Understanding Myths and Realities of Test-Suite Evolution

Leandro Sales Pinto
Politecnico di Milano
pinto@elet.polimi.it

Saurabh Sinha
IBM Research – India
saurabhsinha@in.ibm.com

Alessandro Orso
Georgia Inst. of Technology
orso@cc.gatech.edu

ABSTRACT

Test suites, once created, rarely remain static. Just like the application they are testing, they evolve throughout their lifetime. Test obsolescence is probably the most known reason for test-suite evolution—test cases cease to work because of changes in the code and must be suitably repaired. Repairing existing test cases manually, however, can be extremely time consuming, especially for large test suites, which has motivated the recent development of automated test-repair techniques. We believe that, for developing effective repair techniques that are applicable in real-world scenarios, a fundamental prerequisite is a thorough understanding of how test cases evolve in practice. Without such knowledge, we risk to develop techniques that may work well for only a small number of tests or, worse, that may not work at all in most realistic cases. Unfortunately, to date there are no studies in the literature that investigate how test suites evolve. To tackle this problem, in this paper we present a technique for studying test-suite evolution, a tool that implements the technique, and an extensive empirical study in which we used our technique to study many versions of six real-world programs and their unit test suites. This is the first study of this kind, and our results reveal several interesting aspects of test-suite evolution. In particular, our findings show that test repair is just one possible reason for test-suite evolution, whereas most changes involve refactorings, deletions, and additions of test cases. Our results also show that test modifications tend to involve complex, and hard-to-automate, changes to test cases, and that existing test-repair techniques that focus exclusively on assertions may have limited practical applicability. More generally, our findings provide initial insight on how test cases are added, removed, and modified in practice, and can guide future research efforts in the area of test-suite evolution.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Test-suite evolution, test-suite maintenance, unit testing

```
1 public void testDiscardSemicolons() throws Throwable {
2    Tokenizer t = new JavaTokenizer();
3    SourceCode sourceCode = new SourceCode("1");
4    String data = "public class Foo {private int x;}";
5    Tokens tokens = new Tokens();
6    t.tokenize(sourceCode, tokens, new StringReader(data));  // Broken statement
7    assertEquals(9, tokens.size());
8 }
```

(a)

```
1 public void testDiscardSemicolons() throws Throwable {
2    Tokenizer t = new JavaTokenizer();
3    SourceCode sourceCode = new SourceCode("1");
4    String data = "public class Foo {private int x;}";
5    Tokens tokens = new Tokens();
6    sourceCode.readSource(new StringReader(data));  // Added statement
7    t.tokenize(sourceCode, tokens);                 // Modified statement
8    assertEquals(9, tokens.size());
9 }
```

(b)

Figure 1: Two versions of a test case from PMD's unit test suite: (a) version 1.4, broken, and (b) version 1.6, repaired.

1. INTRODUCTION

Test cases form the first line of defense against the introduction of software faults (especially when retesting modified software). With the availability of convenient testing frameworks, such as JUnit, and the adoption of agile development methodologies, in particular, writing unit test cases is an increasingly common practice nowadays. Developers routinely create test suites that consist of unit tests and that they run periodically on their code.[1]

Test suites are not static entities: they constantly evolve along with the application they test. In particular, changes in the application can break test cases—in some cases, even a small change in the application code can affect a large number of tests. In general, a test failure on a new version of the software can either expose application faults or result from a problem with the test itself. On observing a failure, the first task for the developer is to determine which of these two is the cause of the failure and, once this distinction is made, either fix the problem in the code or fix the broken test.

A broken test, if it covers a valid functionality, should ideally be repaired. Alternatively, if the repair is unduly complex to perform or if the test was designed to cover a functionality that no longer exists in the application, the test should be removed from the test suite. To illustrate with an example, Figure 1 shows two versions of a unit test case from the test suite of `PMD`, one of the programs used in our empirical study. A change in `PMD`'s API broke the original version of the test case, which had to be fixed by adding a call to method `SourceCode.readSource` and removing one parameter from the call to method `Tokenizer.tokenize` (lines 6 and 7 in Figure 1(b), respectively).

[1]In this paper we use the terms "test", "test case", and "unit test case" interchangeably to indicate a pair (input, expected output).

Because test repair can be an expensive activity, automating it—even if only partially—could save a considerable amount of resources during maintenance. This is the motivation behind the development of automated test-repair techniques, such as the ones targeted at unit test cases [7, 8, 17] and those focused on GUI (or system) test cases [4, 11, 13, 14].

We believe that, to develop effective techniques for assisting manual test repair, we must first understand how test suites evolve in practice. That is, we must understand when and how tests are created, removed, and modified. This is a necessary, preliminary step because it can (1) provide evidence that test cases do get repaired, (2) support the hypothesis that test repairs can be (at least partially) automated, and (3) suitably direct research efforts. Without such understanding, we risk to develop techniques that may not be generally applicable and may not perform the kind of repairs that are actually needed in real-world software systems.

To the best of our knowledge, to date there are no studies in the literature that investigate how unit test suites evolve. To address this issue, we defined a technique that combines various static- and dynamic-analysis techniques to compute the differences between the test suites associated with two versions of a program and categorize such changes along two dimensions: the static differences between the tests in the two test suites and the behavioral differences between such tests.

We implemented our technique in a tool, TESTEVOL, which enables a systematic study of test-suite evolution. TESTEVOL targets Java programs and JUnit test suites, as Java is a widely used language and JUnit is the de-facto standard unit-testing framework for Java. Given two versions of a program and its test suite, TESTEVOL automatically computes differences in the behavior of the test suites on the two program versions, classifies the actual repairs performed between the versions, and computes the coverage attained by the tests on the two program versions.

We used TESTEVOL to conduct an extensive empirical study on how test suites evolved over a number of years for six real-world open-source software systems. In the study, we investigated several questions on test evolution, such as: What types of test-suite changes occur in practice and with what frequency? How often do test repairs require complex modifications of the tests? Why are tests deleted and added? Overall, we studied 88 program versions, 14,312 tests, and 17,427 test changes (5,111 test modifications, 2,541 test deletions, and 9,775 test additions). This is the first study of this kind and magnitude, and our results reveal several interesting aspects of test-suite evolution.

In the first part of our study, we focus on test repair. We provide evidence that, although test repairs are a relatively small fraction of the activities performed during test evolution, they are indeed relevant. We also show that repair techniques that just focus on oracles (*i.e.,* assertions) are likely to be inadequate and not applicable in many cases. Finally, our findings can be used to guide future research efforts in the development of new repair techniques that are grounded in the realities of how tests actually evolve.

Because our results show that test repair is only part of the picture, in the second part of our study we investigate the characteristics of deleted and added test cases. Our results provide evidence that many test cases are not really deleted and added, but rather moved or renamed. We also show that test cases are rarely removed because they are difficult to fix, but rather because they have become obsolete. Finally, we discover that test cases are not only added to check bug fixes and test new functionality, as expected, but also to validate changes in the code. This result supports the argument that the development of techniques for test-suite augmentation is useful and can have practical impact.

The main contributions of this paper are:

- The identification of the problem of understanding how test suites evolve as a prerequisite for performing research in the area of test repair (and test evolution in general).
- A technique for studying test-suite evolution and a publicly available tool, TESTEVOL,[2] that implements the technique for Java programs and JUnit test cases.
- An extensive analysis, performed using TESTEVOL, of the evolution of six real-world systems and their unit test suites over a number of years, versions, and test changes.
- Results and findings that allow for assessing existing techniques for supporting test evolution and can guide future research in the broader area of test maintenance.

The rest of the paper is organized as follows. The next section presents definitions and terminology. Section 2 describes our approach for studying test-suite evolution. Section 3 presents the TESTEVOL tool. Section 4 presents our empirical study and results. Section 5 discusses related work. Finally, Section 6 summarizes the paper and lists possible directions for future research.

