JUnitMX — A Change-aware Unit Testing Tool

Jan Wloka[*], Barbara G. Ryder[†], and Frank Tip[‡]
[*] Dept. of Computer Science, Rutgers University, Piscataway, NJ 08854, USA
jwloka@cs.rutgers.edu
[†] Dept. of Computer Science, Virginia Tech, Blacksburg, VA 24061, USA
ryder@cs.vt.edu
[‡] IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA
ftip@us.ibm.com

Abstract

Developers use unit testing to improve the quality of software systems. Current development tools for unit testing help to automate test execution, to report results, and to generate test stubs. However, they offer no aid for designing tests aimed specifically at exercising the effects of changes to a program. This paper describes a unit testing tool that leverages a change model to guide developers in the creation of new unit tests. The tool provides developers with quantitative feedback and detailed information about change effects, which not only facilitate the writing of more effective tests, but also motivate developers with an achievable coverage goal.

1 Introduction

Unit testing is used by software developers for various purposes. First and foremost, unit tests are used to test the effect of changes on existing functionality. When used in regression testing, they can also protect against unintentional change effects caused by other developers. Moreover, a set of unit tests is a prerequisite for any refactoring activity [4], and in test-driven development, unit tests are used to guide developers with the design of new functionality before implementing it [2].

While most developers agree on the advantages of having a solid test suite with good code coverage, most also admit the difficulty of developing such a test suite. For example, test-driven development encourages developers to implement the "simplest thing that could possibly work" [5] for a given test, which ideally results in a test suite that reveals the effect of any change on existing functionality. However, if a developer makes changes to an application that has low test coverage, the absence of test failures (a "green bar" in JUNIT) does not and should not give much confidence about the correctness of these changes. In fact, the situation is even worse because developers often *do not know* what is covered by their tests, and therefore do not know how much confidence they should gain from successful tests. Furthermore, it can be difficult to decide how to construct tests that effectively exercise newly added functionality. Often, developers write unit tests "blindly", that is, without really knowing which parts of the new functionality are covered by existing tests and which require additional tests. Since successful tests do not show the absence of errors, but rather the inability of the test suite to find any error, the "green bar" may leave the developer feeling overconfident.

In this paper, we present a unit testing tool, JUNITMX[1], which is aware of the developer's edit in a program, and thus can guide him in writing those unit tests that effectively exercise all changed parts of a program and their effects on program behavior. The remainder of the paper is structured as follows. Section 2 describes background, terms and concepts on change impact analysis. Section 3 presents the tool JUNITMX, how it is used, and implementation details. Section 4 illustrates how JUNITMX can guide the development of unit tests, and Section 5 presents concluding remarks.

2 Change-aware Development Tools

Developers are changing software systems in their daily work, e.g., to implement new features, fix faults or refactor overly complex program structures. Their tool support for program changes, however, is often limited to change history logs, access to version control systems, or textual merge views. Other tools within an integrated development

[1] The MX in JUNITMX stands for multi-extension. The testing model of JUNIT is extended with a custom class loader that allows for instrumentation before tests are run, and a post-processor that augments test results with data from additional analyses.

environment (IDE) for, e.g., testing, debugging, or quality assurance, have no access to the developer's program edit. Change-aware development tools are an attempt to expose change information within an IDE to other development tools.

2.1 Change Model and Classifications

A technique that can be used to predict the possible effects of a program edit on a code base is called *change impact analysis*. It computes an abstract representation of a program edit, subdividing it into a set of atomic changes. This representation enables a classification of different kinds of changes and their dependences, making program edits amenable to program analysis. The specific change impact analysis used consists of decomposition of the edit, computation of change dependences and impact classification [10, 6, 9].

Decomposing a developer's edit. First, the textual difference between two program versions, is decomposed into a set of *atomic changes*. An atomic change represents the modification of a program element. Atomic changes reflect the semantics of the language, e.g., adding a method (AM), changing the body of a method (CM), or deleting a field from a class (DF). The element in the program that is affected by a change is called *denoted program element*. A complete set of atomic changes and a full introduction to change impact analysis is presented in [10, 9].

