

Tool Support for Change-Centric Test Development

Jan Wloka, IBM Rational

Einar W. Host, Norwegian Computing Center

Barbara G. Ryder, Virginia Tech

Applying change impact analysis to test-driven development provides software designers quantitative feedback they can use to meet a coverage goal and avoid unanticipated change effects.

Testing increases confidence in software's correctness, completeness, and quality.¹ By executing a test on a program, developers can check the outcome against the program's specification to identify faults. Various testing levels can serve different purposes during development—for example, unit and integration testing let developers test an implementation and its effects on existing functionality. In test-driven development, a unit test acts as a functionality specification before implementation, letting developers apply only the code necessary to pass the test.²

Thus, a set of unit tests is a prerequisite for any refactoring activity.³

Although most developers agree on the advantages of having a solid test suite with good code coverage, most also admit the difficulty of developing it. Implementing the “simplest thing that could possibly work”⁴ ideally results in a test suite that reveals any effect the added code has on existing functionalities. However, what if the test coverage for the system's unchanged parts is low and the test suite returns a green bar, indicating all tests passed? Because successful tests don't show the absence of faults, this green bar could leave the developers feeling overconfident. Moreover, they might not have implemented the simplest thing that could possibly work, which implies additional tests might be needed to validate the entire program edit (in this context, the textual difference between two program versions). In both cases, the developer has written his or her tests “blindly”—that is, missing possible side effects or being unable to verify that the edit caused no unexpected alteration in system behavior.

Our approach to test development uses a specific change impact analysis^{5,6} to guide developers in creating new unit tests. This analysis specifies those developer-introduced changes not covered in

the current test suite, thereby indicating that some tests are missing. It supports developers in test-driven development by indicating whether their newly added functionality was the simplest thing that could possibly work and which additional effects on system behavior the test suite doesn't cover. The developer can then choose to add or extend a test to cover every effect on existing code. Even if the test suite covers all the changes and all the tests pass, the system might still have faults, but such coverage makes it more likely that new faults haven't been introduced and that all changes can be committed safely into the shared repository. We define a new metric—*change coverage*—that supports *change-centric test development* with our tool JUnitMX. We also discuss a feasibility study that shows the potential benefit of using our approach in current software development practice.

Change-Centric Test Development

When developing unit tests for improved or new functionality, developers don't always know whether they've done a good job. Two challenging aspects of writing a good unit test suite are to ensure that the suite can exercise the program elements involved and that the suite covers all effects on other functionalities.

