Tool Support for Change-centric Test Development

How do developers know they have done a good job?

Jan Wloka, Rutgers University, USA, *Einar W. Høst*, Norwegian Computing Center, Norway,
and *Barbara G. Ryder*, Virginia Tech, USA

***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 method for applying change impact analysis to test-driven development, to provide developers with quantitative feedback of test coverage of their changes. This information can be used to meet defined coverage goals or to help generate new tests to reveal unanticipated changes effects. The approach, called *change-centric test development*, is tool supported; a typical scenario shows the effectiveness of our tool JUNITMX in a practical feasibility study.**

***Index Terms*— Change Impact Analysis, Test-driven Development, Unit Testing.**

I. INTRODUCTION

Software testing aims at validating the correctness, completeness and quality of developed computer software [3]. A test makes the correctness of a program against its specification more likely by executing the program and comparing the outcome against expectations to find faults. A good test has a high probability of finding an as yet undiscovered fault. There are various testing levels used for different purposes within a development process. For example, unit and integration testing enable developers to test an implementation and its effects on existing functionality. In test-driven development, a unit test acts as a specification for a specific functionality before it is implemented. It guides the developer to implement only the code necessary to pass the test [1]. Moreover, a set of unit tests is a prerequisite for any refactoring activity [4].

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. Implementing the "simplest thing that could possibly work" [5], results ideally in a test suite that reveals any effect of the added code on existing functionalities. However, what if the test coverage of the unchanged system parts is low and the test suite results in a *green bar*? Since successful tests do not show the absence of faults, but rather the inability of the test suite to find any fault, the *green bar* in the developer's unit testing tool may leave her feeling over-confident. Moreover, the developer might not have implemented the "simplest thing that could possibly work", thus additional tests are needed to validate her entire edit. In both cases, the developer has written tests "blindly", that is by missing possible side effects or by being unable to verify that her edit has caused no unexpected alteration in the system's behavior.

This paper presents an approach to test development that uses change impact analysis [9], [8] to guide developers in the creation of new unit tests. This analysis specifies those changes introduced by the developer that are not covered by the current test suite, and hence indicates that there are tests 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 the system's behavior are not covered by the test suite. The developer can decide if she needs to add or extend a test to cover every effect on existing code. Even if the test suite covers all the changes introduced and all the tests pass, there still may be faults in the system; however, such coverage makes it more likely that new faults have not 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* through use of our tool, JUNITMX. Also, we discuss a feasibility study that shows the potential benefit of using our approach in current software development practice.

II. ILLUSTRATING EXAMPLE

We use the Java program in Figure 1 to illustrate change-centric test development, to describe a *hands-on* scenario and to show the support offered by JUNITMX. The example is a simple counter application that can increase, store and return a single integer value. The application will be extended to a multi-counter that manages several instances of a counter. On the left-hand-side of the figure is the actual program code and on the right-hand-side, the associated test suite. Program changes are indicated with annotated boxes. The original program, *version V1*, consists of all the code, except that shown in boxes. Each of the three subsequent program versions are shaded with gray labels. For example, *version V2* is constructed from *version V1* by applying all changes whose boxes are within the label *V2*, *version V3* is constructed similarly from *version V2* and so forth.

III. CHANGE-CENTRIC TEST DEVELOPMENT

When developing unit tests for improved or new functionality, a developer does not always know whether she has done a good job. Two challenging aspects of writing good unit tests are to ensure that (i) involved program elements can be exercised by the test and (ii) all effects on other functionalities are covered by the test. This section introduces the specific change impact analysis used to calculate a new test coverage metric that indicates whether a developer's changes are sufficiently covered by a test suite.