2. OUR APPROACH FOR STUDYING TEST-SUITE EVOLUTION

2.1 Definitions and Terminology

A *system* $S = (P, T)$ consists of a program P and a test suite T. A *test suite* $T = \{t_1, t_2, \ldots, t_n\}$ consists of a set of unit test cases. $Test(P, t)$ is a function that executes test case t on program P and returns the outcome of the test execution. A *test outcome* can be of one of four types:

- $Pass$: The execution of P against t succeeds.
- $Fail_{CE}$: The execution of P against t fails because a class or method accessed in t does not exist in P.[3]
- $Fail_{RE}$: The execution of P against t fails due to an uncaught runtime exception (*e.g.,* a "null pointer" exception).
- $Fail_{AE}$: The execution of P against t fails due to an assertion violation.

We use the generic term *Fail* to refer to failures for which the distinction among the above three types of failures is unnecessary.

$Cov(P, t)$ is a function that instruments program P, executes test case t on P, and returns the set of all statements in P covered by t. $Cov(P, T)$ returns the cumulative coverage achieved on P by all the tests in test suite T.

Given a system $S = (P, T)$, a modified version of S, $S' = (P', T')$, and a test case t in $T \cup T'$, there are three possible scenarios to consider: (1) t exists in T and T', (2) t exists in T but not T' (*i.e.,* t was removed from the test suite), and (3) t exists in T' but not in T (*i.e.,* t was added to the test suite). These scenarios can be further classified based on the behavior of t in S and S', as summarized in Figure 2 and discussed in the rest of this section.

[2]TESTEVOL can be downloaded at `http://www.cc.gatech.edu/~orso/software/testevol.html`.

[3]These failures can obviously be detected at compile-time. For the sake of consistency in the discussion, however, we consider such cases to be detected at runtime via "class not found" or "no such method" exceptions. In fact, our TESTEVOL tool detects such failures at runtime by executing the tests compiled using the previous version of P on P.

(a) Test t exists in S and S' and is modified

$Test(P', t) = Fail \wedge$ $Test(P', t') = Pass$	t is repaired [TESTREP]
$Test(P', t) = Pass \wedge$ $Test(P', t') = Pass$	t is refactored, updated to test a different scenario, or is made more/less discriminating [TESTMODNOTREP]

(b) Test t is removed in S'

$Test(P', t) = Fail_{RE} \mid Fail_{AE}$	t is too difficult to fix [TESTDEL$_{(AE\|RE)}$]
$Test(P', t) = Fail_{CE}$	t is obsolete or is too difficult to fix [TESTDEL$_{(CE)}$]
$Test(P', t) = Pass$	t is redundant [TESTDEL$_{(P)}$]

(c) Test t' is added in S'

$Test(P, t') = Fail_{RE} \mid Fail_{AE}$	t' is added to validate a bug fix [TESTADD$_{(AE\|RE)}$]
$Test(P, t') = Fail_{CE}$	t' is added to test a new functionality or a code refactoring [TESTADD$_{(CE)}$]
$Test(P, t') = Pass$	t' is added to test an existing feature or for coverage-based augmentation [TESTADD$_{(P)}$]

Figure 2: Scenarios considered in our investigation. Given two system versions $S = (P, T)$ and $S' = (P', T')$, the three scenarios are: (a) t exists in T and T' and is modified, (b) t exists in T but not in T', (c) t' exists in T' but not in T.

2.2 Test Modifications

Figure 2(a) illustrates the scenario in which t is present in the test suites for both the old and the new versions of the system. To study different cases, we consider whether t is modified (to t') and, if so, whether the behaviors of t and t' differ. (We have not considered the cases in which t is not modified because they are irrelevant for studying test evolution.) For *behavioral differences*, there are two cases, shown in the two rows of the table: either t fails on P' and t' passes on P' or both t and t' pass on P'.

2.2.1 *Category* TESTREP*: Repaired Tests*

The TESTREP category corresponds to the case where t is repaired so that, after the modifications, it passes on P'. As discussed in the Introduction, Figure 1 shows an example of such a test repair. The code fragments in Listings 1 and 2 present another example of repair that involves a simpler code modification than the one in Figure 1. The example is taken from `Gson`, one of the programs used in our empirical study, and involves a test case from `Gson` version 2.0 that was fixed in the subsequent version 2.1. (Section 4.1 describes the programs used in the empirical study.) Test `testNullField` had to be fixed because constructor `FieldAttributes(Class<?> declClazz, Field f)` from version 2.0 was modified in version 2.1 to take only one parameter (of type `Field`).

Listing 1: Unit test for class FieldAttributes (Gson v2.0)

```
public void testNullField() throws Exception {
  try {
    new FieldAttributes(Foo.class, null);
    fail("Field parameter can not be null");
  } catch (NullPointerException expected) { }
}
```

Listing 2: Unit test for class FieldAttributes (Gson v2.1)

```
public void testNullField() throws Exception {
  try {
    new FieldAttributes(null);
    fail("Field parameter can not be null");
  } catch (NullPointerException expected) { }
}
```

For this category, we wish to study the types of modifications that are made to t. A test repair may involve changing the sequence of method calls, assertions, data values, or control flow. Based on our experience, for *method-call sequence changes*, we consider five types of modifications:

1. *Method call added*: a new method call is added.
2. *Method call deleted*: an existing method call is removed.
3. *Method parameter added*: a method call is modified such that new parameters are added.
4. *Method parameter deleted*: a method call is modified such that existing parameters are deleted.
5. *Method parameter modified*: a method call is modified via changes in the values of its actual parameters.

A test repair may involve multiple such changes. For example, the repair shown in Figure 1 involves the addition of a method call (in line 6) and the deletion of a method parameter (in line 7). The repair illustrated in Listings 1 and 2 also involves the deletion of a method parameter. For *assertion changes*, we consider cases in which an assertion is added, an assertion is deleted, the expected value of an assertion is modified, or the assertion is modified but the expected value is unchanged. Finally, we currently group together *data-value changes* and *control-flow changes*.

The rationale underlying our classification scheme is that different classes of changes may require different types of repair analyses. Although this is not necessarily the case, at least the search strategy for candidate repairs would differ for the different categories of changes. Consider the case of method-parameter deletion, for instance, for which one could attempt a repair by simply deleting some of the actual parameters. Whether this repair would work depends on the situation. For the code in Figure 1, for example, it would not work because the deletion of one of the parameters in the call to `tokenize()` is insufficient by itself to fix the test—a new method call (to `readSource()`) has to be added as well for the test to work correctly. Similarly, for the case of method-parameter addition, one could conceivably attempt straightforward fixes by constructing equivalence classes for the new parameters and selecting a value from each equivalence class (*e.g.,* positive, negative, and zero values for an integer parameter). In our study, we found cases where such an approach would in fact repair broken tests. In this case too, however, such a solution is in general not enough.

2.2.2 *Category* TESTMODNOTREP*: Refactored Tests*

The TESTMODNOTREP category captures scenarios in which a test t is modified in S' even though t passes on P'. Listings 3 and 4 show an example of one such change from `Commons Math`. Unit test `testDistance` was refactored to invoke method `sqrt()` in class `FastMath`, a newly added class in the new release, instead of the same method in `java.lang.Math`. There are different reasons why changes in this category might occur, as we discuss in Section 4.3.

