Identifying Test Refactoring Candidates with Assertion Fingerprints

Zheng (Felix) Fang*
Microsoft Corporation
zfang@uwaterloo.ca

Patrick Lam
University of Waterloo
patrick.lam@uwaterloo.ca

ABSTRACT

Test cases constitute around 30% of the codebase of a number of large software systems. Poor design of test suites hinders test comprehension and maintenance. Developers often copy-paste existing tests and reproduce both boilerplate and essential environment setup code as well as assertions. Test case refactoring would be valuable for developers aiming to control technical debt arising due to copy-pasted test cases.

In the context of test code, identifying candidates for refactoring requires tedious manual effort. In this work, we specifically tailor static analysis techniques for test analysis. We present a novel technique, *assertion fingerprints*, for finding similar test cases based on the set of assertion calls in test methods. Assertion fingerprints encode the control flow around the ordered set of assertions in methods.

We have implemented similar test case detection using assertion fingerprints and applied it to 10 test suites for open-source Java programs. We provide an empirical study and a qualitative analysis of our results. Assertion fingerprints enable the discovery of tests that exhibit strong structural similarities and are amenable to refactoring. Our technique delivers an overall 75% true positive rate on our benchmarks and reports that 40% of the benchmark test methods are potentially refactorable.

CCS Concepts

•Software and its engineering → Automated static analysis; Software testing and debugging;

1. INTRODUCTION

Modern software systems often use unit tests to ensure proper code behaviour. We have found that test-intensive systems contain 30 lines of unit tests for every new 100 lines of system code [13]. Tests therefore make up a significant part of the codebase in modern systems. While coverage tools are ubiquitous, and while many tools exist for generating tests, we are aware of far fewer tools for statically analyzing test cases. Yet, based on the sheer number of lines of extant test code, test analysis and maintenance must consume a significant amount of developer resources, and have been cited as a threat to the viability of at least one project [15, Preface]. Furthermore, Bavota et al have found that the presence of code smells in tests hinders comprehension of test cases [3]. Our overall goal is to develop static analysis techniques specifically applicable to test code.

Test cases are often thought of as being self-contained entities. At a class level, JUnit test classes must run independently and take no input parameters. Inspired by this self-contained worldview, a common way to develop a set of related test cases is to copy-paste an existing test case, replacing the inputs and expected outputs. As system size increases, this lack of design manifests as technical debt. The presence of multiple copies of boilerplate code increases test maintenance overhead and propagates pre-existing errors.

As with system code, refactoring is a powerful tool for incrementally improving the design of test suites. Van Deursen et al were the first to propose refactoring test code [5]. Meszaros built on this work and, more generally, advocates for the application of agile software design techniques to test design. His book [15] explains strategy and tactics for developing maintainable test suites.

Our primary goal is to facilitate the refactoring of suites of unit tests by identifying suitable refactoring candidates. We describe the design and implementation of a tool that applies static analysis techniques to find similar test cases throughout a test suite. Our tool leverages the fact that most test cases rely on assertions to validate system state. We compute sets of assertion fingerprints for test cases and identify similar test cases using these fingerprints. We then show these results to the developer, along with contextual information, as potential candidates for refactoring.

A number of options exist for implementing test refactoring, once candidates have been found. First, developers may simply refactor tests using language features such as inheritance or generics. Test frameworks allow test classes to provide common setup and teardown methods for the test cases in a particular class. Tillman and Schulte have proposed *parametrized unit tests (PUTs)* to increase the expressive power of unit testing and to enable test reuse [22]. Similarly, Saff proposed *theories* to simplify and increase the robustness of unit tests [20].

We believe that refactoring existing test suites to improve their design is valuable. According to Saff, the use

*This work was done while the author was a student at the University of Waterloo.

PPPJ '15, September 08 - 11, 2015, Melbourne, FL, USA

ISBN 978-1-4503-3712-0/15/09...$15.00

DOI: http://dx.doi.org/10.1145/2807426.2807437

of theories reduces the long term maintenance cost for test suites. Moreover, Thummalapenta et al. conclude from an empirical study of existing test suites that parametrization is beneficial—their results indicate that test suites, when retrofitted with parametrization, can detect new defects and provide increased branch coverage [21]. Test refactoring, either with language extensions or with standard language features, generally reduces the brittleness and improves the ease-of-understanding of test code.

However, test refactoring technologies currently require the developer to manually identify opportunities for refactoring, which becomes increasingly difficult as test suite size grows. Refactoring candidates are already difficult to manually identify across different test classes, let alone across packages. While writing new tests using test refactoring techniques is often the correct long-term decision, it is not always clear when such techniques apply. Incremental evolution and short-term imperatives may slowly lead to situations where technical debt, due to copy-pasted test cases, gets out of control. The output of our tool—sets of similar test cases, or clones—enables developers to regain control, refactor tests, and thus improve the quality of their test suites.

We present a technique for statically analyzing suites of unit tests to detect potentially refactorable test cases. We implemented our technique using the Soot program analysis framework [12]. Our tool analyzes test suites and displays sets of similar test methods along with the evidence, in terms of specific program fragments, used to draw this inference.

Using our tool, we conducted an empirical study based on a suite of 10 benchmark programs (ranging from 8,000 to 246,000 lines of code) with test suites ranging from 6,000 to 57,000 lines of test code, and 12,332 test methods in all. Our technique finds that the benchmarks have from 19% to 70% potentially refactorable test methods. Suite-wide, 44% of methods are similar to at least one other method in the suite. We manually inspected 191 randomly sampled refactoring candidates (as identified by our tool) and found that 75% of our recommendations appeared to be copy-pasted from other methods.

Our primary contributions[1] are:

- the application of static analysis techniques towards test code, in this case for identifying similar test cases;
- the concept of an assertion fingerprint set to summarize key aspects of a unit test; and
- an empirical study of a significant suite of benchmark programs using our technique for identifying similar unit tests.

Our technique computes, for each test method, an ordered set of fingerprinted assertions. It then identifies sets of tests with identical ordered sets of assertions, and filters out likely false positives. Finally, our display tool shows the computed sets of similar tests to the developer. Our tool enables many viable test refactorings and reports few false positives.

Our tool is available under the GNU GPL at:

https://bitbucket.org/felixfangzh/similartestanalysis

and our benchmark set at:

http://patricklam.ca/files/pppj15-benchmarks.tgz.

[1]Supplementary results available in the companion thesis [7].

```
1  public void testNominalFiltering() {
2     m_Filter = getFilter(Attribute.NOMINAL);
3     Instances result = useFilter();
4     for (int i = 0; i < result.numAttributes(); i++)
5        assertTrue(result.attribute(i).type() != Attribute.
               NOMINAL);
6  }

1  public void testStringFiltering() {
2     m_Filter = getFilter(Attribute.STRING);
3     Instances result = useFilter();
4     for (int i = 0; i < result.numAttributes(); i++)
5        assertTrue(result.attribute(i).type() != Attribute.
               STRING);
6  }
```

Figure 1: Similar test methods (set of size four; `NOMINAL` and `STRING` shown) from Weka's test suite, reported by our tool. Unshown `NUMERIC` and `DATE` methods are analogous.

```
1    static final int [] filteringTypes = {
2      Attribute.NOMINAL, Attribute.STRING,
3      Attribute.NUMERIC, Attribute.DATE
4    };
5
6    public void testFiltering() {
7      for (int type : filteringTypes)
8        testFiltering(type);
9    }
10
11   public void testFiltering(final int type) {
12     m_Filter = getFilter(type);
13     Instances result = useFilter();
14     for (int i = 0; i < result.numAttributes(); i++)
15       assertTrue(result.attribute(i).type() != type);
16   }
```

Figure 2: Refactored `testFiltering()` motivating example from Figure 1 (using a Test Utility Method in standard Java).