A. Applying Change Impact Analysis

Change impact analysis is a technique whose goal is to predict the possible effect of a program edit on a code base [9], [6], [8]. Consider the following scenario. A developer has written several tests and modifies a working program, in order to add some new functionality, yielding an edited version of the program. She has a mental picture of the program parts that may be affected by her edit. If change impact analysis is applied, then its results

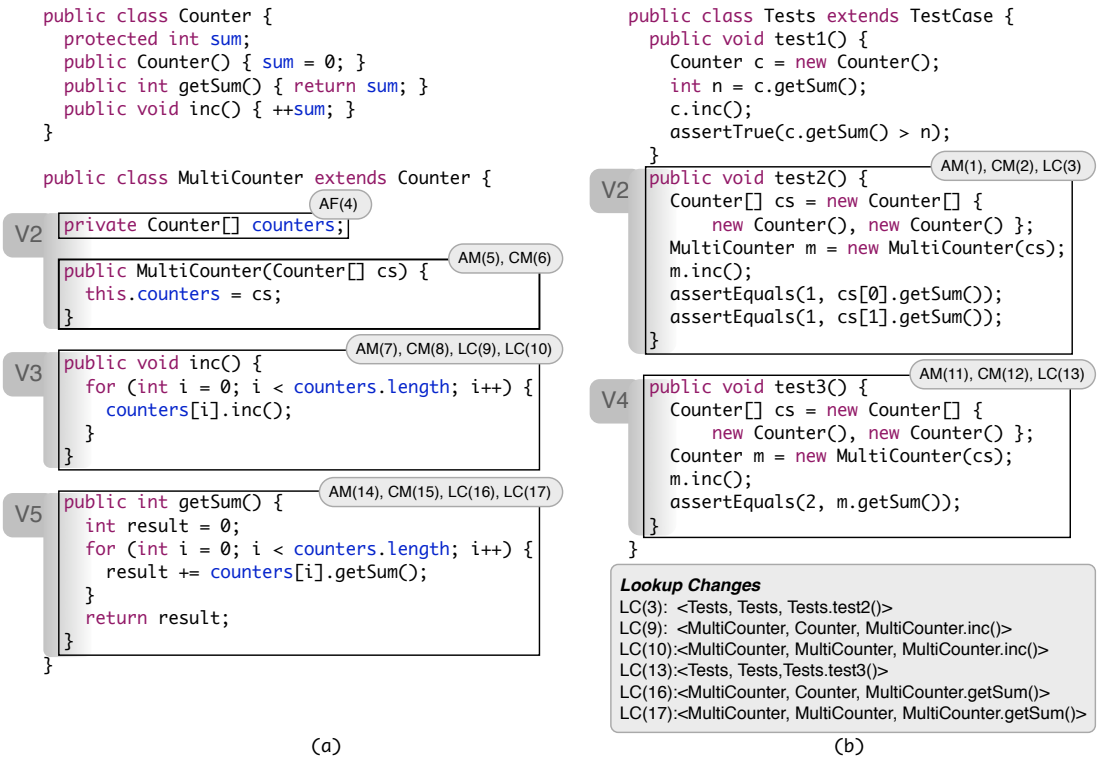

Fig. 1. **(a)** Original and edited version of the example program. The original program consists of all program fragments *except* those shown in boxes. The edited program is obtained by adding all boxed code fragments. Each box is annotated with the IDs of the corresponding atomic changes. **(b)** Tests associated with (all versions of) the example program and lookup changes (LC) describing effects to dynamic dispatch.

can be used to ascertain whether the developer's expectations are fulfilled. Otherwise, if unexpected effects are predicted, the developer may want to explore the results more fully to make sure no unintended outcomes were introduced by her edit. The specific change impact analysis used combines static (i.e., compile-time) analysis and profiling to predict the method-level effects of an edit on the behavior of a Java program. Each Java program is assumed to have an associated test suite; thus the impact induced by the edit can be reported as possible behavioral changes to tests in the suite [9], [8], [7].

