Going Interactive: Combining Ad-Hoc and Regression Testing

Michæl Kölling[1], Andrew Patterson[2]

[1] Mærsk Mc-Kinney Møller Institute, University of Southern Denmark, Denmark
mik@mip.sdu.dk
[2] Deakin University, Australia
patto@deakin.edu.au

Abstract. Different kinds of unit testing activities are used in practice. Organised unit testing (regression testing or test-first activities) are very popular in commercial practice, while ad-hoc (interactive) testing is popular in small scale and experimental development environments and teaching situations. These testing styles are usually kept separate. This paper introduces a design and implementation of a system that combines these testing styles to create a tool for new testing activities that both communities – professional and educational – may benefit from.

Keywords: unit testing, ad-hoc testing, JUnit, development environments

1 Introduction

Testing always has been, and in all likelihood always will be, an important part of software development. Unit testing – the test of individual parts of a system on their own – is one important form of testing systems during and after development.

Recently, support for unit testing in object-oriented systems has increased in popularity. This support exist in various forms: support for test development as an aid in specification ("test first" strategies), support for organised regression testing of program units and support for *ad-hoc testing*.

Currently one of the most popular tools for supporting unit testing in both the extreme programming (XP) community and the Java community is JUnit [2].

JUnit is a small and elegant unit testing framework that supports organised regression testing for application units. It can be used both as a pure regression testing tool, as well as a test-first tool following the extreme programming methodology.

Ad-hoc testing is the interactive testing process where developers invoke application units explicitly, and individually compare execution results to expected results. Ad-hoc testing requires support via a dedicated development environment. In the Java domain, ad-hoc testing is provided by some environments or environment plug-ins such as BlueJ [4] and BeanShell [6].

BlueJ is an integrated development environment developed specifically for teaching of object orientation to novices of the object-oriented paradigm, which provides sup-

port for interactive execution of individual methods (ad-hoc testing) via a graphical user interface.

BeanShell is a Java execution engine that allows interactive evaluation of individual textual Java expressions, including method calls. BeanShell is often used via a plug-in inside larger development environments. Popular BeanShell plug-ins exist, for example, for the Eclipse environment [1] and for the Sun One Studio [7].

The work described in this paper consists of the design and development of a single system that combines a unit testing framework with ad-hoc testing functionality. This system is based on BlueJ and JUnit.

We demonstrate that the result is not only a side-by-side coexistence of ad-hoc testing and regression testing, but that new functionality emerges through the combination of the two, which was not previously available in either of the separate systems.

Central in the resulting functionality is the ability to record interactive (ad-hoc) testing sessions and use these test recordings to automatically create JUnit test cases.

2 Testing with JUnit

The JUnit testing framework has become a de facto standard for implementing unit tests in Java. It provides a set of classes and methods that aid in writing unit tests for organised regression testing.

Programmers implement *test classes* by extending a JUnit class called *TestCase*. In this class, they implement *test methods*, which can later be executed through the framework. Several assertion methods are available for use in test methods.

JUnit also provides support for *fixtures* – a set of objects in a specific state as a basis for tests – via standard *setUp* and *tearDown* methods. Different interfaces exist to run tests and display results; the most popular is a GUI named SwingRunner.

JUnit has been extensively described in the literature, so we will, in this paper, leave it at this short summary. For the remainder of this paper, we assume that the reader is familiar with JUnit.

3 Ad-Hoc Testing in BlueJ

Ad-hoc testing – the immediate and interactive testing of code fragments – has been available for a long time in functional languages and in Smalltalk.

Most statically typed object-oriented languages were not originally integrated with interactive environments. For Java, ad-hoc testing is provided by various systems such as BlueJ, BeanShell, or DynamicJava [3].

Since we have chosen BlueJ as the basis for our ad-hoc/JUnit integration, we will give a brief summary of BlueJ's interaction mechanism prior to this work.

The main display of BlueJ (Fig. 1) 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».