Listing 3: Unit test from class Vector3DTest (Commons Math v2.1)

```
public void testDistance() {
  Vector3D v1 = new Vector3D(1, -2, 3);
  Vector3D v2 = new Vector3D(-4, 2, 0);
  assertEquals(0.0, Vector3D.distance(Vector3D.MINUS_I, Vector3D.MINUS_I), 0);
  assertEquals(Math.sqrt(50), Vector3D.distance(v1, v2), 1.0e-12);
  assertEquals(v1.subtract(v2).getNorm(), Vector3D.distance(v1, v2), 1.0e-12);
}
```

Listing 4: Unit test from class Vector3DTest (Commons Math v2.2)

```
public void testDistance() {
  Vector3D v1 = new Vector3D(1, -2, 3);
  Vector3D v2 = new Vector3D(-4, 2, 0);
  assertEquals(0.0,Vector3D.distance(Vector3D.MINUS_I, Vector3D.MINUS_I), 0);
  assertEquals(FastMath.sqrt(50), Vector3D.distance(v1, v2), 1.0e-12);
  assertEquals(v1.subtract(v2).getNorm(), Vector3D.distance(v1, v2), 1.0e-12);
}
```

2.3 Test Deletions

Figure 2(b) illustrates the scenario in which a test t is deleted. To study the reasons for this, we examine the behavior of t on the new program version P' and consider three types of behaviors.

2.3.1 Category $\text{TESTDEL}_{(AE|RE)}$: *Hard-To-Fix Tests*

This category includes tests that fail on P' with a runtime exception or an assertion violation. These may be instances where the tests should have been fixed, as the functionality that they test in P still exists in P', but the tests were discarded instead. One plausible hypothesis is that tests in this category involve repairs of undue complexity, for which the investigation of new repair techniques to aid the developer might be particularly useful. We performed a preliminary investigation of this hypothesis by manually examining ten randomly selected tests in this category and found that all the examined tests were in fact obsolete (we further discuss this point in Section 4.4).

2.3.2 Category $\text{TESTDEL}_{(CE)}$: *Obsolete Tests*

A test that fails with a compilation error on the new program version is obsolete because of API changes. Listing 5 illustrates a test in this category taken from `JodaTime`. This test was deleted because the tested method `Chronology.getBuddhist()` was removed in the subsequent version of `JodaTime`.

Listing 5: Unit test from class TestChronology (JodaTime v2.0)

```
public void testGetBuddhist() {
  assertEquals(BuddhistChronology.getInstance(), Chronology.getBuddhist());
}
```

Although for this category of deletion too, one could postulate that the tests were removed because they were too difficult to fix, we believe this not to be the case in most practical occurrences. Instead, the more likely explanation is that the tests were removed simply because the tested methods were no longer present. Also in this case, we investigated our hypothesis by manually examining ten randomly selected cases. Indeed, our manual investigation confirmed that, for the cases we analyzed, the tested functionality was either removed or provided through alternative methods, which requires the development of new tests rather than fixes to the existing ones.

2.3.3 Category $\text{TESTDEL}_{(P)}$: *Redundant Tests*

This category includes tests that are removed even though they pass on P'. Listing 6 illustrates an example of one such test, taken from `Commons Lang`. In the new version, this test was replaced by a more sophisticated one, shown in Listing 7.

Listing 6: Unit test from class MatrixIndexExceptionTest (Commons Lang v2.1)

```
public void testConstructorMessage(){
  String msg = "message";
  MatrixIndexException ex = new MatrixIndexException(msg);
  assertEquals(msg, ex.getMessage());
}
```

Listing 7: Unit test implemented to replace the removed one (Commons Lang v2.2)

```
public void testParameter(){
  MatrixIndexException ex =
        new MatrixIndexException(INDEX_OUT_OF_RANGE, 12, 0, 5);
  assertEquals(12, ex.getArguments()[0]);
  assertEquals(0,  ex.getArguments()[1]);
  assertEquals(5,  ex.getArguments()[2]);
}
```

2.4 Test Additions

Figure 2(c) illustrates the cases of test-suite augmentation, where a new test t' is added to the test suite. The behavior of t' on the old program can indicate the reason why it may have been added.

2.4.1 Category $\text{TESTADD}_{(AE|RE)}$: *Bug-Fix Tests*

This category includes added tests that fail on P with a runtime exception or an assertion violation. In this case, the functionality that the added test t' was designed to test exists in P but is not working as expected (most likely because of a fault). The program modifications between P and P' would ostensibly have been made to fix the fault, which causes t' to pass on P'. Thus, t' is added to the test suite to validate the bug fix. Listing 8 illustrates a test that fits this profile, as it was added to `Commons Math` version 2.1 to validate a bug fix (`https://issues.apache.org/jira/browse/MATH-391`). In the previous version of `ArrayRealVector`, the creation of a zero-length vector results in a runtime exception.

Listing 8: Unit test from class ArrayRealVectorTest (Commons Math v2.2)

```
public void testZeroVectors() {
  assertEquals(0, new ArrayRealVector(new double[0]).getDimension());
  assertEquals(0, new ArrayRealVector(new double[0], true).getDimension());
  assertEquals(0, new ArrayRealVector(new double[0], false).getDimension());
}
```

2.4.2 Category $\text{TESTADD}_{(CE)}$: *New-Features Tests*

The added tests in this category fail on P with a compilation error, which indicates that the API accessed by the tests does not exist in P. Thus, the added test t' is created to test new code in P', where the code could have been added as part of a refactoring or, more likely, to add new functionality. Listing 9 illustrates a test from `JFreeChart` that covers a new functionality: method `getMinY()` did not exist in previous versions of class `TimeSeries`.

Listing 9: Unit test from class TimeSeriesTests (JFreeChart v1.0.14)

```
public void testGetMinY() {
  TimeSeries s1 = new TimeSeries("S1");
  assertTrue(Double.isNaN(s1.getMinY()));
  s1.add(new Year(2008), 1.1);
  assertEquals(1.1, s1.getMinY(), EPSILON);
  s1.add(new Year(2009), 2.2);
  assertEquals(1.1, s1.getMinY(), EPSILON);
  s1.add(new Year(2002), -1.1);
  assertEquals(-1.1, s1.getMinY(), EPSILON);
}
```

2.4.3 Category $\text{TESTADD}_{(P)}$: *Coverage-Augmentation Tests*

This category considers cases where the added test t' passes on P. Clearly, t' would have been a valid test in the old system as well. One would expect that the addition of t' increases program coverage (*i.e.,* $Cov(P', T') \supset Cov(P', T' - \{t'\})$). Moreover, if t' covers different statements in P and P' (assuming that there is a way of matching statements between P and P'), the plausible explanation is that t' was added to test the changes made between P and P'. However, if t' covers the same statements in both program versions, it would have been added purely to increase code coverage (and not to test any added or modified code). Listing 10 illustrates an added test case that increases code coverage: the new test method `testFindRangeBounds` covers the case where the parameter of method `findRangeBounds()` is null.

Listing 10: Unit test from class XYErrorRendererTests (JFreeChart v1.0.14)

```
public void testFindRangeBounds() {
  XYErrorRenderer r = new XYErrorRenderer();
  assertNull(r.findRangeBounds(null));
}
```

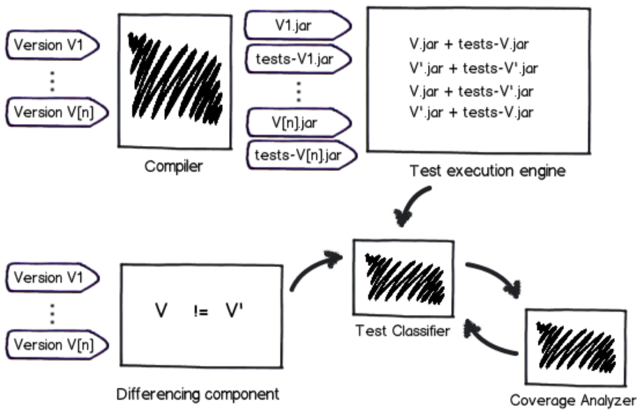

Figure 3: High-level architecture of TESTEVOL.