An *edit* or the textual difference between two program versions, can be decomposed into a set of *atomic changes* or 'smallest changes' to a program. Adding a method (AM), changing the code in a method (CM), or adding a new field to a class (AF) are examples of atomic changes. The element in the program that is affected by a change is called *denoted program element*. The complete set of atomic changes we are using is presented in [8]. 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 [7], [2]. These *structural dependences* do not capture all effects of an edit on program behavior. Certain changes *indirectly* impact program behavior. For example, the addition of a virtual method may give rise to changes in dispatch behavior, and 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. Such effects are captured by *mapping dependences* between changes. Unlike structural dependencies,mapping dependences are symmetrical, because it is not possible to apply one without the other.

In Figure 1, the developer adds a constructor to class `MultiCounter` as part of the edit that leads to *version 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. Moreover, CM(6) cannot be applied without AM(5), which makes it dependent on AM(5). Just as the added field `counters` (AF(4)), which is structurally required by CM(6).

Many kinds of edits may alter the existing dynamic dispatch behavior of a Java program, such as adding an overriding method in a subclass, or changing visibility from *private* to *public*. A lookup change (LC) represents the effect of an edit 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()`. LC(9) corresponds to the redirected dispatch of `MultiCounter` objects referred to by a `Counter` reference, in a call of `inc()` which after the code edit, will be directed to `MultiCounter.inc()` rather than to `Counter.inc()`. All of the LC changes corresponding to the edits in Figure 1, are shown in the shaded box in the right-hand corner of the figure.[1]

After the dependencies have been computed, the test suite is run on the edited program and profiles are collected to obtain the calling structure of each test. By mapping method-level atomic changes to a tests calling structure (e.g., a call graph), the analysis computes the set of *affected tests* [8], tests whose

[1]The first element in each LC change is the instance type of the receiver object; the second element is the static (compile-time) type of the method invocation, and the third is the actual method defintion.

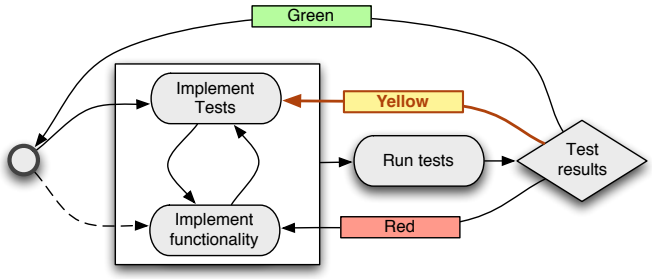

Fig. 2. Development cycle in change-centric test development: **red**: test failures, **yellow**: tests missing, some changes in the program edit are not covered, **green**: all tests pass and all changes are covered. Note: the solid arrow eminating from the start circle (on the left) shows the test-driven development process. The dashed arrow from the start circle shows the test-last development process.

behavior may be different after the edit. In Figure 1, when the program transitions from *version V2* to *version V3*, `test2` is affected, but `test1` is not. Intuitively, this is because `test2` calls `MultiCounter.inc()` after the edit, whereas it called `Counter.inc()` in *version V2*. By contrast, the same set of (unchanged) methods is called by `test1` before and after the edit [8].

For each affected test, the analysis is able to isolate those parts of the edit which may have affected it, called its *affecting changes*. Affecting changes are the part of the edit that could be mapped to the test's calling structure. Considering the edit from *version V2* to *version V3* in Figure 1; for example, {AM(7), CM(8)} are affecting changes for `test2` since after the edit, `test2` will call `MultiCounter.inc()`.

Thus, change impact analysis allows a developer to check if she has anticipated the overall effect of her edit correctly. The following section describes how a developer can judge if the test suite is sufficient for good test coverage after an edit, by using change impact analysis.

B. The Change-centric Test Development Approach

In test-driven development the goal requiring program changes is defined by both the newly created and failing tests. Ideally every feature request, a bug fix, or other improvement should be encoded with unit tests. A developer must modify the program to fulfill the specification encoded by the tests.