Change dependences. After the decomposition of the edit, dependencies between atomic changes are computed. An atomic change may be dependent on one or more other atomic changes, that must be applied also in order for the resulting program to compile (structural dependences) [7]. Other effects of an edit on program behavior are captured by *mapping dependences* [12] (e.g., changing a field initializer may result in an implicit change to the bodies of the constructors for the class in which the field is declared).

Impact classification. In the last step, atomic changes are correlated with program representations, such as a call graph, to compute structural and behavioral effects of every atomic change. A dynamic call graph is captured for every unit test in the suite. The call graphs are used to obtain the (method-level) changes that can affect a test's outcome. Every change that can be mapped to a graph node or edge, or is a transitive dependence of such a change is considered to be covered by the test [9]. All other changes are reported as *un-covered changes*. Since atomic changes are associated with program elements (AST nodes), every denoted element can be classified as *Addition*, *Change*, or *Deletion*, which again can be either *Structural* or *Behavioral*. For example, a method associated with changes {AM, CM} is classified as *Behavioral Addition*.

2.2 Applications

Our change impact analysis results in two primary mappings (i) atomic changes to program elements and (ii) atomic changes to program behavior represented by a dynamic call graph for each test. Both mappings can be used to classify change effects on the program's code base and on program behavior. Several classifications of changes have been shown useful in support of various development activities:

Structurally dependent changes can be used to form a syntactically valid program version from a subset of atomic changes [3, 7].

Structural and mapping dependences combined can form a program version from a subset of atomic changes with a valid program behavior according to the test suite associated with this intermediate version [12].

Affected tests can be determined as the set of tests that exercise a program element denoted by a set of atomic changes [9].

Covered changes can be computed for each test, representing those atomic changes that are directly or indirectly associated with a program element exercised by the test, also termed *affecting changes* [9].

Failure-inducing changes that are most likely to cause a test failure can be computed by analyzing the relationship between changes and the passing and failing tests that they affect [11, 8].

Safely committable changes are the subset of atomic changes that can be released safely to a version control repository because they will not alter any test outcome, even in the presence of failing tests [12].

Based on these classifications, core development activities, such as debugging [3, 7], fault localization [11, 8], testing and test development [1], and resolution of change conflicts [12] can be supported by tools.

3 The Change-aware Unit Testing Tool

The unit testing tool JUNITMX is an extension to the JUnit Eclipse plug-in. It leverages the change impact analysis described in the previous section to guide developers in writing more effective unit tests.

3.1 Using JUNITMX

The tool JUNITMX is built as an extension to the widely used *JUnit Eclipse* plug-in. Developers already familiar with JUnit and the *Eclipse JDT*[2] can build on knowledge with a familiar tool when using JUNITMX. Change-aware tools operate on two program versions: an *original* and an

[2] http://www.eclipse.org/jdt/

edited version. JUNITMX provides different options for selecting these program versions. By default it uses the latest version in the repository (CVS HEAD) as the original version, and the program version in the local workspace as the edited version. Alternatively, a developer may select a specific version in the repository as the original version, or to choose two different projects in the local workspace as the original and edited versions.

A special run configuration enables developers to run a JUnit test suite with JUNITMX. This configuration runs the test suite associated with the edited version. JUNITMX augments the existing JUnit plug-in by showing the following information:

1. The number of changes that are not covered by any test is shown along with the number of test runs, errors and failures. This number is an upper bound on the changes that should be covered by additional tests.

2. JUnit's familiar *red–green* color scheme is extended to *red–yellow–green*. Intuitively, the *green bar* is only shown if all tests pass and the all changes are covered by the tests. If all tests pass but some changes are not covered by tests, a *yellow bar* is shown. Whenever a test fails or crashes, a *red bar* is shown.

3. An additional tab lists all changes not covered by the test suite. Changes are classified into two groups, *Additions* and *Changes* [3], and further classified into code and dispatch changes. An associated source code comparison view outlines the source code differences that led to the change. Clicking on such a change will result in opening the source code editor and navigating to the denoted program element.

Given this information, a developer can start creating new tests to ensure that as yet not covered changes have no unanticipated effect on the program behavior.