2. APPROACH

Our technique detects potentially refactorable methods in a test suite by collecting and matching sets of assertions across test methods. Our key insight is that assertions form a fundamental part of the structure of JUnit tests. In this section, we define the notion of an assertion fingerprint and explain how it helps identify similar test methods.

An assertion fingerprint summarizes the parameters of an assertion along with information about the assertion's control-flow reachability. A test method's assertion fingerprint set is an ordered set of assertions. Our tool computes assertion fingerprint sets, collects methods with matching fingerprint sets, and filters out common false positive patterns.

2.1 Motivating Example

Our goal is to report candidates for refactoring to the developer. Intuitively, our analysis declares that method m_1 is similar to method m_2 if they contain similar assertions (or related calls to `fail()`) with similar control flow.

Figure 1 presents an excerpt from one set reported by our analysis. The full set contains 4 similar methods from Weka's test suite (which test nominal, string, numeric, and

date filtering). Manual inspection confirms that these methods are indeed viable refactoring candidates. Figure 2 shows the result of an Extract Method refactoring of the code from Figure 1, creating a Test Utility Method. While, after refactoring, individual tests become more complicated to understand, we argue that the suite of 4 refactored tests is collectively easier to understand and maintain; Meszaros advocates using a Test Utility Method "whenever test logic appears in several tests and we want to be able to reuse that logic" [15].

Our analysis identifies the set in Figure 1 by matching fingerprints from the assertion invocations in the methods. In this case, all four test methods only have one static assertion invocation, `assertTrue(boolean)`.

We next compute assertion fingerprints for each of the assertions. Assertion fingerprints consist of 5 components; we illustrate branch counts and merge counts here. Each assertion is reachable from its method start through a single branch, `{i < result.numAttributes()}`, and thus has a branch count of 1. Furthermore, there is an implicit branch and merge in the boolean comparison in the assertion parameter (`{.type() != ...}`), which leads to a merge count of 1 for each assertion. Also, the assertion is in a `for` loop. Finally, these assertions are not in catch blocks or caught by exception handlers. This gives assertion fingerprints `{(bc:1, mc:1, inLoop:true)}` for all asserts. (Although we include predicates in Figure 3 for expository reasons, they do not play any role in the assertion fingerprint.)

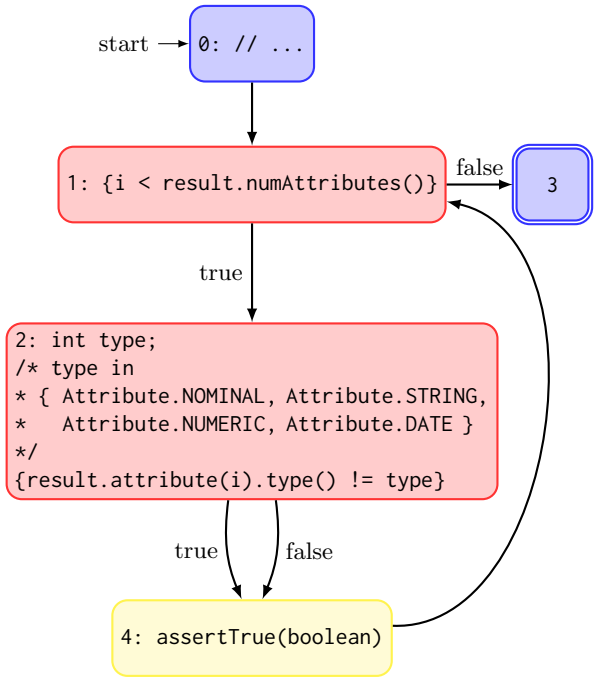

Figure 3: A control-flow graph for the example in Figure 1. Only assertions and predicates included.

Because all four of the test methods share the same ordered set of assertion fingerprints, and no other methods in the suite have the same fingerprints, our technique reports that the test methods in Figure 1 are similar, and a developer can then refactor the code to that in Figure 2.

2.2 Assertion Fingerprints

We next describe all of the parts of our assertion fingerprints. Our analysis computes a fingerprint for each JUnit assertion or fail call. It then collects the set of assertion fingerprints for a method to form the method's assertion fingerprint set, which is ordered using the method's statement ordering (as written by the developer in the source code). Our analysis then uses assertion fingerprint sets, along with the parameter types in the assert calls, to identify similar methods. Fingerprints include 5 attributes for each assertion: a branch count; a merge count; an exceptional successors count; and two boolean flags, indicating whether the given assertion is in a loop or a catch block. These attributes characterize the control flow reaching each assertion.

The intuition behind the components of an assertion fingerprint are as follows. Normal control flow edges, as captured in the branch and merge counts and the loop flag, are important to understanding the control-flow structure of test methods and how assertions make up test methods. In particular, they summarize the control-flow features that an execution of the test must traverse to reach each assertion. Section 4.3 further discusses the relevance of merge counts. We also include exceptional control flow in our fingerprints after noticing that try and catch blocks are common in some of our test suite benchmarks, particularly JDOM.

Our implementation uses the Soot framework [12] to statically analyze test suites, generate assertion fingerprints, and compute sets of similar methods. Soot provides a control-flow graph (CFG) over three-address code for each (test) method, as well as functions for computing useful properties such as dominators. Each control-flow graph has a distinguished start vertex (node 0 in Figure 3), which corresponds to the first statement in the method.

Branch Count.

The most straightforward positional attribute of a vertex is its branch count.

Definition 1. *The* branch count *of a vertex n is the minimal number of branches needed to reach that vertex from the start of a method, excluding n.*

We traverse the control-flow graph from its head, annotating each vertex with its branch count the first time that vertex is visited. (A vertex may be visited multiple times if there are multiple paths from the start vertex to that method.) Every time our traversal encounters a branch, we increment the branch count. Note that we never decrement the branch count; instead, we account for merges by increasing the merge count.

In Figure 3, vertex 1 is a branch. We increment the branch count at all of its successors. Vertex 2 is also a branch.

Merge Count.

The merge count is dual to the branch count. We ignore vertices with more than 2 successors or predecessors.

Definition 2. *A* branch vertex *is a vertex b in the control-flow graph with two successors b^+ and b^-. A* merge vertex *m_b for b is a vertex that is a transitive successor of both b^+ and b^- such that the sum of the path lengths from b to m_b through b^+ and b^- is minimal.*

In Figure 3, vertex 4 is the merge vertex for vertex 2. There is no merge vertex for vertex 1.

Definition 3. *The merge count for a vertex n is the minimal number of merge vertices needed to reach n from the start of the method, including n.*

The merge count at nodes 0, 1, 2, and 3 is 0, while the merge count at node 4 is 1.