3. IMPLEMENTATION

We implemented a tool, called TESTEVOL, which facilitates a systematic study of test-suite evolution for Java programs and JUnit test suites. TESTEVOL analyzes a sequence of versions of a software system, where each version consists of application code and test code. The versions can be actual releases or internal builds. TESTEVOL consists of five components, as illustrated in the architecture diagram shown in Figure 3.

The *compiler* component builds each system version and creates two jar files, one containing the application classes and the other containing the test classes. The *test-execution engine* analyzes each pair of system versions (S, S'), where $S = (P, T)$ and $S' = (P', T')$. First, it executes T on program P and T' on program P' (*i.e.,* it runs the tests on the respective program versions). Then, it executes T' on P and T on P'. For each execution, it records the test outcome: $Pass, Fail_{CE}, Fail_{AE}, or Fail_{RE}$. The *differencing component* compares T and T' to identify modified, deleted, and added tests. This component is implemented using the WALA analysis infrastructure for Java (`wala.sourceforge.net`).

The test outcomes, collected by the test-execution engine, and the test-suite changes, computed by the differencing component, are then passed to the *test classifier*, which analyzes the information about test outcomes and test updates to classify each update into one of the eight test-evolution categories presented in Section 2. For each pair of broken test case and its repaired version, the test classifier also compares the test cases to identify the types of repair changes—different types of method-sequence changes and assertion changes—as discussed in Section 2.2.1; this analysis is also implemented using WALA.

TESTEVOL performs a further step for the test cases in categories TESTDEL$_{(P)}$and TESTADD$_{(P)}$. For these tests, the test classifier leverages the *coverage analyzer* to compute the branch coverage achieved by each test; this facilitates the investigation of whether the deleted or added tests cause any variations in the coverage attained by the new test suite.

4. EMPIRICAL STUDY

Using TESTEVOL, we conducted an empirical study on several Java open-source software systems to investigate different aspects of test-suite evolution. In this section, we discuss the results of our analysis of this information and the insight on test-suite evolution that we gained. Specifically, we discuss our experimental setup, analyze the overall occurrence—in the programs and versions considered—of the categories of test changes presented in Section 2, and investigate the different categories (*i.e.,* tests modified, deleted, and added) in detail.

4.1 Programs, Tests, and Versions Investigated

Table 1 lists the programs that we used in our study, which are all real-world open-source programs that we selected from popular websites, such as SourceForge (`www.sourceforge.net`). The general criteria that we used in selecting the programs were that they had to be popular, be actively maintained, and have a JUnit test suite. For each of the programs selected, we downloaded all of its official releases (with the exception of `PMD`, as explained below). Columns 3 and 4 of the table show the number of versions of each program and the IDs of these versions, respectively. Columns 5–8 show various metrics for the first version of the program: number of classes, number of methods, number of JUnit tests, and release date. Columns 9–12 show the same data for the last version considered. Finally, the last column in the table shows the cumulative number of tests for each program over all of its versions.

Note that one of the programs, `PMD`, appears twice in the table (the second time indicated as `PMD-2004`). For this program, we studied two sets of versions: the official releases from November 2002 to November 2011 and 17 versions from January to December 2004. (The 17 versions are between one and six weeks apart and were selected, using Kenyon [2], so as to contain at least one test change. Because of the way we selected them, the sets of versions for `PMD` and `PMD-2004` may be completely disjoint.) We used this second set of versions to assess whether our findings depended on the granularity of the changes considered, as further discussed in Section 4.6.

We ran TESTEVOL on each pair of subsequent versions of each program. TESTEVOL analyzed each pair and produced the information described in Section 2. This information, together with the actual programs and test versions, were the basis for our study.

4.2 Occurrence of Test-Change Categories

In the empirical evaluation, we first studied how often tests change, during program evolution, and how they change. To do this, we used the information computed by TESTEVOL to answer the following research question:

- **RQ1.** How often do the different categories of test-suite changes that we consider occur?

To answer RQ1, we used the data about the distribution of test-change categories in the programs, which are reported in Tables 2 and 3. Table 2 shows the distribution in absolute numbers, which gives a sense of the relevance of each category throughout the lifetime of a program. For example, the table shows that, for `Commons Math`, our study analyzed more than 4,300 test changes overall.

Although looking at absolute numbers is useful to get an idea of the magnitude of the number of test changes, it makes it difficult to compare the results across programs, as different programs could have different versions and lifetime spans. We thus normalized the distribution of test changes with respect to the number of studied versions by considering the test differences between each pair of versions of the programs. We report these numbers in Table 3.

The columns in both tables contain analogous data. Column 2 reports the number of test updates (modifications, deletions, or additions), whereas columns 3–10 report the distribution of the eight categories of test changes.

Let us first consider the total number of test-suite updates. Over all programs, we observed 17,427 test changes, of which 5,111 (29%) were test modifications, 2,541 (15%) were test deletions, and 9,775 (56%) were test additions. The average number of test updates per version pair ranged from 112, for `PMD`, to 732 for `Commons Math`. Consider now the data for test modifications (columns 3–4 of Tables 2 and 3). Among the 5,111 modifications, only 1,121 (22%

Table 1: Programs used in our empirical study.

Program	Description	Number of Versions	Versions	First Version				Last Version				$\bigcup_{v_1}^{v_n}$ Tests
				Classes	Methods	Tests	Release	Classes	Methods	Tests	Release	
Commons Lang	Extra utilities for the java.lang API	11	1.0, 2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 3.0, 3.0.1, 3.1	31	373	318	Dec/2002	137	1174	2027	Nov/2011	2531
Commons Math	Java mathematics and statistics library	7	1.0, 1.1, 1.2, 2.0, 2.1, 2.2, 3.0	83	758	501	Dec/2004	534	4329	2415	Mar/2012	3164
Gson	Library for converting Java objects into their JSON representation and viceversa	16	1.0, 1.1, 1.1.1, 1.2, 1.2.1, 1.2.2, 1.2.3, 1.3, 1.4, 1.5, 1.6, 1.7, 1.7.1, 1.7.2, 2.0, 2.1	73	414	131	May/2008	57	424	800	Dec/2011	1200
JFreeChart	Chart creation library	12	1.0.03, 1.0.04, 1.0.05, 1.0.06, 1.0.07, 1.0.08a, 1.0.09, 1.0.10, 1.0.11, 1.0.12, 1.0.13, 1.0.14	423	5790	1297	Nov/2006	510	7736	2186	Nov/2011	2211
JodaTime	Replacement for the Java date and time API	15	0.9.8, 0.9.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.5.1, 1.5.2, 1.6, 1.6.1, 1.6.2, 2.0, 2.1	177	2619	1950	Nov/2004	201	3436	3887	Feb/2012	3963
PMD	Java source-code static analyzer	27	1.0, 1.2, 1.4, 1.6, 1.8, 2.0, 2.1, 2.2, 2.3, 3.0, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 4.0, 4.1, 4.1.1, 4.2.2, 4.2.3, 4.2.4, 4.2.5, 4.2.6, 4.3	316	1846	340	Nov/2002	602	4266	596	Nov/2011	1243
PMD-2004		17	r2515, r2559, r2567, r2628, r2679, r2723, r2750, r2769, r2781, r2832, r2846, r2911, r2936, r2962, r2975, r3016, r3065	288	1908	425	Jan/2004	343	2241	504	Dec/2004	573
Total	–	88	–	1103	11800	4537	–	2041	21365	11911	–	14312

Table 2: Distribution of the test-change categories in the programs we analyzed—absolute numbers.