3.2 JUnit Integration

JUNITMX hooks into the execution of a JUnit test run and adds pre- and post-processing phases.

In the *pre-processing* phase, CHIANTI[4] is used to compute a coarse-grained representation of the edit as a set of *atomic changes* with structural dependences and mapping dependences between them.

During execution, JUNITMX uses a custom class loader for instrumenting the target application's classes as they are loaded. This classloader was developed using DILA[5], a library for dynamic load-time instrumentation that is based

[3] *Deletions* are not shown since they are removed from the program and cannot be covered by any test. Only their effects, e.g., change of dispatch, are considered by JUNITMX.

[4] http://www.prolangs.rutgers.edu/projects/chianti/

[5] http://www.prolangs.rutgers.edu/projects/dila/

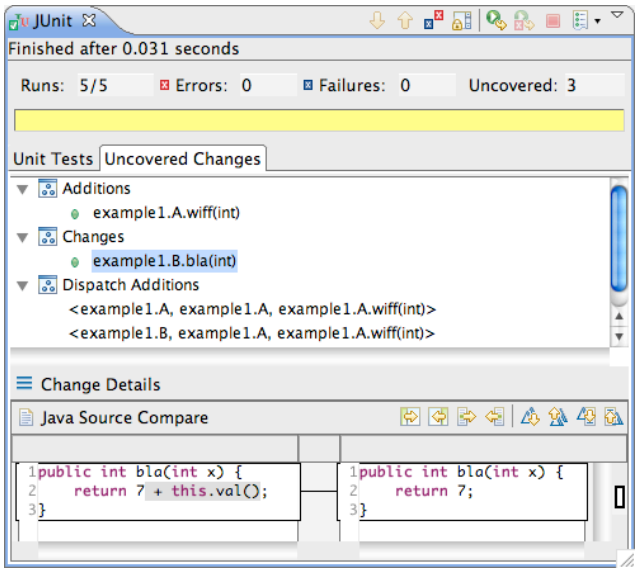

Figure 1. The User Interface of JUNITMX.

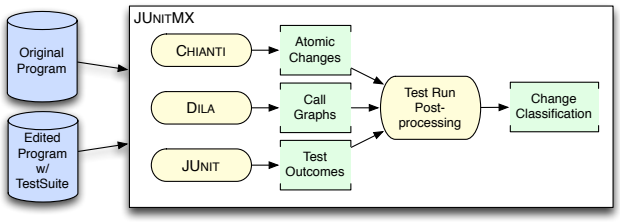

Figure 2. Overview of the JUNITMX tool.

on the bytecode utilities from the WALA program analysis library[6]. The instrumented classes contain instrumentation code for constructing a dynamic call graph for every test as it executes.

After the execution of the test suite of the edited program version has completed, the *post-processing* performs the actual change impact analysis. The dynamic call graphs are correlated with the atomic changes to calculate those changes that are covered by each test. Since changes cannot be applied or tested in isolation, also their interdependences have to be considered. We say a change is *covered by a test* if its denoted element is exercised by the test or one of its (transitive) dependences is covered by the test [12]. Once all covered changes are computed, we calculate those remaining *uncovered* changes that are not covered by any test.

4 Test Development with JUNITMX

Assume that the developer is working on an extension the Counter application shown in Figure 3. The orig-

[6] http://sourceforge.net/projects/wala

```
public class Counter {
  protected int sum;
  public Counter() { sum = 0; }
  public int getSum() { return sum; }
  public void inc() { ++sum; }
}
public class MultiCounter extends Counter {
b  private Counter[] counters;
   public MultiCounter(Counter[] cs) { counters = cs; }
   public void inc() { /*call inc() on each counter*/ }
}
public class Tests extends TestCase {
a  /*tests for Counter omitted*/
   public void testMultiInc1() {
     MultiCounter m = new MultiCounter(
        new Counter[] { new Counter(), new Counter() };
     m.inc();
     assertEquals(1, cs[0].getSum());
     assertEquals(1, cs[1].getSum());
   }
c  public void testMultiInc2() {
     Counter m = new MultiCounter(
        new Counter[] { new Counter(), new Counter() };
     m.inc();
     assertEquals(2, m.getSum());
   }
}
```