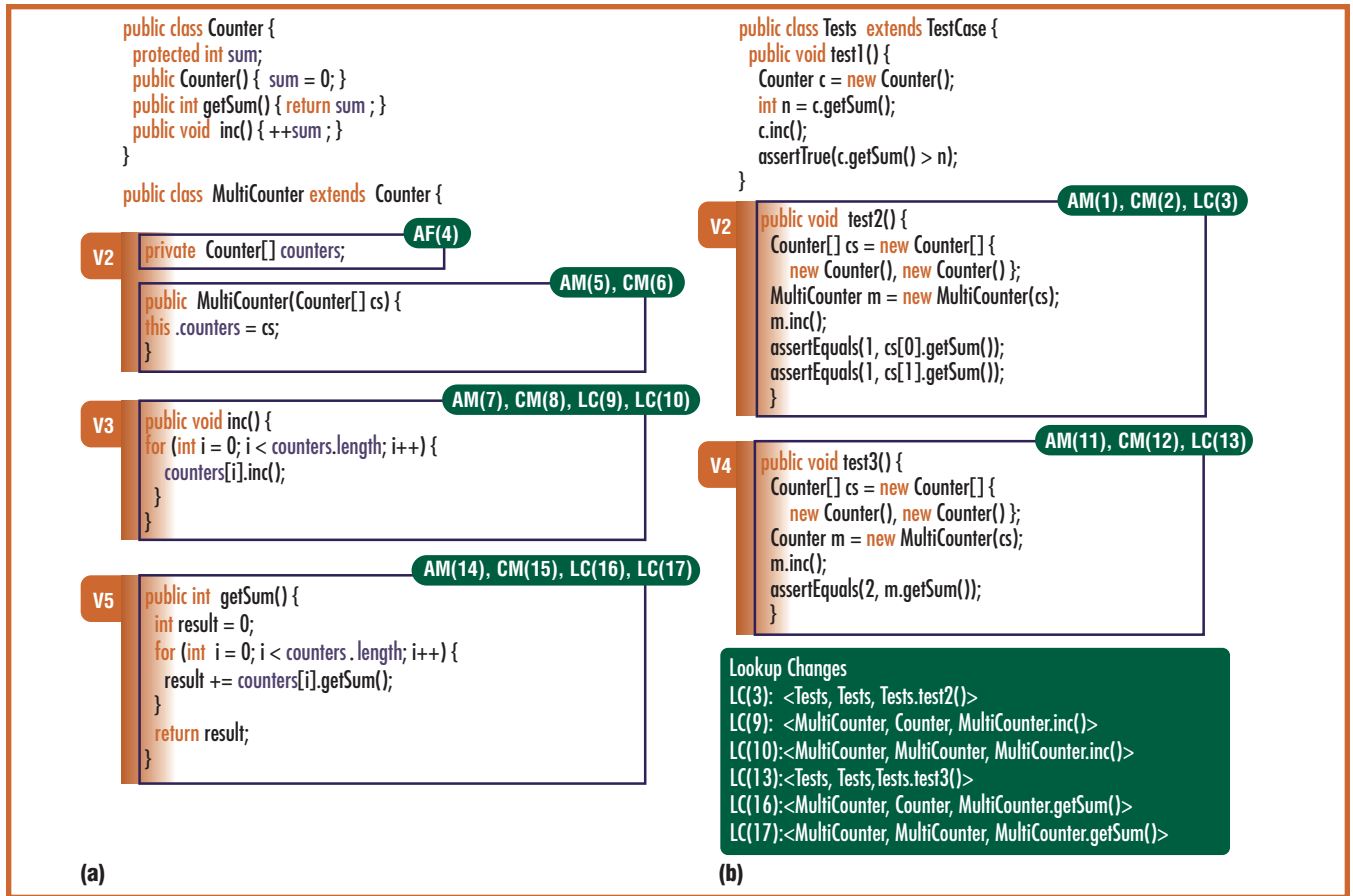


Figure 1. Change-centric test development. Annotated boxes show program changes from (a) the original and edited version of the example program. The original program consists of all program fragments except those shown in the boxes. The edited program is obtained by adding all boxed code fragments. (b) Tests associated with the example program and lookup changes (LC) describe the effects via dynamic dispatch.

Applying Change Impact Analysis

Change impact analysis is a technique for predicting the possible effects of a program edit on a code base by computing an abstract representation of that edit and subdividing it into a set of changes.⁷ 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 we describe here consists of decomposition of the edit, computation of change dependences, and change classification.^{5,6,8}

We can decompose an edit into a set of *atomic*—or smallest possible—*changes* to a program. Examples of atomic changes include adding a method (AM), changing a method’s body (CM), adding a field (AF), or deleting a field from a class (DF). The element in the program that a change affects is called the *denoted program element*. A complete set of atomic changes and a full introduction to change impact analysis appears elsewhere.^{5,6}

After the decomposition of the edit, dependences between atomic changes are computed. An atomic change might depend on other atomic changes that must also be applied for the resulting program to compile.⁸ Other dependences stem from specific atomic changes that indirectly impact program be-

havior—for example, changing a field initializer might implicitly change the bodies of the constructors for the class in which the field is declared.⁸

The Java program in Figure 1 illustrates change-centric test development with a simple counter application that can increase, store, and return a single integer value. Let’s say a developer wants to extend it to a multi-counter that manages several instances of the original counter. Figure 1a shows the actual program code; Figure 1b, the associated test suite. Annotated boxes indicate program edits. The original program, V1, consists of all the code, except that shown in the boxes. Each of the four subsequent program versions have gray labels—for example, we can construct V2 from V1 by applying all changes whose boxes are within the label V2, and so forth.