Program	Number of Test Updates	Test Modification		Test Deletion			Test Addition		
		TestRep	TestModNotRep	$\text{TestDel}_{(AE\|RE)}$	$\text{TestDel}_{(CE)}$	$\text{TestDel}_{(P)}$	$\text{TestAdd}_{(AE\|RE)}$	$\text{TestAdd}_{(CE)}$	$\text{TestAdd}_{(P)}$
Commons Lang	4284	257 (6.0%)	1274 (29.7%)	25 (0.6%)	393 (9.2%)	122 (2.8%)	200 (4.7%)	1679 (39.2%)	334 (7.8%)
Commons Math	4387	204 (4.7%)	701 (16.0%)	18 (0.4%)	673 (15.3%)	128 (2.9%)	203 (4.6%)	2313 (52.7%)	147 (3.4%)
Gson	1761	195 (11.1%)	97 (5.5%)	20 (1.1%)	237 (13.5%)	143 (19.8%)	284 (16.1%)	422 (24.0%)	363 (20.6%)
JFreeChart	1318	59 (4.5%)	320 (24.3%)	6 (0.5%)	0 (0%)	19 (1.4%)	292 (22.2%)	345 (26.2%)	277 (21.0%)
JodaTime	2786	145 (5.2%)	552 (19.8%)	0 (0%)	18 (0.6%)	58 (2.1%)	166 (6.0%)	1576 (56.6%)	271 (9.7%)
PMD	2891	261 (9.0%)	1046 (36.2%)	43 (1.5%)	161 (5.6%)	477 (16.5%)	213 (7.4%)	418 (14.5%)	272 (9.3%)
Total	17427	1121 (6.4%)	3990 (22.9%)	112 (0.6%)	1482 (8.5%)	947 (5.4%)	1358 (7.8%)	6753 (38.8%)	1664 (9.5%)

of the modifications and 6% of all test changes) were performed to fix broken tests (*i.e.,* they were actual test repairs). The remaning 3,990 modifications (78% of the modifications and 23% of all test changes) were changes made to passing tests, that is, changes made for reasons other than fixing broken tests. This data supports the observation that test repairs are only one type of change in test-suite evolution, and definitely not the most frequent. It may therefore be worth investigating techniques that support other kinds of test modifications and test changes in general.

Test repairs are only part of the story: test repair, and test modifications in general, are a minority of all test changes.

Looking at the data in more detail, however, we can also observe that test repairs, although not prevalent, are not irrelevant either. If we ignore the case of added and deleted tests, which we will discuss in detail in Section 4.4, test repairs represent a significant fraction of test modifications; the percentage of actual test repair instances over all test modifications varies, in fact, from 16%, for `JFreeChart`, to 67%, for `Gson`. This data seems to provide a fair amount of motivation for the development of automated unit test-repair techniques [7, 8, 17].

Test repairs occur often enough in practice to justify the development of automated repair techniques.

Based on the above observations, we further analyzed the data to get a better understanding of the kinds of both repair and non-repair modifications that occur in practice. We discuss our findings in the next section.

4.3 Test Modifications

The goal of our in-depth analysis of test modification was twofold. Our main goal was to study the types of repairs that are performed in practice, so as to gauge the applicability of existing repair techniques. Another, secondary goal was to understand what kinds of non-repair modifications occur in real programs, so as to assess the feasibility of developing automated techniques that can support these types of modifications too.

The starting point for the first part of our analysis were existing test-repair techniques. To the best of our knowledge, most existing techniques in this arena at the time of this study (*e.g.,* [7, 8]) focus on repairing the assertions associated with the failing test cases. (The main exception is the technique recently presented by Mirzaaghaei, Pastore, and Pezzé [17], which aims to handle changes in method signatures and which we discuss in Section 5.) Daniel and colleagues [7], in particular, present seven repair strategies for automatically fixing broken JUnit test cases. Six of these focus on modifying the failing `assert` statement in different ways, such as by replacing the expected value with the actual value, inverting a relational operator, or expanding object comparisons. (The final strategy applies in cases where a test fails with a runtime exception and works by surrounding the failing method call with a `try-catch` block that catches a particular exception type.) To study the applicability of assertion-focused repair, we investigated the following research question:

- **RQ2.** How often do test repairs involve modifications to existing assertions only? How often they require more complex modifications of the tests instead?

Table 3: Distribution of the test-change categories in the programs we analyzed—averaged over version pairs.

Program	Number of Test Updates	Test Modification		Test Deletion			Test Addition		
		TestRep	TestModNotRep	TestDel$_{(AE\|RE)}$	TestDel$_{(CE)}$	TestDel$_{(P)}$	TestAdd$_{(AE\|RE)}$	TestAdd$_{(CE)}$	TestAdd$_{(P)}$
Commons Lang	429	26 (6.0%)	127 (29.7%)	3 (0.6%)	39 (9.2%)	12 (2.8%)	20 (4.7%)	168 (39.2%)	33 (7.8%)
Commons Math	732	34 (4.7%)	117 (16.0%)	3 (0.4%)	112 (15.3%)	21 (2.9%)	34 (4.6%)	386 (52.7%)	25 (3.4%)
Gson	118	13 (11.1%)	6 (5.5%)	1 (1.1%)	16 (13.5%)	10 (8.1%)	19 (16.1%)	28 (24.0%)	24 (20.6%)
JFreeChart	121	5 (4.5%)	29 (24.3%)	1 (0.5%)	0 (0%)	2 (1.4%)	27 (22.2%)	31 (26.2%)	25 (21.0%)
JodaTime	200	10 (5.2%)	39 (19.8%)	0 (0%)	1 (0.6%)	4 (2.1%)	12 (6.0%)	113 (56.6%)	19 (9.7%)
PMD	112	10 (9.0%)	40 (36.2%)	2 (1.5%)	6 (5.6%)	18 (16.5%)	8 (7.4%)	16 (14.5%)	10 (9.4%)
PMD-2004	*28*	*6 (20.3%)*	*5 (17.6%)*	*0 (0%)*	*1 (3.6%)*	*6 (21.7%)*	*1 (3.6%)*	*2 (7.7%)*	7 (25.4%)
Average	285	16 (6.7%)	60 (21.9%)	2 (0.7%)	29 (7.4%)	11 (5.7%)	20 (10.2%)	124 (35.5%)	23 (12.0%)

Table 4: Types of modifications made to repair broken tests.

Program	Number of Test Repairs	Assertion Changes Only	Method-Call Changes					Assertion Changes			
			Call Added	Call Deleted	Parameter Added	Parameter Deleted	Parameter Modified	Assertion Added	Assertion Deleted	Assertion Modified	Exp Value Modified
Commons Lang	257	6 (2.3%)	138	109	0	161	28	59	35	17	8
Commons Math	204	48 (23.5%)	79	73	0	70	11	94	89	17	10
Gson	195	18 (9.2%)	85	76	13	100	28	23	25	2	14
JFreeChart	59	4 (6.8%)	21	4	1	53	19	3	3	0	3
JodaTime	145	11 (7.6%)	36	125	2	53	1	4	55	0	10
PMD	261	24 (9.2%)	171	143	25	103	31	50	64	1	16
PMD-2004	*84*	*2 (0.8%)*	*41*	*29*	*20*	*12*	*55*	*5*	*2*	*0*	*2*
Total	1121	111 (9.9%)	530 (47.3%)	530 (47.3%)	40 (3.6%)	540 (48.2%)	118 (10.5%)	233 (20.8%)	271 (24.2%)	37 (3.3%)	61 (5.4%)

Table 4 presents the analysis data relevant for the investigation of RQ2. Column 2 of the table shows the overall number of test repairs for each program (this is the same information that is shown in column 3 of Table 2). Column 3 reports the number of test repairs that involved changes to assertions only. Columns 4–8 list the number of occurrences of the five types of changes in method calls that we discussed in Section 2.2. Similarly, columns 9–12 list the number of occurrences of the four types of changes in `assert` statements.