Figure 3. Partial code for Counter application.

inal program version contains only the lines *not* shown in boxes, and is extended to provide counting of multiple values. The developer makes sure that he is working on the latest version of the repository and creates the test `testMultiInc1` (see 3(a)). The new test asserts that every call to method `inc()` on a `MultiCounter` object increments each counter stored in this object. To get the new test compile, the developer defines a new constructor in `MultiCounter` and adds the field `counters` to store multiple `Counter` object (see 3(b)). The code satisfies the compiler but the test run fails as expected. A redefinition of method `inc()` in class `MultiCounter` that increments all counters stored in field `counters` can satisfy the test. While `testMultiInc1` passes now, JUNITMX is still not satisfied and shows a yellow bar. Apparently, there are changes that are not covered by the current test suite.

The developer reviews the list of changes shown by JUNITMX and indeed, there is a lookup change associated with method `MultiCounter.inc()` that is not covered by any test. A double click on this change opens the editor focussing on method `inc()`. Inspecting test `testMultiInc1`, the developer proceeds to write another test (see 3(c)) that exercises method `inc()` on a `MultiCounter` object but with the declared type `Counter`. It asserts that method `inc()` increases all counters with a call to method `getSum()`. The benefit of the new test is evident from the fact that it fails. In response to the failure, the developer proceeds to fix the exposed fault with a redefinition of method `getSum()` in class `MultiCounter`. Now, that all test cases succeed and all changes are covered the bar finally turns green.

5 Conclusions

JUNITMX seems to be a powerful extension to JUnit that can help developers to write more effective tests. Moreover, a full change coverage is not just an achievable goal but also seems to reduce the likelihood of introducing faults to the program. This correlation and a change-centric test development approach are major targets of our current evaluation.

Finally, JUNITMX is a good example for the potential of change-aware tools. It demonstrates how even complex development activities can be supported by tools when they are aware of what a developer has done to the code.

References

[1] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold. Matrix: Maintenance-oriented testing requirements identifier and examiner. In *TAIC-PART '06: Proceedings of the Testing: Academic & Industrial Conference on Practice And Research Techniques*, pages 137–146, Washington, DC, USA, 2006. IEEE Computer Society.

[2] K. Beck. Aim, fire. *IEEE Software*, pages 87–89, September/October 2001.

[3] O. Chesley, X. Ren, and B. G. Ryder. Crisp: A debugging tool for Java programs. In *21st IEEE International Conf. on Software Maintenance (ICSM), Budapest, Hungary*, pages 401–410, Sept. 2005.

[4] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[5] P. McBreen. *Questioning Extreme Programming*. The XP Series. Addison-Wesley Professional, 1st edition, July 2002.

[6] A. Orso, T. Apiwattanapong, M. J. Harrold, G. Rothermel, and J. B. Law. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the International Conference on Software Engineering (ICSE 2004)*, pages 47–50, May 2004.

[7] X. Ren, O. C. Chesley, and B. G. Ryder. Identifying failure causes in Java programs: An application of change impact analysis. In *IEEE Trans. on Softw. Eng.*, volume 32, pages 718–732, 2006.

[8] X. Ren and B. G. Ryder. Heuristic ranking of java program edits for fault localization. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 239–249, New York, NY, USA, 2007. ACM.

[9] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for practical change impact analysis of Java programs. In *Proc. Conf. on Object Oriented Programming, Systems and Applications (OOPSLA'04)*, pages pp 432–448, Oct. 2004.

[10] B. G. Ryder and F. Tip. Change Impact Analysis for Object-oriented Programs. In *Proc. Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, pages 46–53, 2001.

[11] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip. Finding Failure-inducing Changes in Java Programs Using Change Classification. In *Proc. 14th Symp. on the Foundations of Software Engineering (FSE-14)*, pages 57–68, Portland, OR, USA, Nov. 7–9, 2006.

[12] J. Wloka, B. Ryder, F. Tip, and X. Ren. Safe-commit analysis to facilitate team software development. In *31st International Conf. on Software Engineering (ICSE 2009)*, 2009. To appear.