Introducing Unit Testing With BlueJ

Andrew Patterson
Faculty of Information Technology
Monash University
Australia
ajp@infotech.monash.edu.au

Michael Kölling
Mærsk McKinney Møller Institute
University of Southern Denmark
Denmark
mik@mip.sdu.dk

John Rosenberg
Faculty of Information Technology
Monash University
Australia
johnr@infotech.monash.edu.au

ABSTRACT

The teaching of testing has never been easy. The introduction of object orientation into first year courses has made it even more difficult, since more and smaller units need to be tested more often. In professional contexts this is addressed by the use of testing support software. Unfortunately, no adequate software to support testing for introductory students is widely available, leaving teachers and students of first year courses struggling.

In this paper we describe an attempt to address this problem by combining two existing systems that partly address our needs. We describe an integration of JUnit into BlueJ, which creates a testing tool that exhibits the flexibility and ease-of-use of the BlueJ system combined with the structured unit test approach provided by JUnit.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments - *integrated environments, interactive environments, programmer workbench*

General Terms

Experimentation, Verification.

Keywords

BlueJ, testing, JUnit, CS1.

1. INTRODUCTION

The importance of testing in software development has long been understood, and testing is an important activity in every professional development project. Good testing requires skill, which can be learned and practiced – a student's intuition is not usually enough to become a good software tester.

Several papers over many years have addressed these observations by proposing greater emphasis on the teaching of testing in introductory programming courses [3, 4].

Overall, these proposals did not have an overwhelming effect on the presence of testing in introductory curricula. Most introductory courses still cover testing only in a superficial way, or not at all. A more thorough introduction to testing is often left to software engineering classes later on in the curriculum [10]. Those who do cover it more extensively, typically find it hard to motivate students during this stage of the course.

Several problems make the teaching of testing in introductory courses difficult:

- Testing is often perceived by students as boring. It is not a creative activity, and students do not like to spend much time on it.
- Testing properly is very time intensive. Especially when regression tests are necessary, doing so without proper tool support becomes tedious.
- The teaching of testing often involves students writing detailed test plans. These test plans are usually significantly longer than the programs they are designed to test. Writing them creates a large overhead in workload.
- It is hard to motivate students to do good testing. In addition to the writing of test plans being tedious and boring, students fail to see the benefit of such a formal approach since their programs are small and simple.
- Good tools to support testing in a student environment are rare.

Recently, unit testing has been popularised greatly in the general software development community as a side effect of the popularity of the Extreme Programming methodology [1]. This popularity has been further increased by the wide availability of a software unit testing tool, JUnit [5], which helps to set up and automate the execution of unit tests.

In this paper, we argue that the use of unit testing in an introductory course can be a good medium to introduce testing to beginners, reducing some of the difficulties outlined above.

We then present a software tool to support the use of unit testing in a way appropriate for beginning students. This software tool is an extension to the BlueJ environment [8], and includes a modified interface to the JUnit tool that simplifies its use, integrates tightly with BlueJ, and exploits interactions between JUnit and BlueJ to offer highly interactive, flexible and simple support for testing.

2. PREVIOUS WORK

Several proposals have been made in the past that aim at introducing testing into introductory curricula while avoiding some of the problems. Some of these introduce testing methodology by requiring students to submit test plans or test logs which are marked by teaching assistants. Other approaches aim to give hands-on experience in the practice of testing.

Jones [4] suggests some testing activities that can be incorporated into introductory courses. These include:

- students grading other student's programs using their own test data; and

- instructors providing programs with known bugs and assigning marks for discovering bugs and documenting the bug discovery process.

Goldwasser [2] proposes a simple scheme to augment existing programming assignments with the principles of software testing. Each student submits both source code and a test set for the assignment and these test sets are run against all other submitted assignments from the rest of the class. A portion of the student's grade is based on how well the student's test set uncovers bugs in the other students' assignments.

Kay [6] suggests providing the students with automated testers as part of a comprehensive electronic submission system. The system incorporates some initial feedback to the student at submission time regarding the program's performance on a set of public tests, and reporting for teaching assistants of the program's performance on a set of private tests.

The Blue system introduced interactive testing tools [7] – a design that was later transferred into its successor system BlueJ. These tools allow flexible ad-hoc testing, but provide little support for a more organised approach.

3. TESTING OBJECTS

The character of testing has been changed significantly by the introduction of object orientation into introductory courses.

While many of the techniques for testing procedural programs are applicable in an object-oriented system, object orientation introduces some new testing problems. One of these problems is that the overhead for the construction of test cases is higher in object-oriented languages because of the size and number of separate units that require testing.