In Figure 1, the developer adds a constructor to class `MultiCounter` as part of the edit that leads to V2. This addition is expressed as two atomic changes: AM(5), CM(6), as shown in the shaded box label. The constructor is the denoted program element of these two changes—note that the developer can’t apply CM(6) without AM(5), which makes it dependent on AM(5). Similarly, CM(6) requires the added field `counters` (AF(4)).

Ideally, unit tests should encode every feature request, fault fix, or other improvement.

Many kinds of edits can alter a Java program's existing dynamic dispatch behavior, such as adding an overriding method in a subclass or changing visibility from *private* to *public*. A lookup change (LC) represents an edit's effect on dynamic dispatch. For example, the addition of method `inc()` to class `MultiCounter` results in two LC changes: LC(10) corresponds to the newly possible dispatch of `MultiCounter` objects to the new method `inc()`, and LC(9) corresponds to the redirected dispatch of `MultiCounter` objects referred to by a `Counter` reference in a call of `inc()`, which post-edit, will be directed to `MultiCounter.inc()` rather than to `Counter.inc()`. All the LCs corresponding to the edits in Figure 1 appear in the shaded box in the figure's right-hand corner. (The first element in each LC is the receiver object's instance type, the second element is the method invocation's static or compile-time type, and the third is the actual target method.)

After the dependences have been computed, the test suite is run on the edited program and profiles are collected to obtain each test's calling structure (for example, a call graph). By mapping method-level atomic changes to a test's calling structure, the analysis computes the set of *affected tests* (those whose behavior might differ after the edit).⁵ In Figure 1, when the program transitions from V2 to V3, `test2` is affected, but `test1` isn't. Intuitively, this is because `test2` calls `MultiCounter.inc()` after the edit, whereas it called `Counter.inc()` in V2. By contrast, `test1` calls the same set of (unchanged) methods before and after the edit.⁵

For each affected test, the analysis can isolate those parts of the edit that might have affected it—its *affecting changes*—the parts of the edit that can be mapped to the test's calling structure. Considering the edit from V2 to V3 in Figure 1, for example, {AM(7), CM(8)} are affecting changes for `test2` because after the edit, `test2` will call `MultiCounter.inc()`.

Testing Approach

In test-driven development, failing tests require program edits to correctly implement the functionality they define, and newly created tests represent new specifications that might require edits as well. Ideally, unit tests should encode every feature request, fault fix, or other improvement. The developer must modify the program to fulfill these encoded specifications. Work on such a task manifests itself as an edit to the program. When the test suite runs after an edit, some tests might exercise modified program elements denoted by atomic changes (for example, “invoke a modified method”). Every atomic change that affects a test—as well as all the changes on which it transitively depends—

are considered *covered changes*. Conversely, the phrase *changes not covered* refers to changes in an edit that don't affect any test of the suite. We can group the changes not covered by corresponding program elements (because each atomic change denotes some program element) and speak of those elements as *additions, changes, or deletions*.

Newly added tests that cover changes not yet covered elsewhere are called *effective tests*, whether they ran before or after the developer edited the program (that is, as in test-first or test-last methodologies). An effective test renders unexpected side effects from the edit much less likely. If newly added tests don't cover any changes, they're called *unrelated tests*; such tests are still useful in a global sense because they can reveal or protect against faults in other parts of the program not caused by the ongoing edit.