¡¡¡¡¡¡¡ task-aware-test-development.tex

======= ¿¿¿¿¿¿¿ 1.27

Work on such a task manifests itself as an edit to the program, (i.e., a set of *atomic changes*). When the test suite is run again, some tests ¡¡¡¡¡¡¡ task-aware-test-development.tex may exercise modified program elements corresponding to atomic changes. The tests that exercise these elements are affected by the changes, which may or may not alter the outcome of the test. Each atomic change that affects a specific test is covered by this test, irrespective of whether it alters the test's outcome. Such changes, called *covered changes*, affect (and may alter) the outcome of a test in a test suite. Conversely, the term *uncovered changes* is used for changes in an edit that do not affect any test of the test suite.

Newly added tests that cover (as yet not covered) changes are called *effective tests*, regardless whether they are developed before or after the program has been changed (i.e., in *test-first* or *tret-last* mode). An effective test renders unexpected side effects of the edit less likely, in contrast to newly added tests that do *not* cover any change, called *unrelated tests*. they do not exercise any changed program element. Unrelated tests are still useful in a global sense — they may reveal or protect against errors in other parts of the program — but these errors will not be due to any effect caused by the ongoing edit. are not related to the current task of the clean separation of tasks, the developer writing such unrelated tests, and define a new development. ======= may exercise modified program elements denoted by atomic changes, e.g., invoke a modified method. Change impact analysis can be use to obtain the affecting changes for each test. Every atomic change that affects a test and also all the changes on which it depends on, following dependence relationships transitively, is considered as a *covered change*. Conversely, the term *uncovered changes* is used for changes in an edit that do not affect any test of the suite. Moreover, the uncovered changes can be grouped by their impact on program elements, since we know every atomic change that denotes an element. We use ADDITIONS, CHANGES, and DELETIONS for classifying modified program elements.

Newly added tests that cover (as yet not covered) changes are called *effective tests*, regardless whether they are developed before or after the program has been changed (i.e., in *test-first* or *tret-last* mode). An effective test renders unexpected side effects of the edit less likely, in contrast to newly added tests that do *not* cover any change, called *unrelated tests*. Unrelated tests are still useful in a global sense — they may reveal or protect against errors in other parts of the program — but these errors will not be due to any effect caused by the ongoing edit. ¿¿¿¿¿¿¿ 1.27

Traditionally, *test coverage* is taken to mean "a measure of the proportion of a program exercised by a test suite."[2] While this is a useful measure of overall program health, it is too broad to be of much guidance for the developer. Hence writing tests can quickly become daunting: 100% test coverage for the entire application is an unattainable goal, and there is no way to reasonably claim that a sufficient number of tests has been written. This motivates our change coverage:

> **Definition:** *Change coverage* is a measure of the proportion of atomic changes comprising the difference between two program versions, that is exercised by a test suite.

¡¡¡¡¡¡¡ task-aware-test-development.tex The greatest benefit of change-centric test coverage is that it enables quantitative estimation of full coverage of a developer's changes. This is crucial, for the developer to focus an *achievable goal* for her current task: writing tests that provide reasonable coverage for a program edit. Developers justly can claim to have finished their job when the desired functionality is implemented, and all changes are covered with tests. We call this approach *change-centric test development*.

Change-centric test development is achieved by integrating a change impact analysis into the execution of a test suite. The analysis is used to measure the change-centric test coverage each time the suite is run. Given this coverage information, developers can be guided in their standard test-driven development cycle *test – code*, by adopting a completion condition that indicates not only whether the implemented functionality fulfills the requirements

[2]Definition taken from the Free Online Encyclopedia; http:// encyclopedia2.thefreedictionary.com/Test+coverage.

specified by the tests, but also that all effects of an edit are covered by the tests. Moreover, the change-centric testing methodology also can be utilized in a test-last development process, in which the developer is asked explicitly to create specific tests that cover her edit.