Once again, we consider the shortest paths because there may be multiple paths to reach a vertex in the CFG.

Exceptional Successors.

We determine the number of exceptional successors of an assertion by examining the exceptional CFG of the method body. Soot's exceptional CFG augments the regular CFG with exception edges at all CFG vertices; this CFG represents possible exceptions and their handling. We consider only the number of exceptional successors and not the types of exceptions being handled.

Loops & Catch Block Indicator Flags.

We use a depth-first search on the CFG to determine whether an assertion is in a loop. We determine whether an assertion is in a catch block using dominator analysis and examining whether any exception handler statement is a dominator of the assertion call in the CFG.

On approximation.

Our technique collects fingerprints for the assertions in each test method and then identifies methods containing the same assertion fingerprints (in the same order) as being similar. It would also be possible to use graph isomorphism on control-flow graphs (or slices thereof). However, our fingerprinting technique incorporates approximation, thus capturing more results than a strict isomorphism-based approach, while empirically showing a low false positive rate. (Section 3 discusses our estimate of the false positive rate.)

Example.

The code in Figure 4, and its CFG in Figure 5, illustrate all five components of a fingerprint. For expository reasons, we have included fingerprints for each vertex v in Figure 5 (even though only fingerprints for assertion invocations matter for our technique). We include predicates for expository reasons; we do not use them in our algorithm.

Observe that the branch vertex at line 3 (`{i < 10}`) increments the branch count for its successors at line 4 and 5 (inside the for loop) and those at line 10 and 12 (outside the for loop). Line 6 is represented as two vertices, both of which are merge vertices. One merges the two successors of `{i == 2}` (lines 4 and 5), while the other merges `{i != 10}` (introduced at line 6). Although line 10 (`fail(String)`) is actually unreachable and hence does not throw any exceptions, Soot includes its exceptional successors anyway.

2.3 Building Sets of Similar Methods

After computing ordered sets of assertion fingerprints, we then partition a suite's test methods into sets of similar methods. Methods in the same set all have the exact same ordered set of assertion fingerprints. We match methods based on ordered sets of assertions because our key assumption is that test methods that verify state, using asserts, in the same order are likely to be refactorable.

```
1  public void test() {
2     int i;
3     for (i = 0; i < 10; ++i) {
4        if (i == 2)
5           assertEquals(i, 2);
6        assertTrue(i != 10);
7     }
8     try {
9        throw new Exception();
10       fail("Should have thrown exception");
11    } catch (final Exception e) {
12       assertEquals(i, 10);
13    }
14 }
```

Figure 4: Java code used to illustrate assertion fingerprints.

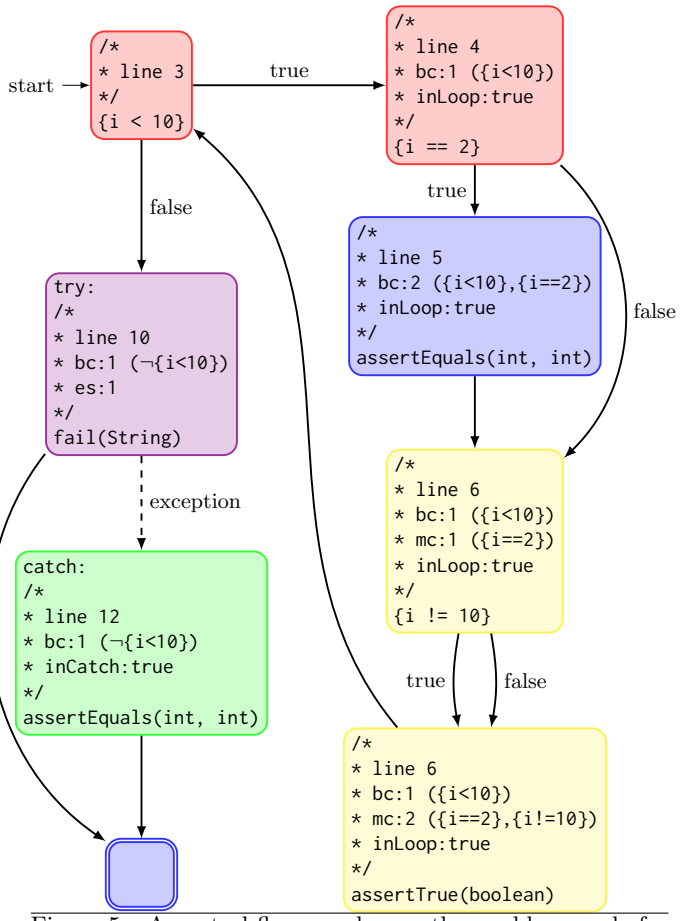

Figure 5: A control-flow graph over three-address code for the example in Figure 4. We elide all but assertions and predicates. Notations `bc`, `mc`, `es`, `inLoop`, `inCatch` represent branch count, merge count, exceptional successors, whether in loops or not, and whether in a catch block or not.

Filtering.

To mitigate the false positive rate, we filter out sets unlikely to be refactorable. Manual analysis of the results showed that the ordered set of assertions of a true positive set generally satisfy at least one of:

1. must contain some control flow—this is enforced by

requiring that at least one component of an assertion fingerprint is either greater than 0 or true;

2. must contain more than 4 assertions; or
3. must be heterogeneous in signature (invoke different assertion types, e.g. `assertEquals(String, double, double, double)` and `assertEquals(int, int)`).

We only report sets which satisfy at least one of the above conditions.

This filtering eliminates sets containing tests that are not likely to be related. Condition 1 eliminates tests that are too simple; Condition 2 guarantees that test methods are large enough to be interesting; Finally, Condition 3, when the assertion calls do not satisfy conditions 1 or 2, empirically helps avoid false positives on simple (homogeneous) but long test methods.

3. RESULTS

We implemented our technique to evaluate its efficicacy and have released our tool and benchmark set (see Section 1 for the URLs). Table 1 briefly describes each of the benchmarks. Our tool takes from 25 seconds (jdom) to 4 minutes (Google Visualization) to run on an Intel Core i7-2600K processor. These benchmarks have manually-generated test suites; as we describe later, some of the test suites have already been manually refactored, while others have not.

Our technique works on any JUnit test suites, although we expect it to be most effective for unrefactored manually-generated test suites. We present an empirical study of the results, a qualitative analysis of our technique's efficacy, and a sampling-based investigation of the false-positive rate.

In our sampling-based investigation, we manually judged whether tests were similar enough to be potentially refactorable. *True positives* were cases where all tests in a set were potentially refactorable; *false positives* had no pair of tests in a set that could be refactorable; and *fragmented true positives* contained at least one pair of refactorable tests. For instance, we found that some sets contained obviously cut-and-pasted clones, while others were obviously false positives. (Consider methods with just a single `assertTrue()` call; our technique explicitly filters out such methods, as described in Section 2.3, but they would otherwise be included in the results.)

Our technique identifies 44% of the test methods in our benchmark suites as being similar to some other test method in that suite. Our technique also enjoys a low false positive rate; 75% of the results reported by our technique appear to be similar enough to be refactorable (potentially with language extensions beyond plain Java 8).