Traditionally, test coverage is taken to mean “a measure of the proportion of a program exercised by a test suite” (<http://encyclopedia.thefreedictionary.com/Test+coverage>), but this definition is too broad to be of much guidance to developers. Writing tests can quickly become daunting; 100 percent test coverage for an entire application is often an unattainable goal, and there's no way to claim that a sufficient number of tests has been written. By contrast, change coverage is a measure of the proportion of atomic changes comprising the difference between two program versions exercised by a test suite. The greatest benefit of change coverage is that it gives quantitative estimation of full coverage. Developers can justifiably claim to have finished their job when they've implemented the desired functionality and all their changes are covered via tests. The use of change coverage in test-driven development is called change-centric test development, as illustrated by the example in Figure 1. Developers using change-centric test development can adopt a completion condition that indicates not only whether the implemented functionality fulfills the requirements specified in the tests but also that the test suite covers all of an edit's effects. Moreover, developers can use the change-centric testing methodology in a test-last development process in which they must create specific tests to cover the edit.

Figure 2 depicts two possible development cycles, along with individual activities and decisions. After developers accept tasks, they must write tests and apply various changes to the program to implement the described improvements. During this activity, developers can run the test suite several times and correct any failures that occur. Even when all tests pass, the tasks are only con-

sidered completed if the tests cover all changes. If some changes aren't covered, developers explore them and the denoted program elements—specifically, additions and changes—that the new tests will have to exercise. (Deletions aren't coverable by any tests and are thus ignored.) Then, developers create new tests and run the test suite again. This process repeats until all changes are covered and the task can be declared completed.

Tool Support

To support change-centric test development as depicted in Figure 2, a tool must augment the feedback the developer receives to flag any changes not covered. This is necessary to establish the boundary condition that tells developers when they've done a good job in providing tests that protect against unexpected side effects.

Standard tools for unit testing communicate the test outcome with a simple metaphor: a red bar when a test fails or crashes and a green bar when all existing tests pass. Our tool, JUnitMX, extends this concept by introducing a new possible result, a yellow bar. This bar appears if all the tests pass but change impact analysis reveals changes not covered by the tests. The green bar appears only when all tests pass and the test suite covers every developer-applied change. The red bar holds the same meaning. With this simple extension, our tool confirms that the test suite passes and that no unexpected effects due to the edit have occurred.

Hands-On Scenario

To illustrate how JUnitMX supports change-centric test development, consider a hypothetical scenario, using the example in Figure 1. Assume that a developer is working on the code from the running example inside the Eclipse Java Development Tooling (www.eclipse.org/jdt) and wants to extend it to count multiple values. She synchronizes the code with the version control system to ensure that she's working on the latest version of the example—say, V1. The developer uses a single passing test, `test1`, which consists of two classes, `Counter` and `MultiCounter`, where `MultiCounter` is a subclass of `Counter`.

Working in a test-first manner, the developer adds a new test, `test2` (AM(1), CM(2), LC(3)), to class `Tests`, to drive the development's desired functionality. The new test asserts that each `Counter` is increased when a program calls method `inc()` on a `MultiCounter`. To compile `test2`, the developer must define a constructor of `MultiCounter` (AM(5), CM(6)) that accepts an array of `Counter` objects to manipulate and add a new field `counter` (AF(4)) to store them. The code then compiles, yielding V2,

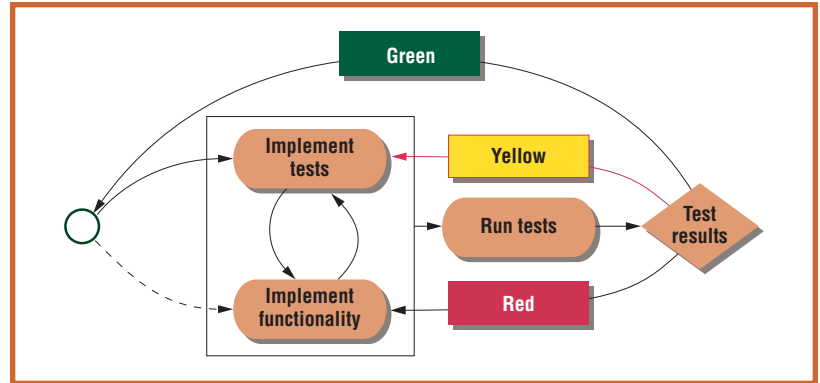


Figure 2. Change-centric development cycle. The solid arrow emanating from the start circle (on the left) shows the test-driven development process; the dashed arrow from the start circle shows the test-last development process. Red indicates test failures; yellow, tests missing; and green, all tests pass.