Procedural programming tended to produce large monolithic applications with long function definitions. Whilst it was hard to construct good tests for these, the infrastructure required to set up and run the tests was relatively straightforward. Object oriented programming tends to produce better separated units of code with smaller and more precise methods. This means that testing can be more effective because methods tend to be more cohesive, but the sheer number of tests means that more testing infrastructure is needed. Because of this, tools to help manage the testing process are now more important.

4. TOOL SUPPORT FOR TESTING

We have identified three activities as being representative of the type of testing performed by students:

1 **Testing immediately after implementation** is covered well for ad-hoc testing by BlueJ, using interactive method invocation. These tests, however, are transient and cannot easily be repeated.

2 **Testing after detecting a bug** is adequately addressed by symbolic debuggers and tools that allow object inspection.

3 **Testing after fixing a bug** (regression testing) is supported by testing frameworks such as JUnit. Professional tools such as Test Mentor [1] also offer relevant functionality, but are too complex for use by students.

No existing tool supports well all relevant forms of testing. The next chapter introduces a design for a tool that supports all three testing activities in a manner appropriate for beginning students. This tool is an extension of BlueJ, which already includes a symbolic debugger that adequately covers (2). Thus, our discussion concentrates on tools to support and integrate the testing activities (1) and (3).

The tool incorporates the quick and efficient BlueJ object interaction with the regression testing facilitated by JUnit, to provide an easy way for students to construct and run test cases.

5. TESTING IN BLUEJ

To prepare for discussion of the testing tool integrated into BlueJ, we first briefly describe BlueJ's testing facilities prior to this work.

The main display of BlueJ is a simplified UML diagram of the classes in the system. Each class is displayed as an icon with UML stereotypes to indicate different class types such as «abstract», «interface» or «applet».

Each of the classes displayed has an associated popup menu that displays all public constructors for the class, as well as all its public static methods. When a constructor is selected, the user is prompted to enter necessary parameters, and then a representation of the constructed object will be created on the object bench. Once an object is available on the object bench, any public method can be interactively invoked. Parameters may be entered and results can be examined.

Using the object interaction mechanism in BlueJ, a user can create the initial setup of a test phase by instantiating objects and placing them on the object bench. Methods can then be tested, without the need to write specific test drivers, by making a sequence of interactive method calls, immediately after the code has been constructed. Parameters can be entered and method results can be examined.

No test harnesses or test drivers are required to execute the methods that have just been constructed. However, this testing is ephemeral. Objects are automatically removed if any change is made to their class or if the project is closed. In particular, the potentially time consuming set up phase, where objects are constructed, must be manually repeated after each compilation. Tests cannot be easily repeated to confirm the behaviour later on in the program development. This acts as an impediment to using the tool to test methods.

6. TESTING IN JUNIT

The JUnit testing framework has become a de facto standard for implementing unit tests in Java. Similar unit testing frameworks have now been released for many other languages.

JUnit is a simple framework that defines classes for some key testing abstractions. A programmer uses JUnit by extending these classes. The main class that programmers extend is `TestCase`. Within the new extended class (called the *test case* or *test case class*), the programmer defines *test methods* (all methods that begin with the prefix 'test'). These test methods contain calls to the application's methods and assertion statements on the results which will be checked when the test is run. Any assertions that fail will be displayed. The assertion statements available in the test case class are methods such as assertTrue(), assertSame() and assertEquals().

The test methods in a test case often need access to a set of objects of known state. A *test fixture* is a common set of test data and collaborating objects shared by many tests. They are

normally implemented as instance variables in the `TestCase` class. Two special methods (the *setUp()* and *tearDown()* methods) are responsible for initialising the test fixture instance variables at appropriate times.

The JUnit framework provides a variety of extensible interfaces for displaying results. One simple implementation is the TextRunner which displays the results in a text terminal. An alternative is the SwingRunner which displays the results using a GUI. Whichever display class is chosen, the JUnit framework will display for each test method the status of the assertions, and, if any failed, provide a stack trace showing the expected values for the assertion that failed.

Once test fixtures and test methods have been written in JUnit, the system provides an easy mechanism to perform regression testing. Tests can be repeatedly run and any failing tests are highlighted.

7. INTEGRATING JUNIT AND BLUEJ

Unit testing frameworks such as JUnit support a standard way to write tests but do not provide any tools that help in this task. The BlueJ interaction mechanism is useful for ad-hoc testing, but does not provide any recording facility, so tests must be redone by hand, eliminating its usefulness for regression testing.

A combination of the two can serve to merge the best of both worlds. The result is not only the sum of both mechanisms, but the creation of a new quality of testing tool with new functionality emerging out of the combination of the two.