3.1 Empirical Study

Table 2 presents statistical properties of our sets of similar methods, including counts of methods, classes, and packages per set. It also presents medians, means, and (assuming a normal distribution) standard deviations (σ) for these counts. Our potentially-refactorable sets are small and relatively local: the median number of methods in a set is 3, classes 2, and packages 1.

Test Methods.

Table 2 shows that our system identifies 5476 test methods (44% of all test methods in 10 test suites) as belonging to sets of similar methods. However, the mean masks a wide dispersion; our sets contain from 2 to 201 methods, and the suite-wide standard deviation in the number of methods per set is 12.3. Joda-Time and JFreeChart, with standard deviations of 19.3 and 16.6 in the number of methods per set, are particularly widely dispersed.

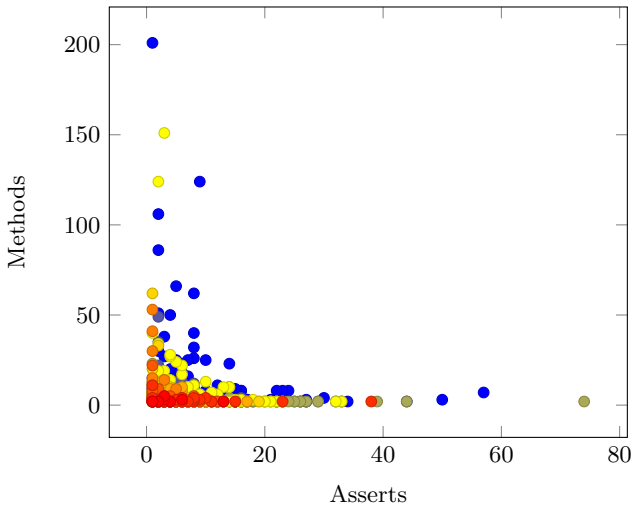

Figure 6: Distribution of set sizes. Each point represents one set, showing the number of asserts common to members of that set on the x-axis, and its distribution (# of similar methods) on the y-axis. 98.26% of sets have fewer than 40 assertions and 50 methods.

Distribution of Assertions.

Since our technique focuses on assertions, we computed per-benchmark statistics counting how many assertions belonged to a set of similar methods (Table 3). Our results indicate that 44% of assertions in the suite belong to some set (the same as the number of methods which belong to a set). Our sets of similar methods contain a median of 4 and a mean of 6.2 assertions. Assuming a normal distribution, the standard deviation is 6.6.

Similar Method Set Size (methods and assertions).

Figure 6 presents a scatter plot summarizing the sizes (in terms of methods and asserts) of our sets. Most of the sets reside in the lower left quadrant: 98% of sets have fewer than 40 assertions and fewer than 50 methods. We find no sets with both lots of methods and lots of assertions.

Estimating True Positive Rate via Sampling.

Since it is impractical to manually investigate the false positive rate for 978 sets, we used random sampling to estimate our false positive rate. For 9 of our benchmarks, we randomly drew 20 reported similar method sets and manually classified them as true positives, false positives, or fragmented true positives. The other benchmark only included 11 sets and we manually classified all of them. Table 4 shows the manual sampling results. To compute confidence intervals, we assume that the underlying distribution of true/false positives is binomial.

Table 1: Our evaluation included 10 benchmark suites, all open-source projects with test suites.

	Version	Test LOC	Total LOC	
Apache POI	v3.9	86 113	247 799	Java API for Microsoft documents
Commons Collections	v3.3	46 129	110 394	Implementation of Collections data structures
Google Visualization	v1.1.2	13 440	31 416	Data visualization framework
HSQLDB	v2.2.9	30 481	32 208	Relational database
JDOM	v1.1.2	25 618	76 734	Java API for manipulating XML data
JFreeChart	v1.0.15	93 404	317 404	Charting library
JGraphT	v0.8.3	12 142	41 801	Graph theory objects and algorithms
JMeter	v2.8	20 260	182 293	Performance testing and measurement framework
Joda-Time	v2.0	67 978	134 758	Date and time library
Weka	v3.7.8	26 270	495 198	Machine learning framework

Table 2: On our benchmark suite, 44% of test methods resemble other test methods in the same benchmark. Detected sets of similar methods tend to have few methods (3), classes (2), and packages (1). % Methods in Sets denotes the number of methods which belonged to a set of similar methods.

	Sets	Test Methods	% Methods in Sets	Methods/Set			Test Classes	Classes/Set			Packages	Packages/Set		
				Median	Mean	σ		Median	Mean	σ		Median	Mean	σ
Apache POI	185	747	26	2	4.0	5.1	571	2	3.1	4.4	411	2	2.2	1.8
Commons Collections	166	501	46	2	3.0	2.2	354	2	2.1	1.8	267	1	1.6	1.0
Google Visualization	35	142	37	3	4.1	3.8	83	2	2.4	1.4	54	1	1.5	0.8
HSQLDB	65	298	40	2	4.6	8.5	99	1	1.5	1.8	75	1	1.2	0.5
JDOM	29	82	31	2	2.8	1.7	37	1	1.3	0.7	32	1	1.1	0.3
JFreeChart	146	1066	49	3	7.3	16.6	900	2	6.2	16.3	373	2	2.6	2.9
JGraphT	11	27	19	2	2.5	0.9	11	1	1.0	0.0	11	1	1.0	0.0
JMeter	60	196	34	2	3.3	2.2	90	1	1.5	1.2	83	1	1.4	1.0
Joda-Time	231	2113	58	4	9.1	19.3	708	2	3.1	3.6	267	1	1.2	0.5
Weka	50	304	70	3	6.1	9.7	139	2	2.8	4.0	100	1	2.0	2.2
Total	978	5476	44	3	5.6	12.3	2992	2	3.1	7.1	1673	1	1.7	1.6

Efficacy of Filtering.

We manually investigated all 197 sets that were filtered out by the technique described in Section 2.3. Overall, 24% of the filtered-out results were false positives and 63% were fragmented true positives. Removing these results increased the quality of our reported results. Filtering worked exceptionally well on POI (40% false positives and 58% fragmented true positives on 43 removed sets) but poorly on Weka (43% true positives in 7 sets removed).

Beyond false positives and highly-fragmented true positives, 13% of the true positive results removed by the filter appeared to be copy-pasted. However, these true positive sets contained small sets of straight-line homogeneous assertions. These sets therefore likely contain ubiquitous clones (Section 4.2) and would typically be difficult or not worth the effort to refactor. We believe such sets provide little value to the developers and should be ignored.

3.2 Qualitative Analysis

We randomly selected 191 sets (all 11 sets from JGraphT and 20 from each of the other benchmarks) for a manual analysis, including a false positive determination and a qualitative inspection. (Section 3.3 will present our manual analysis of the largest sets in our benchmark suite.) Using our best judgment, we determined whether each of these would be potentially refactorable (using the techniques described in Section 4.2, notably Extract Method or parametrization) or whether they were false positives. Table 4 summarizes per-benchmark false positive rates. We report true/false positive rates as $(r \pm c)\%$, where r is the true/false positive rate of the samples, and c is the confidence interval on the reported true/false positive rate. Overall, the samples demonstrated high true positive rate $(75 \pm 5)\%$ and low false positive rate $(10 \pm 5)\%$. Our technique worked particularly well on JGraphT (100% true positives) and Joda-Time $(95 \pm 10)\%$ true positives, but less well on Google Visualization $(60 \pm 15)\%$ and Apache POI $(30 \pm 20)\%$. The remaining benchmarks showed true positive rates between 70% and 85%.

