing techniques.

An example of test cases has been provided for the getFirstPerson method in the form of a test sequence table. Read through the example first, and then work on writing test cases for the getNextPerson method. Generate a table of test cases in the same format as that for getFirstPerson.
Tip: Do you need to test for the end of the array?

The original Address Book program has been extended so that the programmer can access all Person objects in the Address Book. Two methods have been added to the AddressBook class, getFirstPerson and getNextPerson, as defined in the table below. It is intended that these two methods be used to access the Person objects in the Address Book one by one, by first calling getFirstPerson and then repeatedly calling getNextPerson until no more Person objects are returned. An integer attribute nextPersonIndex belongs to the AddressBook class. It is initialised to -1 in the constructor for the AddressBook class.

Design Specification for the new methods of the Address Book class

Method	getFirstPerson
Purpose	To return the first person in the Address Book
Input	none
Output	`Person` object
Precondition	none
Postcondition	The first person in the Address Book has been returned or null has been returned if the Address Book was empty.

Assume that for correct operation, getFirstPerson has been called as it sets nextPersonIndex to valid values.

Method	getNextPerson
Purpose	To return the next person in the Address Book
Input	none
Output	`Person` object
Precondition	nextPersonIndex >= 0
Postcondition	The next person in the Address Book has been returned or null has been returned if there were no more persons in the Address Book.

Let's assume the class invariant insists that the Person objects are stored contiguously from the first element in the Address Book. In other words, there are no empty references between Person objects.

Test Cases for the getFirstPerson method

The test cases have been generated from an examination of the pre/post conditions describing the expected behaviour of the operation and the concept of equivalence sets. The precondition states that the operation will work correctly for any address book, be it empty or not. The postcondition indicates what is a valid value to return. For this operation, either a Person object or a null reference is a valid return value.

The input equivalence sets have been identified as:

empty address book (i.e. no Person objects)

address book contains a single Person object

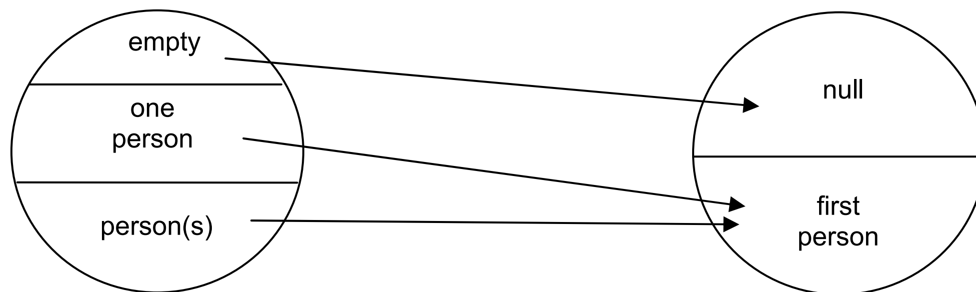
address book contains more than one Person object

The output equivalence sets have been identified as:

null

Person object

The mapping between input and output sets is



We can identify the normal, boundary and illegal test cases from the equivalence sets.

Normal	Boundary	Illegal
> 1 person	0 or 1 person	-

In the table below, the test cases have been categorised with Wu's terminology of normal, end or error cases.

Test Sequence Table

Test Sequence	Expected Results	Purpose
Call getFirstPerson before any Person objects are added to the Address Book.	Null is returned.	End case (empty address book). Verifying that null is returned if the Address Book is empty.
Add a single Person to the Address Book.		End case (one person exists in the address book). Verifying that the first person added to the Address Book is retrieved with the call to getFirstPerson.
Call getFirstPerson		
Display the Person data for both the added object and the returned object.	Output indicates matching data.	
Add a second Person with different data to the Address Book.		Normal case (more than one person exists in the address book). Verifying that the first person in the Address Book is always retrieved.
Call getFirstPerson		
Display the Person data for both the newly added object and the returned object.	Output indicates that the returned object data matches the first object added.	