The design for the new testing functionality integrates BlueJ's object interaction facility with JUnit's ability to perform regression testing. It introduces explicit support for unit testing into BlueJ, based on the design and implementation of JUnit.

The new BlueJ version now recognises JUnit test classes as a special type of class. Associated with these classes, the following functionality has been incorporated into the user interface:

- constructing a test case class;
- running all the tests in a test case;
- running an individual test method from a test case;
- constructing a test fixture from objects on the object bench;
- moveing the test fixture from a test case onto the object bench; and
- constructing new test methods by recording interactions with objects on the object bench.

In order to describe this new tool in more detail, we will walk through the process of testing a simple text based adventure program. This assignment is a modified version of the zork project presented in [9]. The actual example used here is not especially relevant, and the ideas should become clear without the need to know details of the application being tested.

Our game contains five major classes: `Game`, `GameAction`, `ActionFactory`, `Parser` and `Room`. In the following example, we assume that we have implemented the `Parser` class and wish to test it. The class contains a single method:

tokenizeAndLower(java.lang.String)

This method tokenises a string, lowercases the tokens and then returns the tokens in an array..

7.1 Constructing a test class

As our first step we construct a unit test for the `Parser` by selecting the new function "Create Test Class" from the `Parser` class' popup menu. Unit tests are represented in the interface similar to other classes. They appear attached to a class they were created for, are marked with a UML stereotype «unit test» and have a distinct colour. The resulting unit test class will automatically be named `ParserTest` (Figure 1). The visual attachment to the `Parser` class is automatically maintained: when the `Parser` class icon is moved, the `ParserTest` icon moves with it.

The generated unit test class is constructed using a template unit test file. The default template can be customised by the user or system administrator to fit in with any local requirements of coding style.

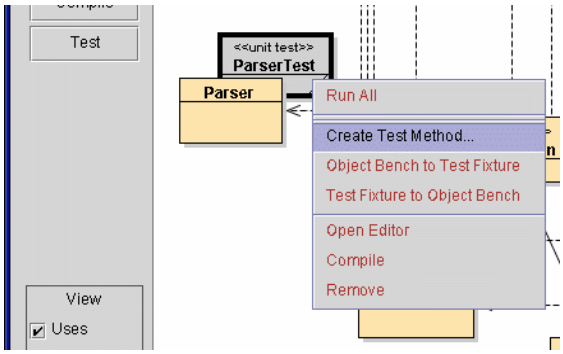

Figure 1: A Java class with associated test class. The test class popup menu is also displayed.

7.2 Creating test methods

The first test we wish to create is a test to see that the `Parser` class is always returning a valid `String` array, no matter what the input. We start the construction of a new test case by selecting "Create Test Method…" from the unit test class' popup menu (Figure 1). We are prompted for the name of the test and we enter the name "NotNull". An "End Test" button appears at the bottom right corner of the object bench. All our interactions with BlueJ, from now until the "End Test" button is pressed, will be recorded as Java source in a *testNotNull()* method of the `ParserTest` class.

We construct a new `Parser` object by selecting "new Parser()" from the `Parser` class' menu. We can now execute methods of the `Parser` object and BlueJ will perform the operation. We select the *tokenizeAndLower(String)* method and are presented with a dialog asking for parameters to the method call. As we are testing to make sure the method always give us a non-null string array, we start with a boundary case like the empty string "". As with normal BlueJ interactions, a result dialog is now displayed showing the returned object. However, because we are in test mode, the result dialog is extended to show an assertion panel that can be used to make assertions in the current test (Figure 2).

7.3 Asserting results

We want to assert that the result we received from the method is not null. To do this we check the "Assert that" checkbox. We can then select the type of assertion that we want from the drop down list of supported JUnit assertions. In our case, we select

the “not null” assertion. When we click on the “Ok” button, this assertion is added to the current test. We repeat this process for some other cases we wish to test such as the strings "a" and "AA ab". After exhausting all the cases that we wish to check we click the “End Test” button in the bottom right corner of the object bench. The *testNotNull()* method has now been added to the `ParserTest` class. After compiling the test, we are now ready to run it.

