Unit Testing in BlueJ

Version 1.0
for BlueJ Version 1.3.0

Michael Kölling
Mærsk Institute
University of Southern Denmark

Copyright © M. Kölling

1 INTRODUCTION 3

2 ENABLING UNIT TESTING FUNCTIONALITY 5

3 CREATING TEST CLASSES 6

4 CREATING TEST METHODS 8

5 RUNNING TESTS 10

6 INTERPRETING TEST RESULTS 11

7 WHAT IS A FIXTURE? 12

8 CREATING AND USING TEST FIXTURES 13

9 WRITING TEST METHODS BY HAND 14

10 WRITING TESTS FIRST 15

11 MULTI-CLASS TESTING 16

12 JUST THE SUMMARIES 17

Copyright © M. Kölling

1 Introduction

Summary: BlueJ provides regression testing functionality by integrating JUnit.

1.1 About this tutorial – scope and audience

This tutorial introduces the unit testing functionality in the BlueJ environment. We assume that you are already familiar with BlueJ's general functionality. If not, read 'The BlueJ Tutorial' first. (You can get that tutorial, and an electronic version of this one, at http://www.bluej.org/doc/documentation.html).

We also assume that you are somewhat familiar with the idea of unit testing (or at least software testing in general). We give a few pointers in the next section.

1.2 What is unit testing?

The term *unit testing* refers to the individual testing of separate units of a software system. In object-oriented systems, these units typically are classes and methods. Thus, in our context, unit testing refers to the individual testing of methods and classes in BlueJ.

This tutorial discusses BlueJ's tools for *systematic* unit testing. If you are familiar with BlueJ's interaction features, then you know that it is easy in BlueJ to test individual methods interactively. We refer to this as *ad-hoc testing*.

Ad-hoc testing is useful, but not good enough for systematic testing. The unit testing features in BlueJ give you the tools to record and replay tests, so that unit tests can easily be repeated later (typically after a change in the system), so that the developer can gain some confidence that recent changes have not broken existing functionality. This is known as *regression testing*.

The concepts of unit testing and regression testing are old, but their popularity was greatly increased recently with the publication of the eXtreme programming methodology[1] and a unit testing tool for Java, *JUnit*.

JUnit is a regression testing framework written by Erich Gamma and Kent Beck. You can find the software and a lot of information about it at *http://www.junit.org* .

---

[1] To find out what eXtreme programming is, have a look, for example, at *"Extreme Programming Explained: Embrace Change", Kent Beck, Addison Wesley, 1999*. There are many other books available. A good online summary is at http://www.xprogramming.com/xpmag/whatisxp.htm

Copyright © M. Kölling

1.3 Unit testing in BlueJ

The systematic testing tools in BlueJ are based on JUnit. Thus, some knowledge about using JUnit helps in understanding unit testing in BlueJ. We recommend reading an article about this (maybe not right now, but some time later). Many such articles exist, and the JUnit web site is a good starting point to find them.

Unit testing in BlueJ combines BlueJ's interactive testing functionality with the regression testing of JUnit. Both testing methods are fully supported. Additionally, new functionality resulting from the combination of the two systems is available: interactive test sequences can be recorded, for example, to automatically create JUnit test methods for later regression testing. Examples are given later in this document.

The unit testing functionality in BlueJ was designed and implemented by Andrew Patterson (Monash University) as part of his PhD work.

Copyright © M. Kölling

2 Enabling unit testing functionality

Summary: Testing tools can be made visible with a switch in the preferences.

The explicit testing support in BlueJ is initially disabled. To use the testing tools select Tools – Preferences... and select the checkbox labelled Show testing tools.

Enabling this functionality adds three elements to the interface: some buttons and a "recording" indicator in the main window's tool bar, a Show Test Results item in the View menu, and a Create Test Class item in the popup menu of compiled classes.

Copyright © M. Kölling

3 Creating test classes

Summary: Create a test class by selecting Create Test Class *from the class popup menu.*

The first step to setting up testing of a class or method in BlueJ is to create a test class.

A test class is a class associated with a project class (which we will call the *reference class*). The test class contains tests for methods for the reference class.