JGraphT.

JGraphT is a Java library that implements graph theory objects and algorithms. Our analysis found 11 sets in the 54 test classes (14 packages) of the JGraphT test suite (version 0.8.3). All reported sets are refactorable. We were surprised by the high quality of JGraphT's sets, since the matched assertion fingerprints are fairly simple. For instance, one of our sets includes the assertion fingerprint consisting of a single `assertEquals(String, String)` with 1 exceptional successor. This set contains two similar test methods, `testGraphReader()` and `testGraphReaderWeighted()`, and no other methods. It appears that our technique performs well on JGraphT's test suite because this suite is composed of heterogeneous test methods and is fairly small (diminishing the likelihood of getting false positives).

JDOM.

JDOM is a Java-based document object model library for

Table 3: Between 17% and 73% of the assertions in a benchmark, depending on the benchmark, belong to some set of similar methods. Coincidentally, 44% of assertions also belong to some set of similar methods. Median set contains 4 assertions.

	Sets	Assertions	% Assertions in Sets	Assertions/Set Median	Mean	σ
Apache POI	185	906	17	3	4.9	4.8
Commons Collections	166	1217	46	5	7.3	8.8
Google Visualization	35	147	23	4	4.2	2.8
HSQLDB	65	213	37	2	3.3	3.0
JDOM	29	205	31	5	7.1	7.6
JFreeChart	146	1031	56	5	7.1	5.7
JGraphT	11	37	18	2	3.4	3.4
JMeter	60	271	31	3	4.5	3.4
Joda-Time	231	1923	73	6	8.3	7.9
Weka	50	141	60	3	2.8	1.8
Total	978	26952	44	4	6.2	6.6

Table 4: Sampling-based investigation shows that 75% of the reported sets are true positives. Confidence intervals included for 95% confidence level.

	Sets	Sample Size	% True Positives in Samples	% Fragmented True Positives in Samples	% False Positives in Samples	Confidence Interval (%)
Apache POI	185	20	30	15	55	20
Commons Collections	166	20	85	10	5	16
Google Visualization	35	20	60	35	5	15
HSQLDB	65	20	75	15	10	17
JDOM	29	20	85	15	0	9
JFreeChart	146	20	70	15	15	20
JGraphT	11	11	100	0	0	0
JMeter	60	20	85	15	0	14
Joda-Time	231	20	95	5	0	10
Weka	50	20	80	15	5	15
Total	978	191	75	15	10	5

XML. The JDOM test suite (version 1.x) includes 16 test classes in 3 packages. Its tests often use try/catch blocks to ensure that exceptions are thrown as expected. Assertion fingerprints' inclusion of try/catch blocks thus work particularly well for JDOM—its true positive rate is (85% ± 9%).

Joda-Time.

Joda-Time is a library for manipulating dates and times. The Joda-Time test suite (version 2.0) consists of 121 test classes in 6 packages. The samples showed that the Joda-Time test suite contains many sets of similar test methods, which contain straight-line assertion fingerprints. Figure 7 presents an example set. This set includes tests `Test{Buddhist,Coptic,Ethiopic,GJ,Gregorian}Chronology`, all of which have identical structure and can be refactored using theories. This example also shows that assertion fingerprints can leverage the *lack* of control-flow structures to match test methods.

```
1 public void testEquality() {
2   assertSame(BuddhistChronology.getInstance(TOKYO),
3              BuddhistChronology.getInstance(TOKYO));
4   // [... 4x assertSame(BuddhistChronology,
        BuddhistChronology)]
5 }
```

(a) `testEquality()` for `TestBuddhistChronology`.

```
1 public void testEquality() {
2   assertSame(CopticChronology.getInstance(TOKYO),
3              CopticChronology.getInstance(TOKYO));
4   // [... 4x assertSame(CopticChronology, CopticChronology)]
5 }
6 }
```

(b) `testEquality()` of `TestCopticChronology`.

Figure 7: Two implementations of `testEquality()` make up one set from the Joda-Time test suite. Both contain straight-line assertion fingerprints.

Apache Commons Collections.

The Apache Commons Collections augments the Java Collections Framework with additional data structures. The 151 test classes in the 11 packages of the Apache Commons Collections test suite (version 3.3) include a number of refactorable tests. Furthermore, some test

methods are exactly identical (modulo whitespace) but reside in different classes and hence invoke different helper code. In Figure 8, test methods `testIterator()` of classes `Test{ArrayIterator, IteratorChain, ObjectArrayIterator, UniqueFilterIterator}` are textually identical. However, their containing classes implement different `makeFullIterator()` methods and `testArray` arrays. Hence, the tests exercise different data structures despite being identical. This implementation pattern makes the test methods in the Apache Commons Collection highly refactorable.

```
1  public void testIterator() {
2    Iterator iter = (Iterator) makeFullIterator();
3    for (int i = 0; i < testArray.length; i++) {
4      Object testValue = testArray[i];
5      Object iterValue = iter.next();
6      assertEquals("Iteration value is correct", testValue,
           iterValue);
7    }
8    assertTrue("[...] should now be empty", !iter.hasNext());
9    try {
10     Object testValue = iter.next();
11   } catch (Exception e) {
12     assertTrue(
13        "NoSuchElementException must be thrown",
14        e.getClass().equals((new NoSuchElementException()).
              getClass()));
15  }
16 }
```

Figure 8: Test method from the Apache Commons Collections, belonging to classes `Test{ArrayIterator, IteratorChain, ObjectArrayIterator, UniqueFilterIterator}`. At runtime, `makeFullIterator()` (line 2) and the value of `testArray` (line 3, 4) bind different values depending on the containing class.

Weka.

Weka is a machine learning software suite. The Weka test suite (version 3.7.8) consists of 98 test classes in 17 packages. Weka has classes that implement similar machine learning techniques in both supervised and unsupervised variants. One example is `weka.filters.supervised.attribute.DiscretizeTest` vs `weka.filters.unsupervised.attribute.DiscretizeTest`. Both tests exercise the methods of class `Discretize` but one targets the supervised version and the other targets the unsupervised version. Test method `testTypical()` is textually identical between the supervised and unsupervised variants of `DiscretizeTest`. One can refactor this set by theorizing data points of supervised and unsupervised `Discretize` objects and applying the same test method.

JFreeChart.

JFreeChart is a library allowing the creation and display of professional quality charts in applications. The JFreeChart test suite (version 1.0.15) consists of 348 test classes in 23 packages, and in particular contains similar methods which test related types. Figure 9 shows that the test method named `testClear()` in classes `VectorSeriesTests` and `XIntervalSeriesTests` are identical but perform tests on `VectorSeries` and `XIntervalSeries`, respectively. (Unlike with the Commons Collection, `testClear()` only differs in terms of the calls to the class under test, not to self-calls on the test class.) Theories would be straightforward to apply to such test methods. Despite testing different data types, the example in Figure 9 appears to be refactorable.

```
1  public void testClear() {
2    ComparableObjectSeries s1;
3    // (a) in VectorSeriesTests:
4    // s1 = new VectorSeries("S1");
5    // (b) in XIntervalSeriesTests:
6    // s1 = new XIntervalSeries("S1");
7    s1.addChangeListener(this);
8    s1.clear();
9    assertNull(this.lastEvent);
10   assertTrue(s1.isEmpty());
11   s1.add(1.0, 2.0, 3.0, 4.0);
12   assertFalse(s1.isEmpty());
13   s1.clear();
14   assertNotNull(this.lastEvent);
15   assertTrue(s1.isEmpty());
16 }
```

Figure 9: JFreeChart contains `testClear()` methods which are identical except for the type under test.