The data in the first column of the table clearly shows that, for the programs considered, test repairs that involve only changes to assertions are rare—less than 10% overall for all programs except `Commons Math`. Looking at the remaining columns, we can also clearly observe that test repairs tend to involve changes to method calls considerably more often than changes to assertions. Over all programs considered, nearly 50% of the repairs involved the addition of a method call, the deletion of a method call, or the deletion of a method parameter. Assertion additions and assertion deletions (which occurred in 21% and 24% of the repairs, respectively) are also fairly common, but much less frequent than method-call changes. Also in this case, the only exception to this trend is `Commons Math`, for which assertion changes (210 in total) occurred nearly as often as method-call changes (233 in total).

Overall, this data provides strong indications that existing test-repair techniques, although useful in some cases, must be extended to be widely applicable (*e.g.,* by including the capability of generating and adapting method calls). Based on the data, we can therefore make the following observation:

Test-repair techniques that focus exclusively on assertions can target only a small subset of all broken (repairable) test cases and must be extended to achieve a wider applicability of automated test repair.

As stated earlier, in this part of the study we were also interested in gaining a better understanding of non-repair modifications. To this end, we analyzed the occurrences of such changes in our programs and discovered that there are different reasons why changes in this category are performed. In some of the cases that we observed, test cases were simply refactored. This was the case, for instance, for the example shown in Section 2.2.2 (Listings 3 and 4). In other cases, a test was updated to cover a different program functionality, which indicates that the old functionality either was removed or did not need to be tested by this particular test case. To distinguish between these two cases, we used coverage information because, intuitively, we would expect a test refactoring not to alter the coverage of the program achieved by the original and the modified tests. Therefore, if $Cov(P', t) = Cov(P', t')$, we consider the change as a test refactoring; otherwise, we conclude that the test has been changed to cover a different functionality. Although these two cases were prominent, we also observed other cases in which tests were made more discriminating, by the addition of assertions, or less discriminating, by the deletion of assertions.

For these three types of test changes, it seems difficult to define techniques that could automatically perform the changes. All three types of changes we observed, in fact, seem to involve a considerable amount of human judgement. Moreover, these changes are typically not strictly necessary, at least when compared to fixing a broken test. Providing support for them seems, therefore, to be less of a priority. Finally, standard refactoring and advanced editing tools, such as the ones provided in modern IDEs, may already provide enough support for performing these changes.

Investigating techniques for automated support of non-repair changes does not appear to be a promising research avenue. These techniques would require considerable manual guidance and may end up being similar to traditional refactoring tools.

4.4 Test Deletions

Although studying test modifications and, in particular, test repairs was the main focus of our study, we also investigated the characteristics of deleted and added test cases. In this section, we describe our findings with respect to test deletions, for which we investigated the following research question:

- **RQ3.** Why are tests deleted? Are tests in different categories deleted for different reasons?

To answer RQ3, we use the data provided in columns 5–7 of Tables 2 and 3. As the data shows, test deletions occur often. Moreover, failing tests are more often deleted (1,594 instances of deletions) than repaired (1,121 instances of repairs). Among the deleted failing tests, tests fail predominantly (over 92%) with compilation errors, whereas the remaining ones fail with assertion or runtime errors.

This phenomenon indicates that, in most cases, the deleted tests cover obsolete functionality, as the application API exercised by the tests no longer exists. The interesting question, however, is whether this is true also for tests in the $\text{TESTDEL}_{(AE|RE)}$ category. Given that these tests could be run, but failed, on the modified system, an interesting additional question arises. Were these tests truly obsolete and served no purpose in the new version of the system? Or could they have been fixed instead, but fixing was too complex? To address this question, we randomly examined 30 instances from category $\text{TESTDEL}_{(AE|RE)}$ and found that, in all the observed cases, the tests were indeed obsolete, and it did not make sense to repair them.

To illustrate, Listing 11 shows a test from `Commons Lang` that belongs to this category. After examining the history of deleted test `testBigDecimal`, we discovered that it was added in version 2.4 to test a new functionality in class `EqualsBuilder`, and that the functionality was later removed due to a side effect (see `https://issues.apache.org/jira/browse/LANG-393`). In this case, and in the other cases that we examined, the test was indeed obsolete and was correctly removed.

Listing 11: Unit test from class EqualsBuilderTest (Commons Lang v2.4)

```
public void testBigDecimal() {
  BigDecimal o1 = new BigDecimal("2.0");
  BigDecimal o2 = new BigDecimal("2.00");
  assertTrue(new EqualsBuilder().append(o1, o1).isEquals());
  assertTrue(new EqualsBuilder().append(o1, o2).isEquals());
}
```

As another example, Listing 12 shows a code fragment from `JFreeChart` version 1.0.10. In this version, any year before 1900 is considered invalid, and method `Year.previous` returns null for such a year. This is no longer true for the following version of `JFreeChart`; therefore, in this case too, the test was correctly removed.

Listing 12: Unit test from class YearTests (JFreeChart v1.0.10)

```
public void test1900Previous() {
  Year current = new Year(1900);
  Year previous = (Year) current.previous();
  assertNull(previous);
}
```

Tests that fail in the new version of a program—because of a compilation error, a runtime exception, or a failed assertion—tend to be deleted not because they are difficult to repair, but because they are obsolete.

Another interesting fact highlighted by the data is that many tests that pass (and are valid tests) in both the old and new systems are deleted. In fact, category $\text{TESTDEL}_{(P)}$, with 947 members, accounts for over 5% of all test updates and over 37% of all test deletions. In general, if t passes on P', it is not obvious why it would be removed. One possibility is that many of these deletions are actually cases where tests were renamed or moved to a different class. To investigate this hypothesis, we studied the effects of the deletions of passing tests on code coverage. Specifically, we investigated whether the removal of each deleted test t resulted in any loss in coverage. ($Cov(P', T') - Cov(P', T' \cup \{t\})$ indicates the loss in coverage, if any, that results from the deletion of t.) The effects on coverage can, in fact, strongly indicate whether a test is truly deleted (if the removal of the test from the test suite *reduces* the coverage of the suite) or simply renamed or moved (if the removal of the test causes no change in coverage).

Table 5: Effects of deleted passing tests on branch coverage.

Program	$\text{TESTDEL}_{(P)}$	Same Branch Coverage	Reduced Branch Coverage
Commons Lang	122	120 (98.4%)	2 (1.6%)
Commons Math	128	50 (39.1%)	78 (60.9%)
Gson	143	140 (97.9%)	3 (2.1%)
JFreeChart	19	15 (78.9%)	4 (21.1%)
JodaTime	58	46 (79.3%)	12 (20.7%)
PMD	477	341 (71.5%)	136 (28.5%)
Total	947	712 (75.2%)	235 (24.8%)

Table 5 presents data on the effects of the deletion of passing tests on branch coverage. As the table shows, in most cases (over 75%) the removal of a test does not reduce branch coverage, a consistent result across all programs except `Commons Math`. This data suggests that many of the deletions that occur in category $\text{TEST-DEL}_{(P)}$ may not be actual deletions.

It is worth noting that the tests might have been deleted and might have simply been redundant coverage-wise. The manual examination of ten randomly selected samples, however, seemed to eliminate this possibility. As an example, Listing 13 illustrates a test that was moved and renamed in `Gson`. Initially, in version 1.1.1, the test was part of class `JsonSerializerTest`. In version 1.2, however, it was moved to class `ArrayTest` and renamed to `testArrayOfStringsSerialization`.