For the examples in this tutorial, we use the *people* project, which is distributed with BlueJ as one of the examples in the *examples* directory. You may like to open it on your system to try out things as you read this.

You can create a test class by right-clicking (MacOS: control-clicking) a compiled class and selecting the Create Test Class item from the popup menu. The test class is automatically named by adding a "Test" suffix to the name of the reference class. For example, if the reference class name is "Student", the test class will be named "StudentTest".

Test classes are shown in the diagram marked with a <<unit test>> tag attached to the reference class. They also have a different colour (Figure 1). Dragging the reference class will keep the test class attached.

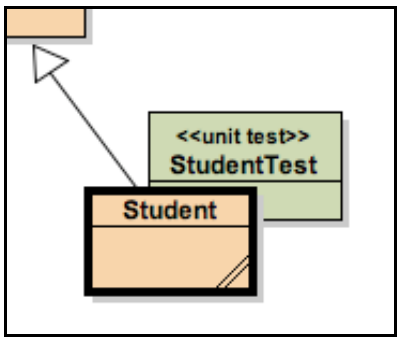

Figure 1: A reference class with an associated test class

Test classes are treated in a specialised way by the environment. They have the usual class functions (such as Open Editor, Compile, Remove), but also some test specific functions (Figure 2). The test class must be compiled to see this menu.

Copyright © M. Kölling

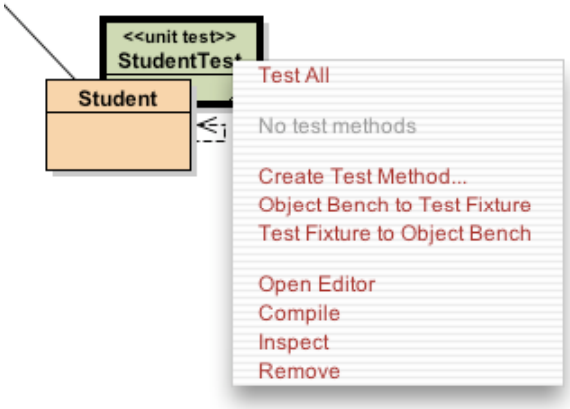

Figure 2: The popup menu of a test class

Creating the test class in itself does not create any tests, but it gives us the option of creating tests now. The test class is used to hold the test we will create.

Copyright © M. Kölling

4 Creating test methods

Summary: Create a test method by selecting Create Test Method... *from the test class's menu.*

Student objects have two methods, `setName` and `getName` (inherited from Person), to set and retrieve the name of the student. Assume we want to create a test to check that these methods work as expected.

We start by selecting Create Test Method... from the `StudentTest` class. A *test method* implements a single test (that is: the testing of one bit of functionality).

After selecting this function, you will be prompted for a name for this test. Test names always start with the prefix "test" - if your chosen name does not start with "test" this will be automatically added. Thus, typing "testName" or "name" will both result in creating a test method called "testName".

After typing the name and clicking OK, all interaction will be recorded as part of this test. The 'recording' indicator is on, and the buttons to end or cancel this test recording are enabled (Figure 3).

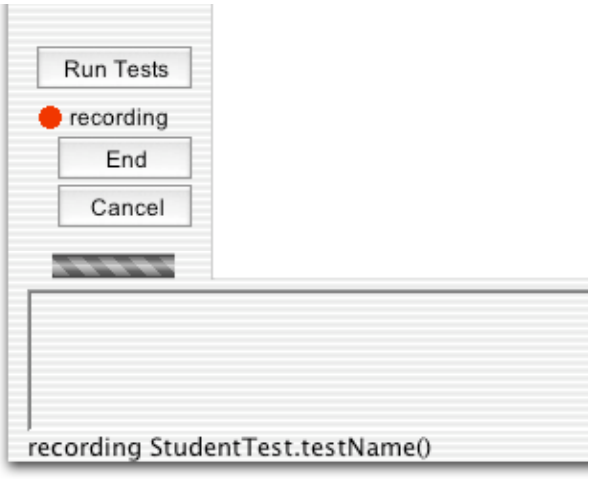

Figure 3: Test buttons during test recording

To record the test in this example, do the following:

- Create a Student object, using the constructor without parameters.
- Call the `setName(newName)` method (inherited from Person) and set the name to "Fred".
- Call the `getName()` method.