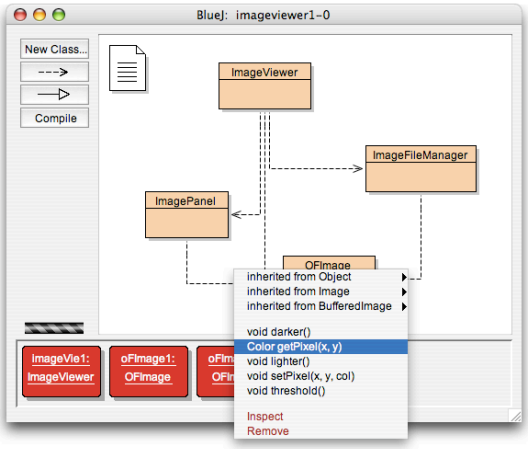

Fig. 1. The main interface of BlueJ. The window includes a class diagram, and interactively created objects on the object bench at the bottom. Both classes and objects have context menus for interaction.

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 via a context menu. Parameters may be entered and results are displayed.

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, by making a sequence of interactive method calls. Parameter values can be chosen 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. Thus, while BlueJ's interaction mechanism provides good tools for ad-hoc testing, it offers little support for a more organised approach.

4 Integrating BlueJ and JUnit

To merge the functionality of BlueJ and JUnit, we have integrated the JUnit framework into the BlueJ environment.

In its most simple form, this could mean to just provide the JUnit classes with the general libraries of the system. This would then make the JUnit functionality avail-

able as it is in any Java environment. The effect would be a side-by-side co-existence of both testing systems without interoperation between the two.

Closer analysis, however, reveals a number of possible interoperations between JUnit and BlueJ that can be provided to enhance the user-level functionality. These are discussed below.

4.1 Recognising JUnit Classes

The most fundamental specific JUnit support can be provided by recognising JUnit test classes (classes that extend TestCase) as a special kind of class type in the BlueJ environment. BlueJ already supports different class types with individually adapted user functionality. Applet classes, for example, are recognised and have an associated 'Run Applet' command that executes the applet in a web browser.

Equally, unit test classes can be recognised and be treated differently. These differences may include:

- using a distinct visual representation for test classes to separate them from implementation classes;
- providing specific default source code skeletons;
- providing functionality for selective display, which hides test classes temporarily;
- association of specific test commands with the test class (see below).

Each of these enhancements can make working with test classes more convenient.

4.2 Executing Single Test Methods

BlueJ's interactive test support allows the interactive invocation of single methods by selecting them from a pop-up menu.

Since JUnit test cases are individual methods, integration in BlueJ results in these methods being interactively executable. Thus, in BlueJ, test cases can easily be executed individually. These test method calls could be recognised as such by the environment, and success or failure be reported in an appropriate manner. (Using a standard interface like the SwingRunner GUI may not be appropriate to display the result of a single test method.)

4.3 Executing All Test Methods of a Test Class

One of the specific commands associated with a test class (provided in its context menu) can be a "Test All" command, which would execute all test methods defined within that class. This function is similar to a standard JUnit test run. An interface similar to the standard JUnit SwingRunner may be used to display results.

4.4 Executing Tests from All Classes

If JUnit test classes are recognised by the environment, a "Test All" function may be provided (e.g., as a toolbar button) which runs all test from all test classes in a package or a project. Again, a SwingRunner-style interface may be used to present results.

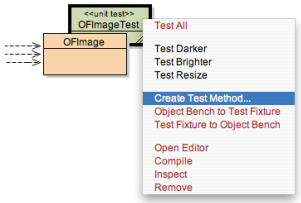

Fig. 2. A reference class and attached test class. The test class is marked with a stereotype and different colour. A context menu shows the test class operations.

4.5 Attached Test Classes

BlueJ presents class diagrams of projects, and interaction is heavily designed around using contextual menus on classes and objects.

This can be used for a further enhancement: Instead of creating test classes independent of other classes, they may be created in direct association with an exiting implementation class. For example, as class's context menu may have a 'Create Test Class' command, which creates a test class *associated with this specific class*. This association is semantic: It signifies that the purpose of the test cases in this test class is to test the associated class (which we term the *reference class*). The association can also be functional: The test class may have a 'Create Test Stubs' command, which automatically creates stubs for all public methods in its reference classes. Lastly, the association may be visual: We can visually attach the test class to the reference class in the class diagram to signify this association to the user (Fig. 2). Dragging the reference class on screen would automatically move the test class with it.

Attached test classes could be supported in addition to *free* test classes (those not attached to a specific class). Free test classes contain tests for multiple reference classes.

4.6 Recording Interaction

Among of the biggest advantages of ad-hoc testing is that it does not require manual writing of test drivers and its action sequences can be decided dynamically: seeing the result of one test can determine the next course of action.