Apache POI.

Apache POI is a Java API for Microsoft documents. Our technique worked poorly on this benchmark, yielding a low true positive rate of $(30 \pm 20)\%$ on the 50 test classes in 10 packages of Apache POI test suite (version 3.9). We noticed that several test methods have identical assertion fingerprints but are not refactorable. Hence, the Apache POI test suite exposes the limitations (Subsection 4.4) of assertion fingerprints. Specifically, because our technique does not gather information on assertion parameters, it cannot detect the differences between the arguments passed to different calls to `assertNotNull` and `assertEquals(int, int)`, making it susceptible to a certain class of false positives.

Google Visualization Data Source Library.

The Google Visualization Data Source Library enables the visualization of data sources. The test suite (version 1.1.2) contains 50 test classes in 10 packages. We found that this suite contains test methods involving complex query-related statements, consequently reducing the roles of assertions in a test method. Furthermore, this test suite also uses custom helper methods such as `void assertStringArraysEqual(String[] expected, String[] found)`. Google Visualization's test suite thus exploits limitations of our technique, resulting in a below-average true positive rate of $(60 \pm 15)\%$.

3.3 Large Sets

Figure 6 shows that 5 of our sets include over 100 methods each. These large sets have from 1 to 9 assertions, and belong to Joda-Time (3 sets) and JFreeChart (2 sets). We investigated them in more detail, and found that most of the reported methods in these large sets were refactorable true positives; 1 of the reported sets was fully true positive, while another 3 sets were fragmented true positives, with a majority of the contents of those sets being true positives.

The JFreeChart true positive set finds 151 methods with

3 matching `assertTrue(boolean)` calls; these calls succeed 1, 2, and 2 control-flow merges respectively. Manual inspection revealed that JFreeChart uses exactly the same code to implement a method called `testCloning()` (148) or `testCloning2()` (3) across different classes.

The remaining 3 sets include one from JFreeChart and 2 from Joda-Time. The JFreeChart set contains 124 methods with 2 straight-line asserts (`assertTrue(boolean)` succeeded by `assertEquals(int, int)`), 123/124 of which are all implementing `testHashcode()` with the same code. The Joda-Time examples are testing arithmetic operations and constructors with identical code; a few false positives sneak into these sets.

The one set containing false positives is a 201-method set with one assert inside a try. The assert is simply a call to `fail()`, with no message. This could be considered bad practice—there is no error message indicating the reason for the failure. This set is the only result that is very fragmented and contains smaller sets.

4. DISCUSSION

We next discuss several aspects of our results in greater depth. Previously, we presented counts of false and fragmented true positives; in this section, we discuss some potential causes of these non-true positive results. We also qualitatively discuss the refactorability of our reported sets. We conclude this section with a survey of limitations and threats to validity.

4.1 False and Fragmented True Positives

While our technique returns many sets of methods that we classify as refactorable, it also returns some sets that are less actionable—false positives and fragmented true positives.

False Positives.

False positives occur when two different methods have, by chance, the same assertion fingerprints. For instance, POI contains two methods, `testBasics()` and `testImageCount()`, which both have 4 straight-line assertions of the same types, but no conceptual similarity. False positive sets are unrefactorable and hence undesirable. Fortunately, they appear relatively rarely: they account for (10±5)% of the manually-investigated random sample and 13% of the filtered sets.

Fragmented True Positives.

Recall that fragmented true positives occur when a set contains some, but not all, refactorable test methods. For instance, one set in Commons Collections includes four methods with the same assertion fingerprint—`fail()` with 1 exceptional successor. This set confounds two 2-element sets. (A more complex algorithm could split such sets; however, simplicity was one of our design goals.) While counting fragmented true positives can approximate the applicability of refactoring in a benchmark suite, their lack of unity makes them difficult to work with and they are hence likely to be lower on the priority list for potential refactoring. We believe that it is helpful to the developer to filter out fragmented true positives whenever possible, and indeed, the sets removed by our filter are 63% fragmented true positive.

4.2 Refactorability of Similar Method Sets

Because our technique uses assertion fingerprints to group similar test methods, methods in the same set are similar in structure. (See Figures 1, 8, and 9 for examples.) This similarity should enable refactoring. In this work, we are agnostic to refactoring technology. Figure 2 showed the result of one refactoring. More generally, Meszaros provides an arsenal of techniques for test refactoring. The most relevant techniques are Test Utility Methods, Custom Assertions, Test Helpers, Testcase Superclasses, and Parametrized Tests [15, Chapter 24]; these techniques result in shorter test methods whose intent is more obvious. Parametrized unit tests and theories can also help. Nevertheless, some sets remain difficult to refactor. This difficulty surely contributes to the existence of clones in benchmark suites.

Non-parametrizable Sets.

Figure 10 demonstrates a non-parametrizable set. Despite similar structures, `testValueList_getByIndex()` tests the `get(int)` method of a `ListOrderedMap` object by iterating forward, whereas `testValueList_removeByIndex` removes items from a `ListOrderedMap` object while iterating until the `ListOrderedMap` object only contains one element. The difference in semantics makes the two test methods difficult or impossible to refactor.

```
1  public void testValueList_getByIndex() {
2    resetFull();
3    ListOrderedMap lom = (ListOrderedMap) map;
4    for (int i = 0; i < lom.size(); i++) {
5      Object expected = lom.getValue(i);
6      assertEquals(expected, lom.valueList().get(i));
7    }
8  }

1  public void testValueList_removeByIndex() {
2    resetFull();
3    ListOrderedMap lom = (ListOrderedMap) map;
4    while (lom.size() > 1) {
5      Object expected = lom.getValue(1);
6      assertEquals(expected, lom.valueList().remove(1));
7    }
8  }
```

Figure 10: A non-parametrizable set from the Apache Commons Collections test suite.

Ubiquitous Clones.

Ubiquitous clones are short methods with high frequency across a system [18]. We have discovered a few ubiquitous clones in our results. For example, copies of the 4-line test method `testToolTips` (Figure 11) exist in 10 test classes of the Weka test suite. Ubiquitous clones appear to be methods that perform short, specific tasks, or that were already refactored with helper methods and are now short. Ubiquitous clones are not only difficult to refactor because of their highly frequent appearance in the system, but also unlikely to be worth the effort to refactor because they are so short.