Listing 13: Unit test from class JsonSerializerTest (Gson v1.1.1)

```
public void testArrayOfStrings() {
  String[] target = {"Hello", "World"};
  assertEquals("[\"Hello\",\"World\"]", gson.toJson(target));
}
```

The data in Table 5 also shows that test deletions that cause reduction in branch coverage are a minority but do occur (24% of the cases). Although we could not find any definite explanation of the phenomenon, we conjecture that this may simply be due to differences in the structure of the code that cause the moved or renamed test to cover different instructions in some parts of the program. To assess the validity of this conjecture, we plan to investigate this category further in the future.

Tests that would pass in the new version of a program but appear to have been deleted have, in most cases, simply been moved or renamed.

4.5 Test Additions

In this final part of our investigation, we study the characteristics of added tests. Similar to what we did for deleted tests, in this case we investigate the following research question:

- **RQ4.** Why are tests added? Are tests in different categories added for different reasons?

Among the three types of test updates we studied, test additions are the ones that occur most frequently. This is somehow expected, if we consider that all the programs we studied grew significantly over the time period of the study, as demonstrated by the number of classes and methods in the first and last versions of the systems (see columns 5, 6, 9, and 10 of Table 1). Of the 9,775 instances

Table 6: Effects of the added passing tests on branch coverage.

Program	TESTADD$_{(P)}$	Same Branch Coverage	Increased Branch Coverage
Commons Lang	334	139 (41.6%)	195 (58.4%)
Commons Math	147	99 (67.3%)	48 (32.7%)
Gson	363	155 (42.7%)	208 (57.3%)
JFreeChart	277	162 (58.5%)	115 (41.5%)
JodaTime	271	198 (73.1%)	73 (26.9%)
PMD	272	177 (65.1%)	95 (34.9%)
Total	1664	930 (55.9%)	734 (44.1%)

of test additions, 1,358 (14%) failed in the previous version of the program with a runtime exception or an assertion failure. As we discussed and illustrated with an example in Section 2.4.1, these are tests that were added to validate a bug fix. However, the majority of the added tests (6,753—69%) simply did not compile for the old program. This fact indicates that they were most likely added to validate newly added code.

New test cases that would fail with a runtime error for the old version of the program are added to validate bug fixes; new tests that would not compile against the old version of the program are likely added to validate new functionality.

The remaining 1,664 added tests (17%) represent an interesting case, as they would both compile and pass on the previous version of the program, as shown by the column for TESTADD$_{(P)}$ in Table 2. Similar to what we did for the deleted tests, we used coverage information to investigate these added tests further.

Table 6 presents the branch coverage data that we collected and used to perform this investigation. As the table shows, and somehow unsurprisingly, a large portion of the added tests did not increase branch coverage. This is particularly significant for `Commons Math`, `JodaTime`, and `PMD`. Considering our findings for the case of deleted tests (see Section 4.4), some of these added tests must correspond to old tests that were moved or simply renamed.

As the table also shows, the added tests did increase coverage in many cases. The increase in percentage ranges from about 27%, for `JodaTime`, to over 58%, for `Commons Lang`, with the average being 44%. To better understand this phenomenon, we collected additional coverage data and studied whether these tests traversed modified parts of the code. Interestingly, in most cases (over 90%), the tests exercised one or more changes in the code. Albeit this is just preliminary evidence and would require a more in-depth analysis, it seems to indicate that developers do perform test augmentation not only to cover newly added code, but also to exercise the changes they made to existing code. If confirmed by further studies, this result would justify existing research in test-suite augmentation (*e.g.,* [19]) and could provide input for further efforts in that area.

A significant number of new tests are added not necessarily to cover new code, but rather to exercise the changed parts of the code after the program is modified.

4.6 Summary and Threats to Validity

Our empirical analysis of 6 real-world programs, 88 versions, 14,312 tests and 17,427 test changes allowed us to make a number of interesting observations. In this section, we summarize the most relevant of such observations. As far as test repair is concerned, our results show that (1) test repairs are needed, although they are not the majority of the test changes that we observed, and (2) techniques that just focus on assertions may not be applicable in a majority of cases.

As for test additions and deletions, we found initial evidence that (1) in many cases tests are not really deleted and added, but rather moved or simply renamed, (2) failing tests are deleted not because they are difficult to fix, but because they are obsolete, and (3) tests are added to check bug fixes, test new functionality, and validate changes made to the code.

Like any empirical study, there are threats to the validity of our results. Threats to internal validity may be caused by errors in our implementation of TESTEVOL that might have produced incorrect results. To mitigate this threat, we have thoroughly tested TESTEVOL with a number of small examples that contained the test changes targeted by our analysis. Moreover, in our analysis of the results, we have manually inspected many real cases of test changes computed by TESTEVOL and found them to be correct, which further increases our confidence in the tool.

There are also threats to the external validity of our results related to the fact that our findings may not generalize to other programs or test suites. One of the potential external threats, in particular, is the granularity at which we selected the program versions—that of external, official releases. Our results could change if evolution were to be studied at a finer grain by analyzing internal builds, with greater frequency. To perform a preliminary investigation of this issue, we obtained 17 internal builds of `PMD` (as mentioned in Section 4.1) and analyzed them using TESTEVOL. The data for these builds, shown in Tables 3 and 4 under the label `PMD-2004`, illustrates that a finer-grained study of test evolution can indeed produce some variations in the results. For example, we can observe a larger percentage of test repairs, which is probably due to the fact that tests can be repaired multiple times, in different internal builds, between two releases. In such cases, a coarser release-level analysis cannot identify the multiple repairs, whereas a finer build-level analysis can do so. These preliminary results show that there may be value in performing further studies at different levels of version granularity.

Overall, we considered applications from different domains, many versions for each application, large test suites, and numerous real-world test changes. Although more studies are needed to confirm our results, we believe that our initial findings provide a solid starting point on which we and other researchers can build.

5. RELATED WORK

Daniel and colleagues [7] presented the first automated technique and tool, called `ReAssert`, for repairing broken JUnit tests. `ReAssert` employs different repair strategies, whose common goal is to fix the failing `assert` statements to make the tests pass. For example, one repair strategy replaces the expected value of an assertion with the actual value observed, and another strategy inverts the relational operator in the `assert` condition. In subsequent work [8], Daniel and colleagues presented a symbolic-analysis-based repair technique that overcomes some of the limitations of `ReAssert` (*e.g.,* its ineffectiveness in the presence of conditional control flow). However, this technique also focuses exclusively on repairing assertions. As our results indicate, assertion-focused repair may have limited applicability in practice—test repairs predominantly involve changes, such as synthesis of new sequences of method calls, that leave the assertions unmodified.

More recently, Mirzaaghaei, Pastore, and Pezzé [17] presented a repair technique for fixing JUnit test cases that are broken because of changes in method signatures, that is, addition, deletion, or modification of parameters. Their technique identifies the broken method call and attempts to create a modified call in which new parameters are initialized with suitable values. Using data-flow analysis, program differencing, and runtime monitoring, the technique searches for initialization values from existing data values generated during the execution of the test case against the original application. Although its general effectiveness is unclear, the technique may work well for specific types of changes (*e.g.,* where a formal parameter is replaced with a type that wraps the parameter). This technique, by trying to handle changes in method signatures, is a right step in the direction of developing more widely applicable automated repairs. Based on our evidence, however, the effectiveness of the technique is still limited, and the development of more sophisticated approaches is required. In particular, the technique attempts to fix a broken method call by adding, deleting, and modifying parameters, but it does not synthesize new method calls. As the data from our study shows, the synthesis of new method calls is often needed when repairing test cases in practice (see column 4 of Table 4). The test repair illustrated in Figure 1, for instance, cannot be handled by this technique.