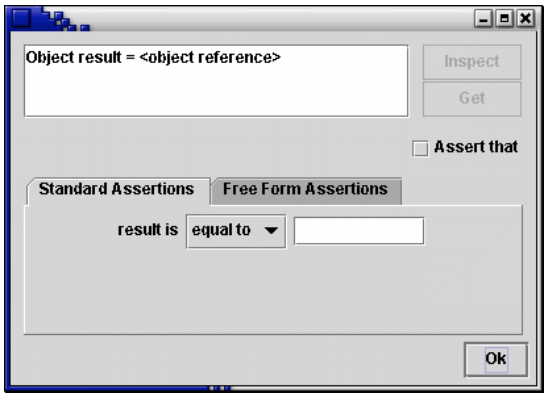

Figure 2: Creating assertions on method results

7.4 Executing tests

Whereas traditional IDE’s only allow interaction with an application in a monolithic way (by executing the main method of the program), BlueJ allows interaction at an object, class and method level. Similarly, while the standard JUnit interface only allows the execution of all tests in a test class, its BlueJ version also allows execution of separate test methods.

The popup menu for a unit test class in BlueJ has a menu item for each test defined in the class and a “Run All” entry. By selecting a test method from the popup menu, just a single test method is run. If a test is successful then a simple message indicating the success is displayed in the status bar at the bottom of the screen. If the test fails then the result is displayed in a dialog showing the failed assertion, similar to the dialog shown by the “Run All” menu. This allows quick execution of specific tests which is useful when editing the particular area of code that those tests target.

The “Run All” function in the test class’s menu runs all the tests in the selected test case.

The results are displayed in a dialog indicating the list of test methods and success or failure of each. As in other JUnit implementations, a large bar is displayed in green if all the test methods pass. If there are any failures then the bar is displayed in red. In a case where the test has failed, the bottom half of the dialog displays the results of the assertion, showing the line where the assertion failed and what the actual and expected results were (Figure 3).

BlueJ uses its own implementation of the standard JUnit SwingRunner user interface. The BlueJ version differs internally from the normal SwingRunner in that the execution of the interface code and the tests occur on different virtual machines. The interface presented however remains the same.

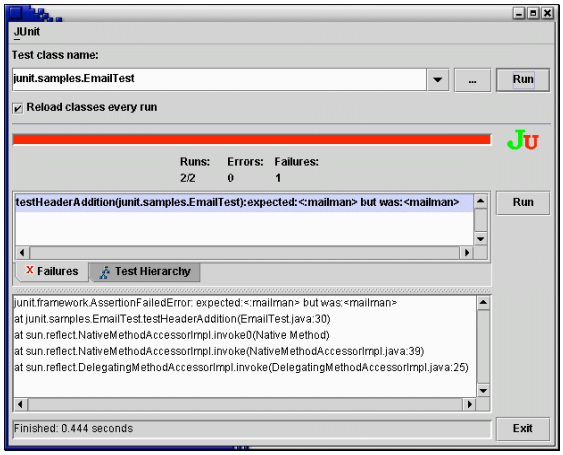

Figure 3: Result of test execution

7.5 Using test fixtures

In constructing tests for the `Parser` class we notice that there are some objects that we use in each test (a `Parser` object for example). It would be useful to be able to share the effort of constructing the objects between all the test methods. JUnit’s fixtures provide a mechanism for doing this.

A design goal for the integration of BlueJ and JUnit was that BlueJ’s object interaction methods should be useable for the construction of test fixtures in the same way that the BlueJ interaction is being used to construct test methods. Similarly, if JUnit test fixtures could be brought into BlueJ, then object interaction and test generation could be performed with existing JUnit tests.

7.5.1 Creating a test fixture

To illustrate the construction of a test fixture we will construct some tests for the `Room` class.

Our first step is to construct a test class for `Room`. We select “Create Test Class” from the `Room` class’ popup menu and the new `RoomTest` class is created. We would like to make a small set of connected rooms to test that rooms can be successfully navigated. Creating the rooms and linking them with exits requires a longish sequence of method calls, so we aim to automate this task in a test fixture.

First, we construct several room objects on the object bench with various descriptions. We then use the method interaction facility to set the exits of each room instance, passing in the other room objects as parameters.

We now want to use the Room objects on the object bench as a fixture in our `RoomTest` class. We select “Object Bench to Test Fixture” from the `RoomTest` class’ menu. The source code to construct the objects present on the object bench in their current state is saved into the `RoomTest` class’s *setUp()* method. The objects which have been saved now disappear from the object bench. They can be restored in two different ways as explained in the next section.

7.5.2 Restoring a test fixture

A test fixture can be restored to the object bench by selecting “Test Fixture to Object Bench” from the test class’s menu. This will execute the *setUp()* method and place the constructed

objects onto the object bench. BlueJ users can interact with these objects on the object bench the same way they do with objects that have been constructed through normal interaction.

The other method of restoring a test fixture to the object bench is by creating a test method. When a test class has a test fixture, the fixture's objects are automatically placed on the object bench whenever the recording of a test method is started. Any method calls which are made on these objects are local to the test method being recorded and will not impact upon the state of the test fixture objects in other test methods.