After calling the `getName` method, you will see the result dialog. While we are recording tests, the result dialog includes a part that lets us specify assertions on the result (Figure 4). We can use this assertion facility to specify the expected outcome of the test. In our case, we expect the result of the method call to be equal to the string "Fred", so we can specify this as an assertion (Figure 4).

Copyright © M. Kölling

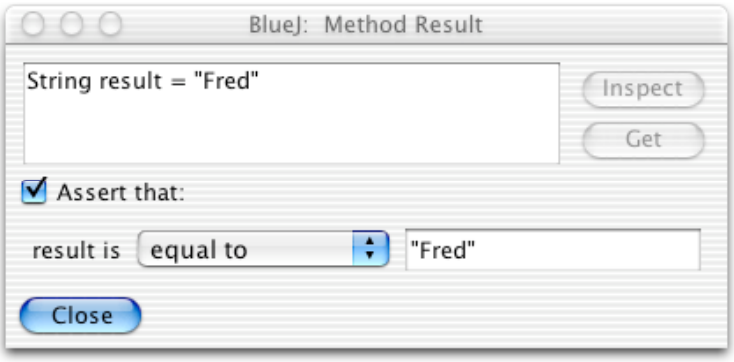

Figure 4: A result dialog with assertion options

Several different kinds of assertions are available from the popup menu, including tests for equality, null, and not null.

This concludes our test case, so we can now click 'End' under the test recording indicator to end the recording of the test.

Ending the test results in a test method being added to the test class. This test method is then available for running.

We can use the recording 'Cancel' button to end the recording and discarding it.

In a similar way to this example, we can record any number of tests. Each class in the project can have its own test class, and each test class can have any number of tests.

Each test recording can consist of an arbitrary number of actions, including arbitrary creation of instances and any number of assertions.

Copyright © M. Kölling

5 Running tests

Summary: Run all tests by clicking the Run Tests *button. Run individual tests by selecting them from the test class's menu.*

Once tests have been recorded, they can be executed. Test classes are Java classes just like the reference classes, so they too must be compiled before execution.

BlueJ automatically attempts to compile test classes after each test recording. If the assertion expression contains an error, or if the test class was edited by hand, it can be necessary to compile the test class explicitly before it can be used.

We can now right-click the test class and see the test we have just recorded in the class's popup menu. Figure 5 shown an example with our `testName` test method from above and a second test called `testStudentID`.

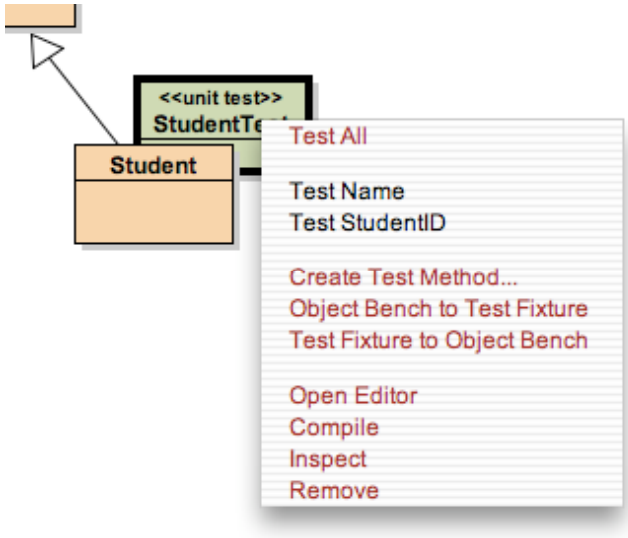

Figure 5: Test class menu with two defined test methods

Selecting a test from the menu executes that test individually. Selecting the Test All option from the test class's menu runs all tests defined in that class.

When a test is run individually, one of two things will happen: if the test is successful (the assertions in the test hold) a brief note indicating success is shown in the project window's status bar at the bottom of the window. If the test fails (an assertion fails or any other problem occurs) a test result window is displayed presenting details about the test run (Figure 6).