Figure 2 depicts two possible development cycles, showing individual activities and decisions. Examining the test-driven process shown, after a developer has *accepted a task*, she *writes tests* and *applies various changes* to the program necessary to implement the described improvement. During this activity she may *run the test suite* several times and correct failures that occur. When *all tests pass*, the task is only completed if all changes are covered by the tests. In case of *uncovered changes*, the developer explores the kind of changes (i.e., additions, removals, or modifications) that are uncovered and the denoted program elements that have to be exercised by new tests. Then, she *creates the new tests* and runs the test suite again. This process is repeated until all changes are covered and the ======= The greatest benefit of change coverage is that it enables quantitative estimation of full coverage of a developer's changes. This is crucial, for the developer to focus an *achievable goal* for her current task: writing tests that provide reasonable coverage for a program edit. Developers justly can claim to have finished their job when the desired functionality is implemented, and all changes are covered with tests. We call this approach *change-centric test development*.

Change-centric test development is achieved by integrating a change impact analysis into the execution of a test suite. The analysis is used to measure the change coverage each time the suite is run. Given this coverage information, developers can be guided in their standard test-driven development cycle *test – code*, by adopting a completion condition that indicates not only whether the implemented functionality fulfills the requirements specified by the tests, but also that all effects of an edit are covered by the tests. Moreover, the change-centric testing methodology also can be utilized in a test-last development process, in which the developer is asked explicitly to create specific tests that cover her edit.

Figure 2 depicts two possible development cycles, showing individual activities and decisions. Examining the test-driven process shown, after a developer has *accepted a task*, she *writes tests* and *applies various changes* to the program necessary to implement the described improvement. During this activity she may *run the test suite* several times and correct failures that occur. When *all tests pass*, the task is only completed if all changes are covered by the tests. In case of *uncovered changes*, the developer explores the kinds of changes (i.e., ADDITIONS, and CHANGES) that are uncovered and the denoted program elements that have to be exercised by new tests. DELETIONS are not coverable by any tests and thus are ignored. Then, she *creates the new tests* and runs the test suite again. This process is repeated until all changes are covered and the ¿¿¿¿¿¿ 1.27 task can be declared *completed*.

IV. TOOL-SUPPORT FOR CHANGE-CENTRIC TEST DEVELOPMENT

To support change-centric test development as illustrated in Figure 2, a tool must augment the feedback given to the developer when a test suite is run to inform her of any uncovered changes. This is necessary in order to establish the boundary condition that tells the developer when she has 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. They display a *red bar* whenever a test failed or crashed and a *green bar* when all existing tests pass. Our tool, JUNITMX, extends this concept by introducing a new possible result, a *yellow bar*. The yellow bar is shown if all the tests pass, but change impact analysis reveals changes that are not covered by the tests. This leads to a redefinition of the meaning of the green bar as well: The green bar is shown only when all tests pass *and* every change applied by the developer is covered by the test suite. The meaning of the red bar is same as before. With this simple extension, the tool confirms that the test suite passes and verifies that the developer's changes have not resulted in any side effects not covered by the test suite.

A. A Hands-on Scenario

To illustrate how JUNITMX supports change-centric test development, consider a hypothetical scenario, using the example in Figure 1. Assume that the developer is working on the code from the running example inside the *Eclipse JDT*[3], and wants to extend it to enable the counting of multiple values. The developer synchronizes her code with the version control system to ensure that she is working on the latest version of the example (i.e., *version V1*). *Version V1* consists of two classes `Counter` and `MultiCounter`, where `MultiCounter` is a subclass of `Counter`. This version of the code is tested by a single passing test, `test1`.

Working in a test-first manner, the developer adds a new test, `test2` (AM(1), CM(2), LC(3)), to class `Tests`, in order to drive the development of the desired functionality. The new test asserts that when method `inc()` is called on a `MultiCounter`, then each `Counter` is indeed increased. To get `test2` to compile, 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 `counters` (AF(4)) to store them. The code then compiles yielding *version V2* 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 are increased. The change results in *version V3* of the code.

While `test2` now passes, the result of running the test suite is a yellow bar, not a green one. Apparently, there are atomic changes that are not covered by the current test suite. Hence there are denoted program elements that need to be covered by additional tests. Indeed, there is a lookup change (LC(9)) associated with program element `MultiCounter.inc()` that is not yet covered by any test.