Ref: Wu chapter 9, section 9.6, p. 461, Step 4 Test Data

Self-Study Module: Test Case Design

Exercise 2 – Unit Testing with JUnit in BlueJ

This lab continues the work done on designing test cases in the previous exercise. The type of testing we are examining here is class testing (unit testing), where we are concerned with proving the correctness of the class given its functional specification. We will focus on functional testing techniques in this exercise.
Once you have designed a set of test cases, then you need to decide the best way to run the tests and collect the resultant data.

There are various approaches to perform functional unit testing and we will mention two here, and work through one of them in detail. One approach is to interact directly with the object under test via the BlueJ interface. The advantage of this approach is that you can utilise BlueJ's object inspection mechanism to view attribute data (see Lab on Debugging). Another approach is to create a separate Tester class that creates an object of the class under test and calls its various methods. We will be following the second approach. It has advantages of automation and repetition; however, it cannot use BlueJ's object inspection mechanism. In the task of this laboratory you are required to use JUnit in BlueJ. It increases the level of automation of testing which facilitates regression testing.

Task. Creating a Tester class using JUnit in BlueJ
In this laboratory you are going to test the solution to the laboratory exercise *JavaCollections* that deals with collections in Java (it is a good idea to review the lab exercise now – Exercise 3 in the Self-Study Module *Java and BlueJ Refresher*). This project includes the class Lab3Tester that was manually written and tests specific methods of the AddressBook class based on the requirements of the exercise.

Your task consists of creating a similar testing class using JUnit and the tools available in BlueJ for this purpose. The first thing you need to do is to study the BlueJ Unit Testing tutorial which will give you a good idea of how JUnit has been introduced in BlueJ. Secondly, review the code of the class Lab3Tester and make a list of the steps that are made in order to test the AddressBook class.

In the Lab3Tester class...

- which objects are created?
- what methods are called and in which order?
- what information is displayed in every test?

Suggested steps and hints:

- If the unit testing tools are not visible yet in your BlueJ environment, go to "Tools-> Preferences…", click on the tab "Miscellaneous" and check the "Show unit testing tools" option.
- Create a testing class for the AddressBook class.
- Create the AddressBook and Employee objects involved in the testing using the BlueJ environment (using constructors listed after right-clicking on a class). Look at the Lab3Tester class to get all the details of the objects and their data.
- Create the boss-employee relationships as indicated in the Lab3Tester class but using the BlueJ environment – not copying the code!
- Add the Employee objects to the AddressBook object.

Your project should look similar to Figure 1.