If all tests are run, the test result window is always displayed to show the outcome of the tests.

You can also use the Run Tests button above the test recording indicator in the main window. Activating this button will run all tests in all test classes in the package. This is the standard way to execute a full test suit for the package.

Copyright © M. Kölling

6 Interpreting test results

Summary: The test result windows shows a summary of test runs and can display failure details.

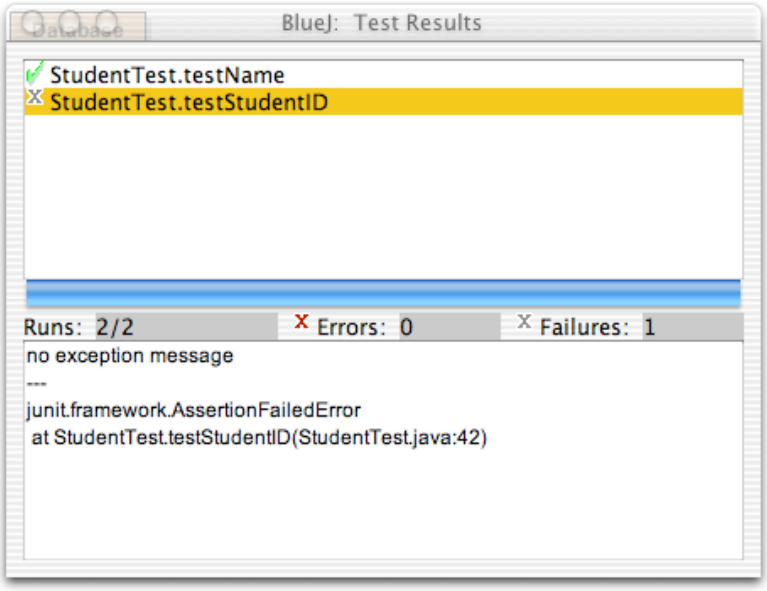

Figure 6: The test result window

When tests have been executed, the Test Result window displays a summary of the testing outcomes (Figure 6). The top pane of that window shows a list of all executed test, tagged with an icon indicating their success or failure. A green tick mark indicates a successful test, a grey cross indicates a test failure and a red cross marks an error.

The number of run tests, errors, and failures is also summarised in the middle section of the window.

A test has a failure (grey cross) if one of its assertions does not hold. For example, the test assertion may specify that a particular method result should not be null, but in this case it was.

A test has an error, if its execution resulted in any other kind of error, such as an unexpected exception being thrown.

For any unsuccessful test, details about its failure can be displayed by selecting the test in the list. The lower pane in the window then displays detail information about this failure or error.

The bar in the middle of the test window is the main summary of the test run: if it appears green, all is well - all tests have been successful. If it is red, there is a problem - at least one test has failed.

Note that on MacOS this bar does not change colour.

Copyright © M. Kölling

7 What is a fixture?

Summary: A test fixture is a prepared set of objects used as a starting point for tests.

Sometimes a test needs some objects set up before the actual test can start. For example, for several of the tests we would want for the `Database` class in the 'people' project, we need a `Database` object, a `Student` object and a `Staff` object. In addition, we might want specific state (such as a set name and age) for the student and staff member.

We could start every individual test by creating the necessary objects and putting them into an appropriate state for doing our test. But as tests get more sophisticated, this can become tedious, and we can use a better mechanism to avoid this overhead.

We can create a state on the object bench (a set of objects, each in a certain state) which we want to use as a starting point for all tests in a specific test class. This starting set of objects is called a *fixture*.

Fixtures can be defined, and they will then automatically be set up at the start of every test of the same test class, thus reducing the overhead of each individual test.

Copyright © M. Kölling

8 Creating and using test fixtures

Summary: To create a fixture for a test class, create the desired objects on the object bench and then select Object Bench To Test Fixture *from the test class's menu.*

We start creating a test fixture by simply creating the objects we need, and making method calls on the objects to set them into the desired state.

For example, for testing our database class in the people project, we may want to have a `Database` object, a `Student` object with name "Fred", which is inserted into the database, and a `Staff` object with name "Jane" who is not in the database. We can start by simply creating the objects and making the necessary call to insert Fred into the database.