which, when run, results in the expected failure of `test2`. Satisfying `test2` requires the developer to redefine method `inc()` in class `MultiCounter` (AM(7), CM(8), LC(9), LC(10)), so that all counters increase. The change results in V3 of the code.

Although `test2` now passes, the result of running the test suite is a yellow bar, not a green one. Apparently, the current test suite doesn't cover some of the atomic changes, so additional tests must cover the denoted program elements. Indeed, a lookup change (LC(9)) associated with program element `MultiCounter.inc()` isn't yet covered by any test, as shown in Figure 3.

In the JUnitMX user interface, the developer can view the changes not covered by clicking on the Untested Changes tab, where changes are organized by type. The developer can inspect each change further by single-clicking to compare the denoted program element's current version with its previous one. The lower pane in Figure 3 acts as a comparison view, showing the previous version on the left and the current version on the right. Because the developer added the method, the left half is empty and the current version's source code appears on the right. Double-clicking on the change opens an editor for the source file in question, specific to the denoted program element.

After inspecting the changes not covered, the developer proceeds to write `test3` (AM(11), CM(12), LC(13)) to cover lookup change LC(9), yielding V4 of the code. The new test exercises method `inc()` on a `MultiCounter` object referred to by the declared type `Counter` and specifies how the interplay between `inc()` and `getSum()` should work.

The benefit of targeted tests is evident because the newly added test fails. Running the test suite now results in a red bar—the test has exposed a fault in the logic. Indeed, the newly covered lookup change points to a place where the code edit had side effects that produced the wrong result.

In response to the failing test, the developer proceeds to fix the exposed fault. In V5, she introduces

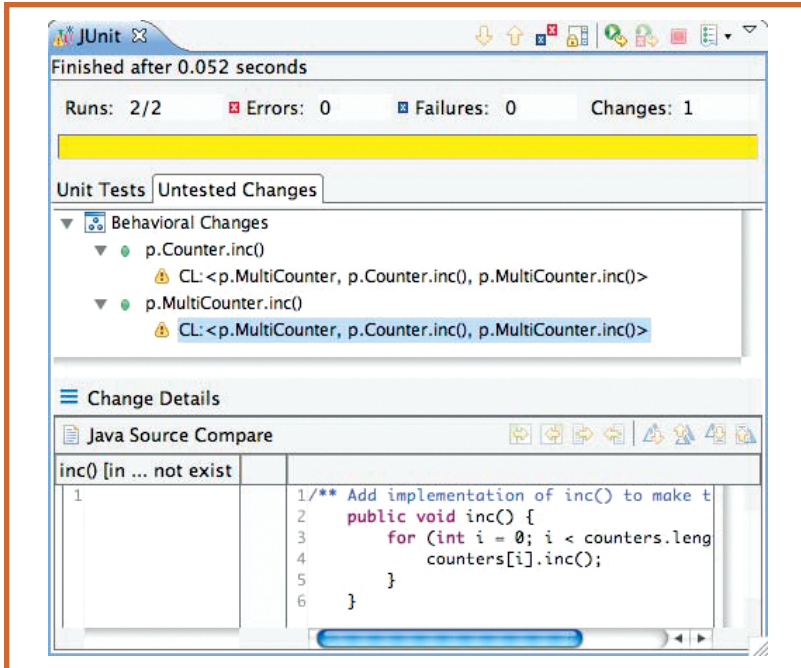


Figure 3. JUnitMX screenshot. Changes not covered by any test appear in a tree view.

a redefinition of method `getSum()` in the class `MultiCounter` (AM(14), CM(15), LC(16), LC(17)), establishing the correct interplay between `inc()` and `getSum()`. Now, all the tests succeed; moreover, the test suite covers all the changes, and the bar finally turns green. The developer can feel confident that her code is free from any unanticipated effects.