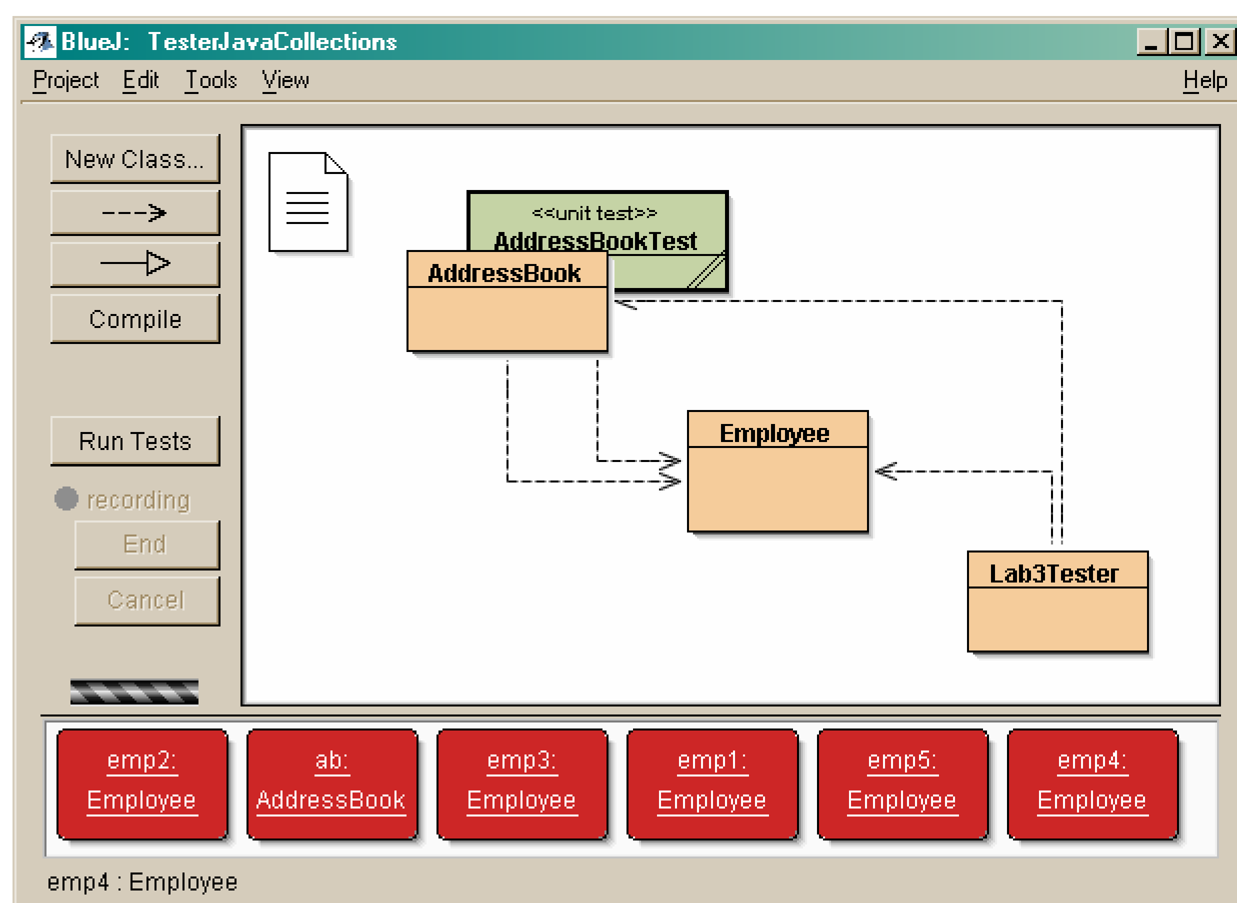


Figure 1. After creating objects

- Use the *Object Bench to Test Fixture* tool to update the testing class with the objects you have created.
- Add a new test method called TestOne corresponding to TEST 1 of the Lab3Tester class. Use the *Recording* tool of BlueJ. Check the *Assert that* checkbox since the first result is the correct one.
- Repeat the same process for the rest of the tests defined in the Lab3Tester class.

Now your project should look similar to Figure 2.