Once the state on the object bench is what we want to start our tests, we can select the Object Bench To Test Fixture function from the `DatabaseTest` class.

Selecting this function will create a test fixture for the class, while removing all objects from the bench.

When a class has a fixture, this fixture will be recreated at the start of every test. For example, if we now create a new test for the `Database` class (by selecting Create Test Method from its test class) the situation defined in the fixture will automatically be restored. The fixture objects will appear in their defined state on the bench at the start of the test recording.

The fixture state can also be explicitly recreated on the bench by selecting Test Fixture To Object Bench from the test class's menu. This can be useful in cases where we want to extend the fixture later, because new test methods need additional fixture objects.

In that case, we would use Test Fixture To Object Bench to regenerate the fixture state, and then manually make the desired additions to the fixture state. Once finished, we can again select Object Bench To Test Fixture to store the fixture. The old fixture will be replaced.

Copyright © M. Kölling

9 Writing test methods by hand

Summary: Test methods can be written directly in the test class's source code.

Generating test methods and fixtures by recording our interaction and the object bench state is only one option of generating unit tests. The other option is to write these test methods by hand.

Test classes are Java classes like the other classes in the project, and they can be treated the same way. In particular, we can open an editor to see the source code, we can edit the code, compile and run it.

In traditional (non-BlueJ) use of JUnit, this is the standard way of creating test methods, and BlueJ allows the same style of working. Recording tests interactively is an addition to writing tests by hand, not a replacement.

For people unfamiliar with JUnit, a good way to start is to generate a test fixture and a few test methods interactively, and then to examine the test classes source code. We will notice that each test class has a method named `setUp()` that is used to set up the test fixture. It also has one additional method for each test.

It is perfectly fine to edit existing test methods by hand to modify their behaviour, or to add completely new, hand-written test methods. Remember that a test method's name must start with the prefix "test" to be recognised as a test method.

Those who want to know more about writing JUnit tests should read some of the JUnit literature referenced at the end of this tutorial.

Copyright © M. Kölling

10 Writing tests first

Summary: To create tests before implementation, tests can be written by hand, or method stubs can be used.

The eXtreme Programming methodology [ref] suggests that tests should be written *before* the implementation of any method. Using BlueJ's unit test integration this can be done in two different ways.

Firstly, tests can be written by hand, as explained in the previous section. Test writing then works the same way as in non-BlueJ implementations of JUnit.

Secondly, we can create method stubs in the reference class, returning dummy values for methods with non-void return types. After doing this, we can create tests using the interactive recording facility, and write assertions according to our expectations of the finished implementation.

Copyright © M. Kölling

11 Multi-class testing

Summary: The New Class... *function with class type* Unit Test *can be used to create unattached test classes.*

The examples given above used test classes which are attached to a reference class. This attachment does not prevent the test classes from making use of other class types in its tests, but it suggests a logical connection of the test class to the reference class.

Sometimes test classes are written that test several classes in combination. These classes do not logically belong to a single class. To document this, they should not be directly attached to a single class.

We can create unattached test classes by using the normal New Class... function, and then selecting Unit Test as the class type in the new-class dialog.

Unattached test classes can be used in the same way as other test classes. We can create test fixtures, make test methods and run the tests.

Copyright © M. Kölling

12 Just the summaries

1. BlueJ provides regression testing functionality by integrating JUnit.
2. Testing tools can be made visible with a switch in the preferences.
3. Create a test class by selecting Create Test Class from the class popup menu.
4. Create a test method by selecting Create Test Method... from the test class's menu.
5. Run all tests by clicking the Run Tests button. Run individual tests by selecting them from the test class's menu.
6. The test result windows shows a summary of test runs and can display failure details.
7. A test fixture is a prepared set of objects used as a starting point for tests.
8. To create a fixture for a test class, create the desired objects on the object bench and then select Object Bench To Test Fixture from the test class's menu.
9. Test methods can be written directly in the test class's source code.
10. To create tests before implementation, tests can be written by hand, or method stubs can be used.
11. The New Class... function with class type Unit Test can be used to create unattached test classes.

Copyright © M. Kölling