Behind the Scenes

We built JUnitMX as an extension of the JUnit Eclipse plug-in, a tool with which many developers are already familiar. The JUnitMX user interface differs slightly by displaying the kinds and numbers of changes not covered and provides an extra tree view to browse them. Given this information, a developer can focus on where to start creating the needed tests. A click on any of the changes opens the source code editor and leads the developer directly to the denoted program element.

JUnitMX (<http://prolang.cs.vt.edu/projects.php>) hooks into the execution of a JUnit test suite by adding pre- and postprocessing phases—specifically, it combines the results from two modules developed at Rutgers University, Chianti and Dila, with those from JUnit to compute the change coverage information. Chianti is a tool for change impact analysis that computes the atomic changes comprising an edit. Classes are instrumented as they're loaded by a custom class loader provided by Dila, a library that uses bytecode utilities from the WALA project (<http://sourceforge.net/projects/wala>). This simple mechanism allows for an efficient building of dynamic application

call graphs. Each JUnit test run constructs its call graph and produces its outcome. In a postprocessing phase, Chianti performs the actual change impact analysis, and Dila calculates the change coverage and test suite effectiveness.

A Case for Change-Centric Test Development

To investigate our hypotheses that developers write too few tests to sufficiently cover their edits and that these tests are sometimes unrelated to the edit, we compared development activities over multiple releases of JUnit3. The results of our study not only support our hypotheses but also suggest the inadequacy of solely using branch coverage—a practical and popular coverage metric—as an achievable boundary for creating a quality test suite.

The popular unit testing framework JUnit3 is a nontrivial program with multiple years of development history in a publicly accessible repository, including a suite of unit tests. Its creators developed JUnit3 in bursts, starting from 2001 over multiple years. We defined successive program versions of JUnit3 using two-week intervals over several years; we considered only those versions with more than 20 atomic changes, resulting in 13 valid version pairs. We ran corresponding test suites with JUnitMX to capture the following data:

- *Size versus changes.* We calculated each version's program size by using the aggregate number of fields, methods, and classes. Atomic changes correspond to program elements, so these two measures are comparable. Our goal was to compare program size with the number of coverable changes to illustrate that the extent of an edit isn't correlated with program size. Note that we counted constructors as methods.
- *Coverage.* We compared the percentages of achieved branch and change coverage to illustrate the differences in their ability to measure the adequacy of testing an edit.
- *Effectiveness.* We compared the test suites' growth over time to test effectiveness (that is, what percentage of newly added tests cover changes not already covered by existing tests). Our goal was to show that newly added tests don't necessarily cover the developer's edit adequately.

Note that we couldn't capture deleted static or nonstatic class initializers because no edge in the edited program's call graphs can witness their deletion. Thus, we didn't include them in the set of potentially coverable changes.

Our feasibility experiment results yielded interesting and surprising insights. Three of the 13 versions experienced significantly large edits, with many changes (for example, more than 3,400, roughly 1,400, and roughly 800) but no significant change in program size. Indeed, program size remained fairly constant (approximately 750 program elements) over all version pairs examined. This indicates that several existing features of JUnit3 changed, rather than an increase in overall functionality. Although the number of applied changes varied greatly across all versions, our results demonstrate that program size isn't correlated with the extent of a program edit.