In the JUNITMX user interface, the uncovered changes can be viewed by clicking on the *Uncovered changes* tab. The changes are organized by type; each change can be inspected further by single-clicking to compare the current version of the denoted program element with its previous one.

The lower pane acts as a comparison view, showing the previous version on the left, and the current version on the right. Because the method was added, the left half is empty and the source code for the current version is shown on the right. Double-clicking on the change opens an editor for the source file in question, focused on the denoted program element.

[3]The Eclipse Java Development Tooling http://www.eclipse.org/jdt

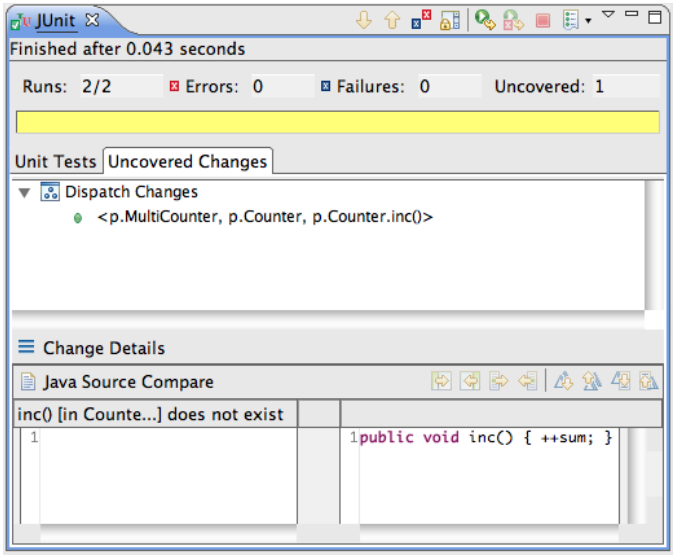

Fig. 3. Uncovered changes are shown in a tree view.

Inspecting the uncovered changes, the developer proceeds to write a test, `test3` (AM(11), CM(12), LC(13)), to cover the uncovered lookup change LC(9). This yields *version 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 from the fact that the newly added test fails. Running the test suite now results in a red bar — a fault in the logic has been exposed. Indeed, the uncovered lookup change pointed to a place in the code 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 *version V5*, a redefinition of method `getSum()` in the class `MultiCounter` is introduced (AM(14), CM(15), LC(16), LC(17)), establishing the correct interplay between `inc()` and `getSum()`. Now all the tests succeed; moreover, there are no more uncovered changes. The bar finally turns green. The developer can justifiably feel confident that her code is free from unanticipated effects.

B. Behind the Scenes

The tool JUNITMX is built as an extension to the *JUnit Eclipse* plug-in.[4] Developers already familiar with *JUnit* and the *Eclipse JDT* can build on knowledge with a familiar tool when using JUNITMX. The UI that displays the test outcomes looks like the well-known *JUnit* plug-in, with some additional information. The JUNITMX UI displays the kinds and numbers of uncovered changes and provides an extra tree view to browse the uncovered changes. Given this information, a developer can focus on where to start creating 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 hooks into the execution of a *JUnit* test suite and adds pre- and post-processing phases. JUNITMX combines the results from two modules, *Chianti* and *Dila*, with those from *JUnit* to compute the change coverage information. *Chianti*[5] is a tool for change impact analysis that computes the atomic changes comprising an edit. Classes are instrumented as they are loaded by a custom class loader provided by *Dila*[6], a library that uses bytecode utilities from the *WALA* project.[7] This simple mechanism allows for an efficient building of dynamic application call graphs. Each *JUnit* run of a test constructs its call graph and produces its outcome. In a *post-processing* phase, the actual change impact analysis is performed, and change coverage and test suite effectiveness are calculated.