Test-repair techniques have also been developed for GUI tests (*e.g.,* [1, 4, 11, 13, 15]); such tests are sequences of events on an application user interface. Memon and Soffa [15] present a repair technique that constructs models of the original and modified GUI components, where the models represent the possible flow of events among the components. Their technique compares the models to identify deleted event flows and attempts to repair the tests that traverse such (invalid) flows. The repair strategy deletes one or more events from a test sequence, or splices one or more events with a different event, so that the resulting sequence can be executed on the modified GUI. Grechanik, Xie, and Fu [11] present a similar approach for repairing broken GUI test scripts: their approach constructs models of the original and modified GUIs, compares the models to identify modified GUI elements, and identifies the test-script actions that access such modified GUI elements. Choudhary and colleagues [4] present a repair technique for web-application test scripts. Their technique collects runtime data for test execution on the original and modified web application. By analyzing the broken script commands and the runtime data, it attempts to repair the commands to make the broken test scripts pass on the modified application. An empirical study similar to ours, but that focuses on the evolution of GUI test scripts, would be useful in assessing the practical applicability of these techniques and in identifying interesting research problems in GUI test repair.

Recently, several techniques for automated program repair have been developed (*e.g.,* [3, 10, 21, 22]). Some of these techniques rely on formal specifications of correct program behavior [10, 21], whereas others use a suite of passing tests as the specification of intended program behavior [3, 22]. It would be worth investigating whether these techniques, and especially the ones based on symbolic analysis (*e.g.,* [3]), could be adapted to be used in the context of automated support for test repair.

Our study revealed that, in many cases, changes in the application cause tests to fail with compilation errors. Such failures are similar in nature to the failures that result in client code when library APIs evolve in ways that are incompatible with the API clients. Therefore, ideas from existing research on automated API migration and adaptation (*e.g.,* [5, 6, 9, 12, 18]) could also be fruitfully leveraged for developing repair techniques for tests. Dagenais and Robillard [6], for instance, presented an approach for assisting with migrating client code to updated versions of framework APIs. Their key idea is to guide the client changes based on the framework code's own adaption to its API changes. Nguyen and colleagues [18] described a related approach where the adaptation of a client is guided by how other clients have adapted to the evolved API. Similar techniques could be developed for fixing unit tests based on the application's adaptation to API-level changes.

The notion of generating abstract "edit scripts" (*e.g.,* [16]) that apply a given program transformation in similar, but not identical, contexts may also be useful for developing test-repair techniques.

Finally, techniques for constraint-based synthesis of method sequences (*e.g.,* [20, 23]) could be leveraged for developing repair techniques that generate method sequences to satisfy the failing assertions in broken test cases.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented an extensive empirical study of how test suites evolve. Our study, which focuses on JUnit test suites, is the *first of its kind* in terms of its goals and magnitude. Although software evolution has been extensively studied, test suites—which can be large software systems in their own right and can also require continuous development and maintenance—have not been the objective of large empirical studies to date. Such studies, however, are essential if we are to develop automated techniques and tools for supporting test-suite evolution that are grounded in real-world scenarios. Our goal for this work was therefore to study test-suite evolution in a systematic and comprehensive manner to understand how and why tests evolve (*i.e.,* how and why tests are modified, added, and deleted). Toward that goal, we studied the evolution history, over a substantial period of time, of several open-source software systems that have large JUnit test suites.

Our study has provided several relevant insights. *First*, we found that test repair does occur in practice: on average, we observed 16 instances of test repairs per program version, and a total of 1,121 test repairs. Therefore, automating test repair can clearly be useful. *Second*, test repair is not the only, or even the predominant, reason why tests are modified: for the programs studied, non-repair test modifications occurred nearly four times as frequently as test repairs. *Third*, test repairs that focus mainly on fixing assertions (*i.e.,* oracles), as most existing techniques do, may have limited applicability: less than 10% of the test repairs in our study involved fixes to assertions only. *Fourth*, test repairs frequently involve changes to method calls and method-call sequences. Thus, the investigation of automated repair techniques targeted toward synthesizing new method sequences for making existing assertions pass is likely to be a worthwhile research direction to pursue. *Fifth*, we observed that test deletions and additions are often refactorings. *Finally*, a considerable portion of real additions occur for the purpose of augmenting a test suite to make it more adequate, which lends support for research on the topic of test-suite augmentation.

We are considering two main future research directions for this work. One direction involves extending our empirical study by considering additional programs and, especially, additional static and dynamic analysis techniques that can help refine our results. In particular, we plan to use code-clone detection techniques to better distinguish tests that are truly deleted or added from tests that are renamed or moved to different classes. The second direction involves research on actual test repair driven by our empirical results. A particularly interesting research problem, in this context, is the development of repair techniques that are *intent-preserving*, that is, techniques that ensure that the repaired tests preserve the "intent" behind the original tests. Often, tests can be repaired in different ways, and selecting the repair that most closely mirrors the original test intent is obviously desirable. The intriguing question is to what extent test intent can be characterized formally and accurately.

Acknowledgements

This work was supported in part by NSF awards CCF-0964647 and CCF-0916605 to Georgia Tech. Jim Whitehead and his colleagues provided us with their Kenyon tool [2], which we used to extract some of the `PMD` versions used in our study.

References

[1] N. Alshahwan and M. Harman. Automated session data repair for web application regression testing. In *Proceedings of the International Conference on Software Testing, Verification, and Validation*, pages 298–307, 2008.

[2] J. Bevan, E. J. Whitehead, K. Sunghun, and M. Godfrey. Facilitating software evolution research with Kenyon. In *Proceedings of the 10th European Software Engineering Conference / 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 177–186, 2005.

[3] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *Proceeding of the 33rd International Conference on Software Engineering*, pages 121–130, 2011.

[4] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso. WATER: Web Application TEst Repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, pages 24–29, 2011.

[5] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *Proceedings of the International Conference on Software Maintenance*, pages 359–369, 1996.

[6] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of the 30th International Conference on Software Engineering*, pages 481–490, 2008.

[7] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. ReAssert: Suggesting repairs for broken unit tests. In *Proceedings of the International Conference on Automated Software Engineering*, pages 433–444, 2009.

[8] B. Daniel, T. Gvero, and D. Marinov. On test repair using symbolic execution. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 207–218, 2010.

[9] D. Dig and R. Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution*, 18:83–107, March 2006.

[10] D. Gopinath, M. Z. Malik, and S. Khurshid. Specification-based program repair using SAT. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 173–188, 2011.

[11] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving gui-directed test scripts. In *Proceedings of the 31st International Conference on Software Engineering*, pages 408–418, 2009.

[12] J. Henkel and A. Diwan. CatchUp!: Capturing and replaying refactorings to support API evolution. In *Proceedings of the 27th International Conference on Software Engineering*, pages 274–283, 2005.

[13] S. Huang, M. B. Cohen, and A. M. Memon. Repairing GUI test suites using a genetic algorithm. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, pages 245–254, 2010.

[14] A. M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Transactions on Software Engineering and Methodology*, 18:1–36, 2008.

[15] A. M. Memon and M. L. Soffa. Regression testing of GUIs. In *Proceedings of the 9th European Software Engineering Conference and the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 118–127, 2003.

[16] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: Generating program transformations from an example. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 329–342, 2011.

[17] M. Mirzaaghaei, F. Pastore, and M. Pezzé. Supporting test suite evolution through test case adaptation. In *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation*, pages 231–240, 2012.

[18] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen. A graph-based approach to API usage adaptation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 302–321, 2010.

[19] R. A. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proceedings of the IEEE and ACM International Conference on Automated Software Engineering*, pages 218–227, 2008.

[20] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. In *Proceedings ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 189–206, 2011.

[21] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 61–72, 2010.

[22] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374, 2009.

[23] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381, 2005.