4.3 Relevance of Merge Counts

Recall that our assertion fingerprints include 5 components: branch count, merge count, exceptional successor count, and indicators for loops and catch blocks. In Section 2.2 we provided an intuitive justification for these components. Clearly, including the merge count (or any other

```
1 public void testToolTips() {
2   if (!m_GOETester.checkToolTips())
3     fail("Tool tips inconsistent");
4 }
```

Figure 11: A member of an ubiquitous clone set from Weka. Identical copies of `testToolTips()` exist in 10 test classes.

component) in assertion fingerprints can only decrease the number of reported assertion sets. This lowers the false positive rate at the expense of recall. Not using merge counts may increase the false positive rate but also the recall. We continue with an empirical examination of the merge count specifically.

In the JGraphT benchmark, using the merge count omits two assertion sets from the results. One of the two sets is a false positive. The other set is a true positive. In the end, the user could choose to enable or disable merge counts depending on their needs. If the result sets are small and the user would like to see more similar test sets, the user can disable merge counts. In the more likely scenario where our tool returns many results, we estimate that the user would like to avoid false positives as much as possible. In that case, the user should stay with the default setting and enable merge counts as part of assertion fingerprints.

4.4 Limitations

The key assumption behind our technique is that test methods with the same assertion fingerprints are likely to be refactorable. Test suites that do not respect this assumption expose the limitations of our technique. Specifically, our technique works less well when the specific arguments to the assertion are important, or when assertions have already been refactored into helper methods. More generally, assertion fingerprints work best when the ordered set of assertions in a method is important, and less well when tests use other means to verify program behaviour.

Assertion Arguments.

Assertion fingerprints focus on the structure of the test methods and not on the data flowing to the particular assertions. We observed that this worked well for most of our test suites. However, Apache POI's test suite would require finer-grained information for best results.

Helper Methods.

Occasionally, test methods are already somewhat refactored and use helper methods to invoke relevant assertions. Our technique currently relies on bare assertions appearing in test methods; it does not understand Custom Assertions [15]. Our technique is unaware of helper methods' effects and does not include the assertions of callees in assertion fingerprints, which may cause additional false positives. Test methods that already use helper methods are unlikely to be worth the effort to refactor.

4.5 Threats to Validity

The selection of 10 benchmarks from different application areas (graph theory, machine learning, data structures, etc) aims to mitigate threats to external validity. However, no benchmark suite is exhaustive and captures all possible test code styles, especially those specific to certain applications or domains. Also, our benchmark suite size does not exceed 270 kLOC (lines of code in the test suites). Larger benchmarks may have different properties than those in our set.

We have not investigated non-Java or non-JUnit benchmarks. Our results might not generalize to such test suites.

The major threat to internal validity in our case is from confounding effects. Our results show that similar assertion fingerprints are correlated with refactorable test methods. However, the assertion fingerprints may simply be reflecting some other property of the code.

A threat to construct validity is that the clone sets were manually analyzed, using a subjective 3-point scale (true positive, fragmented true positive, false positive).

We believe that we have adequately mitigated the threats to validity through benchmark selection and the use of definitions of clone characteristics. The result is an accurate assessment of our technique's performance on JUnit suites.

5. RELATED WORK

Proper agile development techniques require tests to be developed concurrently with system code. In that vein, Van Deursen et al first proposed refactoring test code [5]. Meszaros followed up with an exhaustive description of design patterns for test code [15]. This prior work included instructions for manually refactoring to test code. It did not propose techniques for identifying potential refactorings.

Bavota et al investigated the incidence of test smells along with their impact [3]. Their work used a simple rule-based tool to identify properties of the test code (e.g. "uses external resources"), followed by manual classification. Other approaches use metrics, e.g. Van Rompaey et al [17] and Aniche et al [1]. Greiler et al [9] analyze method and field names, but not the code itself. Many of these works are purely empirical and describe extant test suites. Our goal is to use sophisticated static analysis techniques to produce tools to help developers. The previous work also showed that test smells harmed test suite comprehension, an issue that we empower developers to address through our results.

Guerra and Fernandes describe a framework for (ad-hoc manual) reasoning about test refactorings and present a tool to carry out simple refactorings [11]. Similarly, Estefo describes TestSurgeon [6], a tool for restructuring unit tests. Our tool can provide input to an automatic refactoring tool, but applying the appropriate refactorings appears to still be beyond the state of the art.

The most common automated test suite analysis is the ubiquitous code coverage tool. Such tools help developers increase coverage by pointing out where it is lacking. Combined static and dynamic approaches exist—Zhang et al proposed such an approach for test generation to improve structural coverage [24]. We do not consider coverage.

More relatedly, Greiler et al apply dynamic techniques to measure test case similarity [10]. However, their work attempts to match tests at different levels, namely end-to-end tests with unit tests, to aid test suite understanding—presumably, one could understand the high-level test more easily than the unit test. Our work instead finds lateral relationships between different unit tests, and supports developers when they refactor tests.

Recent research has also investigated automated and guided refactoring. However, our focus on tests is novel. Balazinska et al proposed an approach to refactor common parts of method clones, parametrizing their differences [2].

Their technique is limited to certain types of applicable clones. An alternative approach proposed by Volanschi [23] also covers other domain-specific and application-specific clones but requires manual intervention. Because we focus on test code, our technique would require less manual effort than Volanschi's technique on our domain.

5.1 Comparison with Clone Detectors

Our work aims to discover refactorable test methods using static analysis techniques. We therefore leverage several properties specific to the problem at hand, including the assertion structure of test methods and the fact that the unit of granularity is always going to be a method.

Clone detectors are another, more generic, tool for potentially finding refactorable test methods. Roy et al have surveyed existing clone detectors [19], and classify existing clone detection tools into four broad categories: textual, lexical, syntactic, and semantic. Our approach is closest to a syntactic approach in that it operates on control-flow graphs. However, our focus on test cases' assertion fingerprints bakes in a selection of features that we believe to be most relevant, and our results show that this emphasis results in better results. Related syntactic approaches are those of Baxter [4] and, more recently, the AST-based *ccdiml* clone detector of the Bauhaus suite (henceforth referred to as simply "Bauhaus").

Some syntactic approaches are metrics-based; for instance, the Datrix tool [14] computes metric values for each function and groups together functions with similar metrics. Our approach is similar to Datrix's in that we are sensitive to features such as control-flow loops. However, we collect features for each assertion—assertion fingerprints—rather than for a method as a whole, and we report methods with the same set of assertions and the same control-flow (i.e. the same set of assertion fingerprints). This is possible because of our focus on assertions as the key to understanding test methods, and we expect that it would yield better results. (Datrix is not available for a head-to-head comparison.)

We continue by comparing our technique to syntactic AST-based clone detectors, both in principle and empirically, via a head-to-head comparison with the best-in-class clone detector Bauhaus [16, 8]. We omit quantitative comparisons to Bauhaus; see [7] for full results.

Contextual Information.

As a byproduct of computing assertion fingerprints, our technique has additional information about why certain methods belong to clone sets. We make this information available to the developer. We believe that it will enable developers to better refactor the reported methods.

In particular, our tool reports the branch nodes which contribute to each assertion's branch count and merge count. It also reports quantitative information—counts of methods, classes, and packages for each clone set—that could hint at the relevance and refactorability of the reported clones.

Comparison with Bauhaus.