One of the biggest advantages of written test drivers is that tests can be replayed multiple times in the future.

Each does not have the characteristic of the other: BlueJ's ad-hoc tests cannot be replayed, and writing JUnit test drivers is static: the programmer usually does not see the result of one method call before writing the next.

Merging BlueJ with JUnit allows us to combine both advantages. We can execute dynamic interactive tests, while recording the test sequence and automatically writing JUnit tests cases from that recording. This could be done by providing a "Create Test Method" command in a test class's context menu, which starts a recording process until it is explicitly ended by the user. At that time the recording is transformed into Java source text and written into the test class as a test method.

Tests can now be done interactively and this interaction can later be replayed.

Since test classes are an extension of standard classes in BlueJ (and thus also have the standard functionality), recording test cases interactively is just an added functionality to the standard (manual) creation of methods. The class can, of course, also be manually edited, and test methods may be written by hand. In fact, both techniques could be combined: tests can be recorded first, and later modified by manual edit. There is nothing special about recorded tests: they are transformed to standard Java source code and can be treated and processed like other test methods.

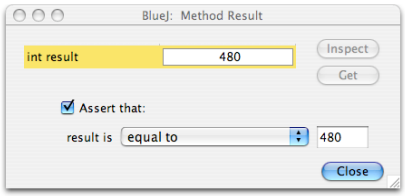

Fig. 3. A method result dialog allows attachment of assertions to method calls.

4.7 Asserting Test Results

If we want to support interactive test recordings, we need to add a mechanism to specify assertions on results during the interactive test activity.

BlueJ already has a method result display dialog. When an interactive method call returns a value, this value is shown on screen.

We propose to extend this result dialog with an assertion panel, which may be shown only while we are in "test recording" mode. This assertion panel would

- provide the option to attach an assertion to the result;
- provide a choice of available assertions; and
- allow us to enter values for the assertion parameter.

Since in many cases the actual return value may be correct, the dialog could enter the actual return value as the default for the assertion parameter, thus making the creation of the assertion convenient in many cases (Fig. 3).

4.8 Creating Text Fixtures From Interaction Objects

JUnit text fixtures correspond to a set of objects interactively created on the object bench in manual tests.

This relationship can be exploited to aid the creation of fixtures for test classes: we can create test fixtures by manually creating and preparing a set of objects, and then invoking an "Object Bench to Test Fixture" command from the test class's context menu. This command would create Java source code in the test class's setup method that creates objects identical to those currently on the object bench. This function thus corresponds to interactive test fixture creation.

Care must be taken in the implementation of this function. Several implementation approaches exist, and these can have subtle differences and associated problems.

One approach is to serialize the objects, and to read them in as part of the fixture setup. This approach is problematic, since it depends on external files and removes the contextual independence of the source code.

The algorithm we have chosen is based on recording all interaction with the object bench since it was last cleared. This could be done in various ways: The interaction can be stored in an action tree that encodes the objects, their methods and their interactions. In such a tree, however, it is difficult to maintain the sequential ordering of the calls, which is important for the actual object state.

Another recording approach is to textually record all object bench interaction, and to replay the creation later from the actual Java source text. This is the approach we have chosen for our implementation. Objects can be re-created later in the exact same sequence. In short: while the algorithm must be designed carefully, recording is possible, and fixtures can be interactively created.

4.9 Using Text Fixtures for Interactive Testing

The test fixture/object bench relationship can be exploited the other way around as well: Existing fixtures (whether created via recordings or manual writing) can be copied onto the object bench. There, they are available for interactive testing. This function could be made available to users via a "Test Fixture To Object Bench" command in the test classes context menu.

There are at least two scenarios where this may be useful: Firstly, it can be used to extend existing fixtures. We could, for example, allow the repeated recording of test fixtures from the object bench, where the last recording replaces the previous one. Since the old one gets replaced, it would be beneficial to have functionality to extend, rather than replace, an existing fixture. This can now be easily achieved by first copying the existing fixture to the object bench, then interactively adding an object, and copying the bench back to the fixture.

The second case where such a function is useful is for pure interactive testing. Even if all tests are intended to be done interactively, the concept of a fixture that can be recreated for further testing later is very useful. Thus, it can be beneficial to use test classes purely for the purpose of storing fixtures for interactive testing, without the intention of ever creating test methods. This can now easily be done using a combination of "Object Bench To Test Fixture" and "Test Fixture To Object Bench" commands.