8. DISCUSSION

The mechanisms discussed so far provide an easy, yet powerful way to do unit testing in BlueJ. BlueJ's flexible interactive testing mechanisms can still be used, avoiding the need to write test drivers. These test sequences can now be recorded and replayed, providing an easy way to implement regression testing. Thus, BlueJ can now write test drivers automatically through recorded interaction.

In addition to this functionality, the current design and implementation also supports other uses of the mechanism.

8.1 Test-driven development

The Extreme Programming community strongly advocates the use of test-driven development (TDD). In this style, test cases are written after class interfaces have been designed, but prior to their implementation.

In our integrated BlueJ/JUnit system, this style of working is still fully supported. The recording facilities described above are added on to the standard JUnit functionality, but do not replace the traditional way of writing test cases. In other words: Test classes can still be written manually before implementing methods. They can then be executed later. The code that is produced by the test recording facility is standard Java source code. The test classes are normal classes that can be edited, compiled and executed like other classes (with some added functionality). Thus, recorded test cases can be modified later by directly editing the source code, and test cases can still be coded by hand.

8.2 Testing multiple classes

The standard way to create test classes has been described above: selecting the "Create Test Class" function from a class's menu creates an attached test class. In addition to this, "free" test classes (not attached to a single class) can be created by writing a new class that extends `TestCase`. BlueJ will recognise this correctly as a test case class. (Creation of such classes is also supported through BlueJ's "New Class" dialogue.)

The free test cases can then be used to hold tests that concern a combination of other classes. All test functionality is still available on these classes.

Test cases written in projects outside of BlueJ will also appear as free test cases when opened in BlueJ.

8.3 Status

All functionality described in this paper has been implemented, and is currently undergoing final testing. It is expected that, at the time of publication of this paper, a BlueJ version including this mechanism will have been released.

9. SUMMARY

The unit testing extensions to BlueJ aim to improve the tool support available for testing in introductory teaching. We have achieved this by integrating the JUnit testing framework into the BlueJ development environment in a manner that diminishes neither. At its most basic, the unit testing extensions allow the recognition and execution of JUnit test classes. We have extended this to also allow a JUnit test fixture to be moved onto the BlueJ object bench, and provided a method for converting objects on the BlueJ object bench into a JUnit test fixture. We have also developed a method for helping in the construction of unit test methods through the recording of object interactions on the object bench.

10. REFERENCES

[1] Beck, K. eXtreme Programming eXplained. Addison-Wesley, (1999).

[2] Goldwasser, M., A Gimmick to Integrate Software Testing Throughout the Curriculum. in Proceedings of the 33rd Annual SIGCSE Technical Symposium on Computer Science Education, (2002), 271-275.

[3] Jackson, U., Manaris, B. and McCauley, R., Strategies for effective integration of software engineering concepts and techniques into the undergraduate computer science curriculum. in Proceedings of the 28th Annual SIGCSE Technical Symposium on Computer Science Education, (San Jose, CA, 1997), 360-364.

[4] Jones, E., An Experimental Approach to Incorporating Software Testing Into The Computer Science Curriculum. in 31st ASEE/IEEE Frontiers in Education Conference, (Reno, NV, 2001).

[5] JUnit. http://www.junit.org, (accessed November 2002).

[6] Kay, D., Scott, T., Isaacson, P. and Reek, K., Automated grading assistance for student programs. in Proceedings of the 25th Annual SIGSCE Technical Symposium on Computer Science Education, (Phoenix, AZ, 1994), 381-382.

[7] Kölling, M. The Design of an Object-Oriented Environment and Language for Teaching Basser Department of Computer Science, University of Sydney, (1999).

[8] Kölling, M. and Rosenberg, J. BlueJ - The Hitch-Hikers Guide to Object Orientation. The Mærsk Mc-Kinney Møller Institute for Production Technology, University of Southern Denmark, Technical Report 2002, No 2, ISSN No. 1601-4219 (2002).

[9] Kölling, M. and Rosenberg, J., Guidelines for Teaching Object Orientation with Java. in Proceedings of the 6th conference on Information Technology in Computer Science Education (ITiCSE 2001), (Canterbury, 2001).

[10] Shaw, M. and Tomayko, J., Models for undergraduate project courses in software engineering. in Proceedings of the Fifth Annual SEI Conference on Software Engineering, (Pittsburgh, PA, 1991), 33-71.

[11] Silvermark, Test Mentor Java Edition User Reference 5.4, http://www.silvermark.com/documentation/, (accessed November 2002).