To further evaluate our results, we performed a head-to-head comparison of our sets with those from Bauhaus. Unlike our technique, Bauhaus operates by comparing tokens at source level and detects similar patterns throughout a codebase. We ran Bauhaus with a token threshold of 50 (normal for Bauhaus). Bauhaus's performance is subject to tuning the token threshold; a lower value detects more clones while giving more spurious results. Our technique does not require thresholds. Recall that our technique works at method level, while Bauhaus is oblivious to code structure and detects duplicate code within methods. If Bauhaus operated at method level rather than matching arbitrary AST fragments, then it would report fewer results. Although that would then enable the user to increase thresholds, there is no conceptual reason to believe that higher thresholds would yield better results.

Due to the unavailability of method-level AST tools, we do not report comparisons with such tools. However, we believe that our qualitative conclusions would still hold.

Inspecting randomly selected Bauhaus results revealed:

- because Bauhaus is insensitive to method boundaries, it detects more clone sets (5367) than we do (978). However, Bauhaus clone sets thus include arbitrary method fragments. We believe that our similar methods are easier to refactor; returning methods also helps avoid spurious results, particularly ubiquitous clones.

- our technique is unaffected by mid-clone textual differences that Bauhaus is sensitive to, as we operate on Java bytecode rather than source code. Hence, some of our clone sets are supersets of Bauhaus clone sets. (An AST-based clone detection tool that works at method granularity would not find clones with mid-method textual differences unless such differences were beneath its threshold.)

- our technique detected 443 clone sets (45% of our results) that Bauhaus missed. In particular, Bauhaus (with threshold 50) was not able to detect our motivating example (Figure 1).

- our technique misses clones where assertions get added or removed, as it requires exact matches on sets of assertion fingerprints. Bauhaus can find such clones by matching around missing assertions. Hence, some of our clone sets are subsets of Bauhaus clone sets. (Matching assertion fingerprint sets with edit distance of 1 caused too many false positives.) Requiring exact matches increases our precision but costs recall. Matching at method granularity would also cause Bauhaus to miss such clones, again unless they were beneath Bauhaus's threshold.

Our technique captures additional, high-quality, clones which are not reported by existing clone detectors.

6. FUTURE WORK

This work presented a technique for detecting similar tests in JUnit test suites. Two avenues for future work are its generalization to other test frameworks as well as the application of this work to test code refactoring.

Test Frameworks.

JUnit inspired many similar unit test frameworks. This family of frameworks is sometimes denoted xUnit, and includes NUnit, PHPTest, AUnit, and many others.

It is straightforward to generalize our technique to xUnit frameworks that support imperative languages. All of the above-listed frameworks would qualify. We suspect that our

technique would work about as well on these frameworks as it does for JUnit, but that claim is subject to empirical validation.

We expect that our technique would work less well on test frameworks for functional languages, since control-flow works differently for those languages. It is possible that developers would often have already refactored test cases in these languages, especially due to the dynamic nature of these languages.

Test Refactoring.

The present work is a useful first step towards test refactoring, both manual and automated. In both cases, it would be useful to develop techniques for prioritizing the resrults of the similar test analysis. We expect that a heuristic approach could work well enough to enable further work. To automatically refactor tests, we would examine high-priority sets and propose suitable program transformations.

7. CONCLUSIONS

Test suites account for a significant proportion of system code, yet are poorly served by static analysis tools. In this paper, we presented a novel application of static analysis techniques for detecting similar test cases. Our tool's output enables developers to improve the quality of their test suites by refactoring similar test cases.

Our technique works by computing assertion fingerprints, which summarize the control-flow surrounding assertion calls in test methods. It then partitions test methods into clone sets according to ordered sets of assertion fingerprints.

We implemented our technique and collected sets of similar methods for 10 open-source Java test suites, analyzing them via empirical study and qualitative analysis. Empirically, our tool reported that 44% of test methods in our suite were similar to other methods. Qualitatively, we verified that 1) many of our reported test methods are amenable to refactoring; and 2) control and exception flow is important for detecting similar test methods. More broadly, we believe that our research points towards future work that applies static analysis techniques to help developers understand and improve test suites.

Acknowledgments

Divam Jain collected our benchmark suite and worked on an earlier technique for detecting similar test methods. This research was supported by Canada's Natural Science and Engineering Research Council.

8. REFERENCES

[1] M. F. Aniche, G. A. Oliva, and M. A. Gerosa. What do the asserts in a unit test tell us about code quality? a study on open source and industrial projects. In *CSMR*, pages 111–120, 2013.

[2] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial redesign of Java software systems based on clone analysis. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, WCRE '99, pages 326–336, 1999.

[3] G. Bavota, A. Qusef, R. Oliveto, A. D. Lucia, and D. Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *ICSM*, pages 56–65. IEEE Computer Society, 2012.

[4] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance*, pages 368–377, November 1998.

[5] A. V. Deursen, L. Moonen, A. Bergh, and G. Kok. Refactoring test code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, pages 92–95, 2001.

[6] P. Estefo. Restructuring unit tests with testsurgeon. In M. Glinz, G. C. Murphy, and M. Pezzè, editors, *ICSE*, pages 1632–1634. IEEE, 2012.

[7] Z. Fang. Test clone detection via assertion fingerprints. Master's thesis, University of Waterloo, Sept. 2014.

[8] N. Gode. Evolution of type-1 clones. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 77–86, 2009.

[9] M. Greiler, A. van Deursen, and M. D. Storey. Automated detection of test fixture strategies and smells. In *ICST*, pages 322–331. IEEE, 2013.

[10] M. Greiler, A. van Deursen, and A. Zaidman. Measuring test case similarity to support test suite understanding. In C. A. Furia and S. Nanz, editors, *TOOLS*, volume 7304 of *LNCS*, pages 91–107. Springer, 2012.

[11] E. M. Guerra and C. T. Fernandes. Refactoring test code safely. In *ICSEA*, page 44. IEEE Computer Society, 2007.

[12] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, Galveston Island, TX, October 2011.

[13] P. Lam and Z. F. Fang. Beyond coverage: What lurks in test suites? In *GTAC*, 2014. http://goo.gl/tVGzy3.

[14] J. Mayrand, C. Leblanc, and E. M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software Maintenance*, pages 244–253, November 1996.

[15] G. Meszaros. *xUnit test patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[16] Project Bauhaus. http://www.bauhaus-stuttgart.de/. Last accessed July 2015.

[17] B. V. Rompaey, B. D. Bois, S. Demeyer, and M. Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *TSE*, 33(12):800–817, 2007.

[18] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report 2007-541, Queen's University, Kingston Ontario, Canada, Sept. 2007.

[19] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, May 2009.

[20] D. Saff. Theory-infected: Or how I learned to stop worrying and love universal quantification. In

Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion, OOPSLA '07, pages 846–847, 2007.

[21] S. Thummalapenta, M. R. Marri, T. Xie, N. Tillmann, and J. de Halleux. Retrofitting unit tests for parameterized unit testing. In *Proceedings of the 14th international conference on Fundamental approaches to software engineering: part of the joint European conferences on theory and practice of software*, FASE'11/ETAPS'11, pages 294–309, 2011.

[22] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 253–262, 2005.

[23] N. Volanschi. Safe clone-based refactoring through stereotype identification and iso-generation. In *Software Clones (IWSC), 2012 6th International Workshop on*, pages 50–56, 2012.

[24] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 353–363, 2011.