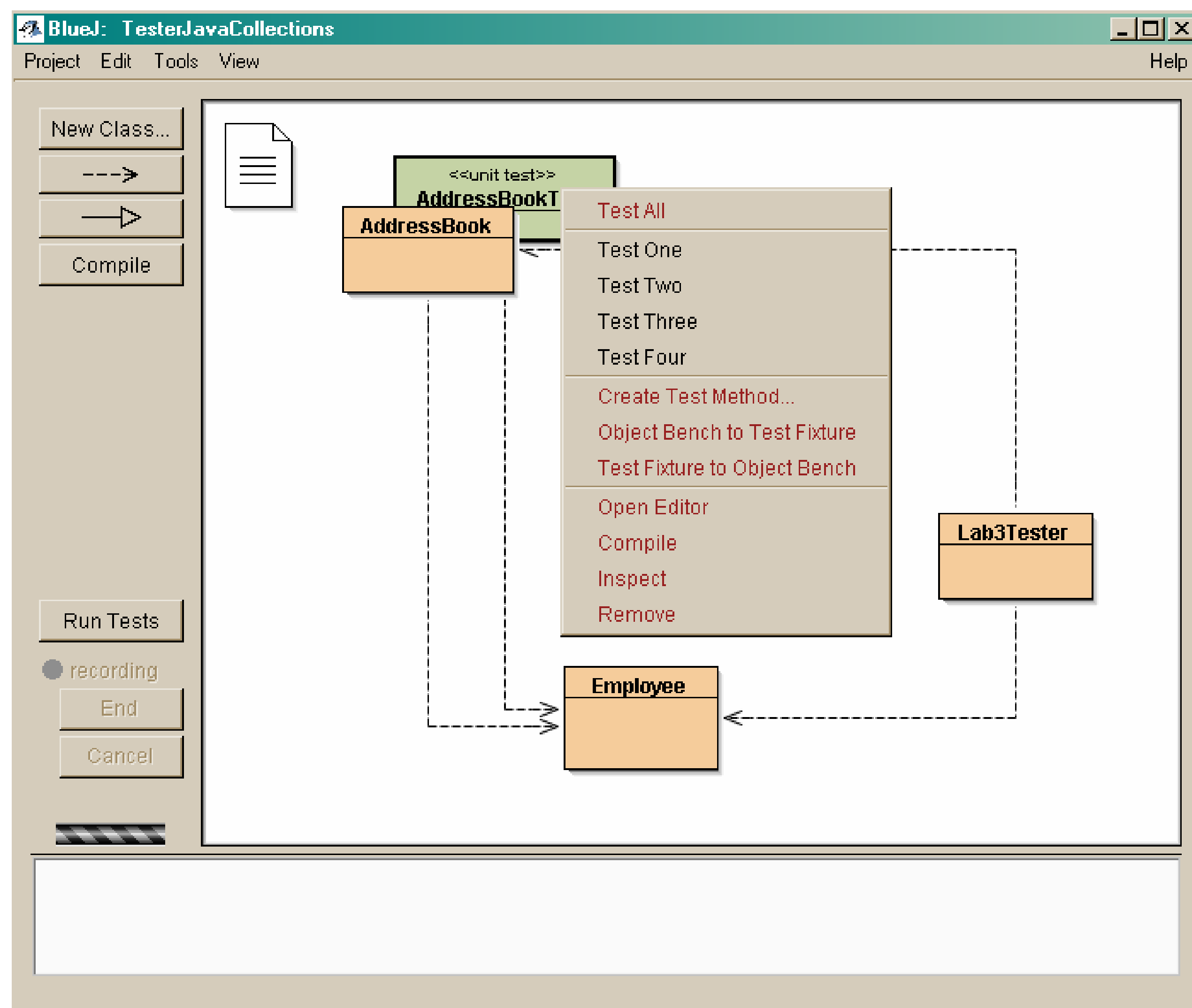


Figure 2. After adding testing methods

- Click the *Run Tests* button. What is the result of the tests?
- Modify any of the methods tested so the returned value is different to the one expected, e.g. in the method `addInChargeName` change return value from `false` to `true` when the error occurs (line 189). Then run TestOne, what's the output of test? Do you get more information about what happened?

If at a certain moment you do a modification to any of the classes of the system, you can then run the tests you just created to verify that the system is still working as desired without writing the test methods again. Play a bit more with this helpful tool!

Review Questions

1. Did you need to or want to make any modifications to the `AddressBook` class itself in order to properly test the class?
2. Would testing of these methods have been easier using the BlueJ interface directly?

Exercise 3 - Worked example: How to unit test GUI and event-handling classes

Unit testing GUI and event-handling classes requires a different approach to designing test cases and running unit tests. It is difficult to test a GUI class directly by calling its public methods, as often the public methods are event listener methods such as actionPerformed( ). What we need to do is generate the events that trigger the listener methods. The easiest way to do this is by instantiating the GUI class and interacting with the event-generating components, such as buttons and textfields.

Take the MySketchPad class as an example (Wu Chapter 13). This class allows the user to draw lines on a canvas by clicking and dragging the mouse.

Where to begin?

Hopefully with a class specification!
If this does not exist, then one better be written.
The class specification should look something like this, written in UML syntax …

Class SketchPad
Purpose: This class allows the user to draw lines on a canvas by clicking and dragging the mouse. It stores the x and y coordinates of the last mouse drag point. These are then used when drawing the line between the current mouse coordinates and the last.