5 Discussion

5.1 Functionality: Cooperation and Coexistence

The discussion in the previous sections demonstrate that combining JUnit and BlueJ's ad-hoc testing mechanism can result in something more than mere co-existence of two test paradigms. Elements from both test-worlds can be mixed and combined, resulting in a new quality of system interaction that can be useful for both original tasks: creating regression tests, and ad-hoc testing.

Creation of regression tests benefits, since these tests can be created via dynamic interaction that allows the tester to see and inspect actual test results during the creation of a test sequence. Equally, fixtures can be created interactively.

Ad-hoc testing benefits since some unit testing functionality provided by JUnit, such as fixture creation, becomes available for ad-hoc testing.

This mix-and-match functionality from elements from both test paradigms creates new possibilities for test interaction.

Existing interaction styles are not negatively affected by this. Test-first methodologies, for example, initially seem to contradict the recording approach – since there is nothing to do the recording with if the test is created before the method. The automatic recording functionality, however, does not replace, but extend the hand-written test functionality. We have taken great care in our design to ensure that the recorded test cases result in pure Java source code in accessible standard classes. Thus, the recording functionality can co-exist with manual test-first functionality, and code produced by each can indeed co-exist in a single source file.

Test-first methodology can, in fact, also make use of the interaction recording, if method stubs are created first. Then interactive calls to those stubs can be used to create test cases before method implementation, and assertions can be written.

The choice, manual writing or interactive recording, is up to the user.

5.2 Implementation

All of the functionality described in this paper has been implemented and tested. The latest, recently published, version of BlueJ (version 1.3.5) implements the described scheme, using the standard JUnit framework as a core extension to the BlueJ system. The JUnit core classes are used unchanged, while the interface is a modified re-implementation of the standard SwingRunner, in addition to the interface elements described in the paper. The SwingRunner had to be adapted for BlueJ, among other things because BlueJ uses two separate virtual machines: one for execution of the environment itself, and one for the execution of user code. This necessitated a restructuring of JUnit to be spread between these two machines.

All figures in this paper are actual screenshots from the released version of BlueJ.

5.3 Significance

In this work, the proposed merging of JUnit with ad-hoc testing functionality was done in the context of an educational environment. As such, the actual resulting product may not be of major interest to professional developers outside the education and training area.

We have tried to show, however, through the discussion in this paper, that the idea is not specific to educational environments. Very similar implementations may be created for professional environments, such as Eclipse or Sun ONE Studio.

In the context of BlueJ, the integration of JUnit functionality seems to have led to an increase in coverage of organised regression testing in introductory programming courses. Even though availability of this functionality is recent, we have already had repeated positive feedback, including reports of teachers including this functionality into their curricula. A short tutorial on this functionality [5] has become a popular download from the BlueJ web site.

In the context of professional programmers, functionality like this may aid in training programmers in the use of JUnit, regression testing and test-first strategies. It may also become helpful in actual daily testing work for professionals. This can be evaluated further, when the functionality is available in a professional environment.

6 Summary

In this paper, we have compared two testing approaches: organised regression testing using JUnit and ad-hoc testing using BlueJ. We have shown that these testing approaches can be combined in a single environment. Doing so results in a fertile interaction of both approaches, where elements of each can be mixed and made to interoperate, so that new styles of test interaction evolve. A detailed description of possible interactions has been given, and an implementation of these has been provided.

The implementation is currently in public use, with very positive feedback.

References

1. Eclipse: website at http://www.eclipse.org, accessed January 2004
2. Gamma, E, Beck, K.: JUnit, website at http://www.junit.org, accessed January 2004
3. Hillion, S.: DynamicJava, website at http://koala.ilog.fr/djava, accessed Jan 2004
4. Kölling, M., Quig, B., Patterson, A., Rosenberg, J., *The BlueJ system and its pedagogy*, Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology, Vol 13, No 4, (2003) 249–268
5. Kölling, M., *Unit Testing in BlueJ*, Technical Report 2004-01, The Mærsk Mc-Kinney Møller Institute for Production Technology, University of Southern Denmark, Technical Report 2004, No 1, ISSN No. 1601-4219.
6. Niemeyer, P.: BeanShell, website at http://www.beanshell.org, accessed January 2004
7. Sun Microsystems: Sun ONE Studio, website at http://wwws.sun.com/software/sundev/, accessed January 2004