V. A Case for Change-centric Test Development

In practice developers write tests "blindly", that is, they have an unclear perception of which parts of their program need to be tested after an edit. This is manifested in two ways: (i) developers write too few tests to sufficiently cover their edit, and (ii) the written tests are sometimes unrelated to the edit. To gather evidence to support this claim, we compared development activities over multiple releases of *JUnit3*.[8] The results of the study support the claim, and provide evidence that (i) and (ii) can be demonstrated for a real open source application. Further, the results show the inadequacy of branch coverage, a practical and popular coverage metric, as an achievable boundary that motivates developers to write effective tests. In contrast, our approach alerts developers to the potential of introducing bugs with their edits, and guides them to write tests that reduce this potential effectively.

A. Goals and Methodology

Our major goal in this study was to show the potential utility of the change coverage metric. To this end, three different kinds of data were collected to show that (i) there are many uncovered changes in the development of a widely-used software tool, (ii) a trusted, existing coverage metric is a poor predictor of change coverage, and (iii) tests are actually written "blindly" by developers.

The popular unit testing framework *JUnit3* is a non-trivial program with multiple years of development history. It is developed in Java, has a publicly accessible repository, and comes with a suite of unit tests. *JUnit3* was developed in bursts starting from 2001 over multiple years including 2002 and 2004. We defined successive program versions of *JUnit3* using two week intervals over several years; only versions with more then 20 atomic changes were considered, resulting in 13 valid version pairs from *JUnit3* development. ¡¡¡¡¡¡¡ evaluation.tex Corresponding test suites were run with JUNITMXto capture the following data. ======= Corresponding test suites were run with JUNITMX to capture the following data. ¿¿¿¿¿¿¿ 1.25

- **Size vs. Changes.** Program size of each version was calculated using the aggregate number of fields, methods and classes. Atomic changes correspond to program elements, so these two measures are comparable. The goal was to compare program size with the number of coverable changes[9]

[4] A more detailed description of JUNITMX is available in the Appendix (i.e., http://thiswillbefixedinfinalversion).

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

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

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

[8] http://www.junit.org

[9] Two kinds of changes cannot be considered in our change coverage data. Deleted static and non-static class initializers are not captured by our analysis, because there is no edge in the call graphs of the edited program that witnesses the deletions of the class initializers. Thus, they are not included in the set of potentially coverable changes.

to illustrate that the extent of an edit is not correlated with program size. Note that constructors are counted as methods.
- **Coverage.** The percentages of branch coverage and change coverage achieved are compared to demonstrate that a widely-used test coverage metric is not helpful in motivating and guiding developers to write effective tests during the development process.
- **Effectiveness.** The growth of the test suites over time is compared to test effectiveness, (i.e., what percentage of newly added tests cover changes that are not covered already by existing tests). The goal is to show that newly added tests do not cover the developer's edit adequately.

B. Evidence in the End

The feasibility experiment results yielded interesting and surprising insights.[10]

Three of the 13 versions experienced significantly large edits, with many changes (e.g., >3400, ∼1400, ∼800), but no significant change in program size. Indeed, program size remained fairly constant (∼750 program elements) over all version pairs examined. This indicates that several existing features of *JUnit3* were changed, rather than overall functionality being increased. Even though the number of applied changes varied greatly across all versions, our results demonstrate that program size is not correlated with the extent of a program edit.

¡¡¡¡¡¡¡ evaluation.tex In comparing branch coverage with change-centric coverage across these program versions, we found that branch coverage barely varied (28-34%), whereas the change coverage ranged from 0 to 68%. Thus, branch coverage does not provide guidance for the development of tests to reduce the likelihood of an edit introducing bugs into the program. Conversely, change-centric 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% change-centric coverage seem to be achievable with a reasonable amount of work. This ability to guide developer actions towards a desirable goal is a major strength of the change-centric coverage metric. By comparing change coverage with the size of the edit (for the same program version), there was no correlation between these measures; that is, there were large and small edits with many covered changes and others with no covered changes.