General user interaction:
Right-button click: erase the contents of the screen
Left-button hold-and-drag used to draw lines.

The input for this class is in the form of user-initiated mouse events.
The output is visible changes to the graphic content of the screen, in this case, lines being drawn or the screen being erased.

Attributes:

last_x: Integer	represents x coordinate of last drag point
last_y: Integer	represents y coordinate of last drag point

Operations:
mousePressed(me: MouseEvent)
Input: me: MouseEvent
Output:
Precondition: A mouse event of type `MOUSE_PRESSED` has occurred.
Postcondition: If the right mouse buton has been pressed, then the graphic screen has been cleared. If the left mouse button has been pressed, then the current coordinates of the mouse cursor have been stored in self.last_x and self.last_y (here, self refers to the instance of the class that we are currently in).

Next the mouse motion listener method…

mouseDragged(me: MouseEvent)
Input: me: MouseEvent
Output: A line has been drawn from old coordinates to new coordinates.
Precondition: A mouse event of type `MOUSE_DRAGGED` has occurred.
Postcondition: A line has been drawn on the graphic canvas between the old coordinates and the new coordinates. The current coordinates of the mouse cursor have been stored in self.last_x and self.last_y (here, self refers to the instance of the class that we are currently in).

A less ambiguous way to express this postcondition would be to use the OCL (Object Constraint Language) @pre operator. It allows you to refer to the value of an attribute or variable just before the operation executes. So...rewriting the postcondition:

A line is drawn between (last_x@pre, last_y@pre) and (current_x, current_y).
last_x = current_x
last_y = current_y

Testing this class

Let's examine the class specification. We need to create test cases based on the specification for each operation. Here we are performing functional (blackbox) testing.

Test cases for operation mousePressed

First, the pre and post conditions…
This operation changes the object state by modifying the attributes, x and y. Sometimes operations change the object state of other objects that it uses. In this case, the operation just changes its own object state.

We should ask, what are the valid values for the x and y coordinates? Is there a range, or is any integer value acceptable? We could think of the mouse coordinates generated by the physical mouse relative to our virtual screen as having a valid range defined by the dimensions of the frame. For example, x: [0..100] and y: [0..200]. That is, x can take any integer value from 0 to 100 inclusive. The values for x and y form the input set.

What can we select as representative test cases? What data values are relevant? We don't want to test the operation with every possible input value or variation in object state. That could take days or years! One strategy is to think in terms of *equivalent* sets or classes of data. For each equivalent set, we normally take a test case for a value in the middle. Then test cases for values on and around the boundary of the set.

So…for the mousePressed operation, what are the valid values for x and y? Can we decompose this set into equivalent sets, or can we simply have a single test case? What are the boundary values? The boundary is given by the boundary of the screen. Do we have to test for negative values? This depends on preconditions and possible input values. For this class, it is not possible to have negative (in this case, illegal) values of the x and y coordinates of the mouse relative to the sketchpad window.

Let's go with the following set of test cases, combined into a test sequence for both operations for ease of testing:
Assume that the drawing screen has dimensions as given above, width 100 and height 200.
Assume that we start with no mouse events and a clear drawing screen.

Test case	Event	Input	Expected Result	Description
1	left mouse press (do not release)	x: 10, y: 15	last_x = 10, last_y = 15	Normal case mousePressed
2	left mouse drag (do not release)	x: 80, y: 50	last_x = 80, last_y = 50 line drawn from (10,15) to (80,50)	Normal case mouseDragged
3	left mouse drag	x: 100, y: 0	last_x = 100, last_y = 0 line drawn from (80,50) to (100,0)	Boundary case mouseDragged
4	left mouse drag (then release)	x: 99, y: 199	last_x = 99, last_y = 199 line drawn from (100,0) to (99,199)	Boundary case mouseDragged (close to boundary)

Is this sufficient? We probably need to test further, near the boundary, on all sides.
What about the right-click mouse event?