In comparing branch coverage with change coverage across these program versions, we found that branch coverage barely varied (28 to 34 percent), whereas the change coverage ranged from 0 to 68 percent. Thus, for this benchmark, branch coverage appears not to provide much guidance for the development of tests that reduce the likelihood of an edit introducing faults into the program. Conversely, change coverage indicates how much of the edit has been covered by tests, yielding a direct measure of additional tests necessary to validate edit effects. Moreover, 100 percent change coverage for this benchmark seems achievable with a reasonable amount of work. This ability to guide developer actions toward a highly desirable goal is a major strength of the change coverage metric. In comparing change coverage with the size of the edit for the same program version, we found no correlation between measures—that is, we found large and small edits with many covered changes and others with no covered changes.

Finally, we compared the test suite's growth with the effectiveness of newly added tests. We measured both metrics as percentages, so the more that the newly added tests covered additional changes, the higher the effectiveness. Of the 13 version pairs, four achieved 100 percent test effectiveness (that is, all added tests covered changes not already covered by existing tests), three had partial effectiveness (that is, 42 percent, 60 percent, and 92 percent), and six had no test suite growth. However, even the 100 percent effective test suites only achieved 12 to 68 percent change coverage, which ultimately supported our original hypotheses.

Although our study doesn't prove that all developers write tests blindly, we've shown that it's difficult to predict and test an edit's effects. We've also shown that change coverage provides a reasonable—and achievable—

About the Authors




Jan Wloka is a former postdoctoral researcher in the Programming Languages Research Group (PROLANGS) at Rutgers University, currently working as a software engineer at the IBM Rational Zurich Lab. His research interests include change-aware development tools and advanced refactoring techniques, aiming for practical applications of change impact analysis for tool-supporting software evolution. Wloka has a PhD in computer science from the Technical University of Berlin. He's a member of ACM SigSoft. Contact him at jan_wloka@ch.ibm.com.

Einar W. Host is a former PhD fellow at the Norwegian Computing Center, currently working as a senior knowledge engineer at Computas AS. His research interests include the formal and informal semantics of computer programs. Host has a MSc in computer science from the University of Oslo. Contact him at einar.host@computas.com.



Barbara Ryder is the J. Byron Maupin Professor of Engineering and head of the Department of Computer Science at Virginia Tech. Her research interests include design and evaluation of static and dynamic program analyses for object-oriented systems for use in practical software tools. Ryder has a PhD in computer science from Rutgers University. She is an ACM Fellow and former member of the Editorial Board of IEEE Transactions on Software Engineering. Contact her at ryder@cs.vt.edu.

goal to motivate the development of effective tests.

For test-driven development, our change coverage metric can serve as an upper boundary, indicating whether a developer did the simplest thing that could possibly work. Changes not covered indicate that an implementation provides more functionality than specified by the tests or a weak test suite that incompletely specifies functionality. Using our change coverage metric and attaining a green bar in JUnitMX shows a developer that he or she did a good job in protecting the program against inadvertent side effects caused by an edit. 

References

1. L. Copeland, *A Practitioner's Guide to Software Test Design*, Artech House, 2004.
2. K. Beck, "Aim, Fire," *IEEE Software*, vol. 18, no. 5, 2001, pp. 87–89.
3. M. Fowler et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
4. P. McBreen, *Questioning Extreme Programming*, Addison-Wesley Professional, 2002.
5. X. Ren et al., "Chianti: A Tool for Practical Change Impact Analysis of Java Programs," *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems and Applications (OOPSLA)*, ACM Press, 2004, pp. 432–448.
6. B.G. Ryder and F. Tip, "Change Impact Analysis for Object-Oriented Programs," *Proc. 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ACM Press, 2001, pp. 46–53.
7. R. Arnold and S. Bohner, *Software Change Impact Analysis*, Wiley-IEEE CS Press, 1996.
8. X. Ren, O. Chesley, and B.G. Ryder, "Crisp, A Debugging Tool for Java Programs," *IEEE Trans. Software Eng.*, vol. 32, no. 9, 2006, pp. 1–16.