Finally, we compared the growth of the test suite and the effectiveness of newly added tests. Both metrics are measured as percentages; thus the more additional changes covered by newly added tests, the higher the effectiveness. Of the 13 version pairs, there were four that achieved 100% test effectiveness (i.e., all added tests cover changes not already covered by existing tests), three that had partial effectiveness (i.e., 42%, 60%, 92%), and six with no test suite growth. However, even the 100% effective test suites only achieved 12-68% change coverage, evidence that developers write tests "blindly", as previously asserted. ======= In comparing branch coverage with change coverage across these program versions, we found that branch coverage barely varied (28-34%), whereas the change coverage ranged from 0 to 68%. Thus, branch coverage does not provide guidance for the development of tests to reduce the likelihood of an edit introducing bugs 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% change coverage seem to be achievable with a reasonable amount of work. This ability to guide developer actions towards a desirable goal is a major strength of the change coverage metric. By comparing change coverage with the size of the edit (for the same program version), there was no correlation between these measures; that is, there were large and small edits with many covered changes and others with no covered changes.

Finally, we compared the growth of the test suite and the effectiveness of newly added tests. Both metrics are measured as percentages; thus the more additional changes covered by newly added tests, the higher the effectiveness. Of the 13 version pairs, there were four that achieved 100% test effectiveness (i.e., all added tests cover changes not already covered by existing tests), three that had partial effectiveness (i.e., 42%, 60%, 92%), and six with no test suite growth. However, even the 100% effective test suites only achieved 12-68% change coverage, evidence that developers write tests "blindly", as previously asserted. ¿¿¿¿¿¿¿ 1.25

[10]More details about the study can be found in the online Appendix (i.e., http://thiswillbefixedinfinalversion).

VI. Conclusions

Although the study does not *prove* that all developers write tests "blindly", we have shown that it is difficult to predict the effects of an edit or to test them. For larger programs written by teams, a developer introduces behavior she wants preserved even after team members change the program; this can be assured by introducing the appropriate tests.

We have shown that change coverage provides a natural and achievable boundary to motivate the development of effective tests. We also presented JUnitMX, an extension to *JUnit*, that alerts developers to changes not covered ¡¡¡¡¡¡¡ conclusion.tex by existing tests. JUnitMXñot only helps developers to focus on the code that must be exercised to reduce the likelihood of introducing bugs with their edit, but it also offers ======= by existing tests. JUnitMX not only helps developers to focus on the code that must be exercised to reduce the likelihood of introducing bugs with their edit, but it also offers ¿¿¿¿¿¿¿ 1.13 a *concrete and achievable* goal for sufficient test coverage just by turning the *yellow bar* into a *green bar*. Particularly in test-driven development, our ¡¡¡¡¡¡¡ conclusion.tex change-centric test coverage can be used as an upper boundary indicating whether a developer did the "simplest thing that could possibly work". The lack of test coverage of some changes can not always be anticipated by the developer. These are an indication of a weak test suite that incompletely specifies the implemented functionality. Whereas the *green bar* reflects not only that the developer has completed the functionality's implementation, but also that she has done it well. ======= change coverage can be used as an upper boundary indicating whether a developer did the "simplest thing that could possibly work". The lack of test coverage of some changes can not always be anticipated by the developer. These are an indication of either an implementation that provides more functionality as specified by the tests or a weak test suite that incompletely specifies the expected functionality. Whereas the *green bar* in JUnitMX reflects not only that the developer has completed implementation, but also that she has done it well. ¿¿¿¿¿¿¿ 1.13

REFERENCES

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

[2] Ophelia Chesley, Xiaoxia Ren, and Barbara 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.

[3] Lee Copeland. *A Practitioner's Guide to Software Test Design*. Number 158053791x. Artech House Publishers, Hardcover edition, January 2004.

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

[5] Pete 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. Chesley, and B.G. Ryder. Crisp, a debugging tool for java programs. *IEEE Transactions on Software Engineering*, 32(9):1–16, September 2006.

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

[9] B. G. Ryder and F. Tip. Change Impact Analysis for Object-oriented Programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53, New York, NY, USA, 2001. ACM Press.