5	right mouse click		screen is erased	Normal case mousePressed

Running the tests

It is difficult to write a tester class for this class, as the input events are initiated by the user. So, we simply run the class, and interact with it according to the test cases. One issue with testing this way, is that obtaining the exact input values as prescribed in the test cases could be difficult. Does it matter? Perhaps we could have provided a range of possible values to select from, if the exact value was not significant.

If we could write a tester class, then here is a skeleton for it, outlining the kinds of things that the class should do.

```
/*
 * Program MySketchPadTester
 *
 * The tester class for testing MySketchPad
 */

class MySketchPadTester
{
   public static void main (String arg[])
   {
      // Create the Object Under Test (OUT)
            MySketchPad sketchpad = new MySketchPad( );
      sketchpad.setVisible(true);

      // Setup the state of the OUT

      // Setup any other objects that are required to test OUT
      // depending on operation preconditions and class
invariant

      // Call testmethod(s) - one per test case, if possible

      // Verify test results
      // 1. By internal code comparison:
       - may need to check postconditions and invariant
       - may need to calculate comparative results
       - may need to retrieve data from OUT or other objects

      // 2. By human inspection
       - of sound or visible output
       - of file contents

   }
}
```

Issues

- **What if the public interface of the class provides no access to the current state and/or attributes of the object of the class under test?**

Should you add public accessor methods?
One line of thinking is that outside classes (be they tester or otherwise) should be able to *observe* the current state of the object. Thus one should include appropriate accessor methods to allow other outside objects to query or observe the object's current state. This does not mean that you have to provide corresponding mutator methods, that allow outside objects to modify the attributes or state! The current state can be returned in a non-mutable form, such as a String or primitive data types. You may have, for example, an attribute that is an array of Button objects. You may not wish outsiders to be able to manipulate this array or change any aspects of the Button objects. However for the purposes of observing object state, you could supply a method that converts the necessary information into a String (or array of Strings).

You can design classes to support testing needs, by adding methods to query the state of the object.

Here is a tip taken from McGregor and Sykes (p.29),

> Review pre- and postconditions and invariants for testability during design. Are the constraints clearly stated? Does the specification include means by which to check preconditions?

- **Temporary test code**

You may need to include code in your classes that is only for testing purposes. Often this is in the form of printing information to standard output. These lines of code should be clearly commented as such.
For example,

```
System.out.println("Inside SketchPad.mousePressed...\n");
System.out.println("Last dragged x: " + last_x + ", y: " +
last_y + "\n"); //for testing only
```

You may choose to leave them in, if they do not interfere with the desired output of your program. Or you may delete them (with care) if you need to have production standard code.

- **How to test private methods? Do you need to make them public temporarily?**

Testing of private methods falls under structural testing. With structural testing, you derive test cases based on the internal control structure of the code. Traditional metrics of path, branch and loop coverage can be used to ensure that all lines of code have been executed under various conditions.
Since we are concerned with black box or functional testing, we will leave aside structural or whitebox testing for the time being. Of course, to fully test your code to production standards, you would need to perform both, plus other kinds of testing.

- **Using automated tester classes versus manual test execution and inspection?**

Automated tester classes are great when you do not need human confirmation of the inputs and outputs of a test case. They have the advantage of being repeatable, and thus greatly aid regression testing. If you can, then write tester classes that reflect a sequence of test cases. Tester classes enable programmed setup of object state, execution of test methods, and confirmation of test results.

When human intervention is required, particularly when audible and visible outputs are produced, or data is written to a file, then automated tester classes are inadequate or inappropriate. Of course, you may test a class using both techniques. The tester class can be commented to indicate when human intervention is required.

Then manual test execution and inspection is required. It is also needed when the class under test provides insufficient access to its object state in order to confirm correct test case results. In this situation, the BlueJ object inspection tool is very useful. You may need to control the testing by running the code in the Debugger, setting breakpoints when you need to inspect an object that may not be accessible from the outside.