An Empirical Investigation into the Impact of Refactoring on Regression Testing

Napol Rachatasumrit

Supervised by: Dr. Miryung Kim

The University of Texas at Austin

May 4, 2012

Abstract

It is widely believed that refactoring improves software quality and developer's productivity by making it easier to maintain and understand software systems. On the other hand, some believe that refactoring has the risk of functionality regression and increased testing cost. This paper investigates the impact of refactoring edits on regression tests using the version history of Java open source projects: (1) Are there adequate regression tests for refactoring in practice? (2) How many of existing regression tests are relevant to refactoring edits and thus need to be re-run for the new version? (3) What proportion of failure-inducing changes are relevant to refactorings? By using a refactoring reconstruction analysis and a change impact analysis in tandem, we investigate the relationship between the types and locations of refactoring edits identified by RefFinder and the affecting changes and affected tests identified by the FaultTracer change impact analysis. The results on three open source projects, JMeter, XMLSecurity, and ANT, show that only 22% of refactored methods and fields are tested by existing regression tests. While refactorings only constitutes 8% of atomic changes, 38% of affected tests are relevant to refactorings. Furthermore, refactorings are involved in almost a half of failed test cases. These results call for new automated regression test augmentation and selection techniques for validating refactoring edits.

Acknowledgements

I would like to thank the advice and guidance of Dr. Miryung Kim, my supervisor. Without her knowledge and assistance, this study would not have been successful. I also thank Lingming Zhang for his help with running FaultTracer and analyzing its results. This work was in part supported by National Science Foundation under the grants CCF-1149391, CCF-1043810, and CCF-1117902, and by a Microsoft SEIF award. I also would like to thank Dr. Dewayne Perry for being the second reader of my thesis.

Contents

List of Figures

List of Tables

1 INTRODUCTION

Refactoring changes a software system in such a way that it does not alter the external behavior of the code but improves the modular structure of software [18]. It is widely believed that refactoring improves software quality and developer's productivity by making it easier to maintain and understand software systems [8]. Many believe that a lack of refactoring incurs technical debt to be repaid in the form of increased maintenance cost [2]. For example, eXtreme Programming claims that refactoring saves development cost and improves software quality [1] and advocates the rule of *refactor mercilessly* throughout the entire project life cycles. On the other hand, there exists a conventional wisdom that software engineers often avoid refactoring, when they are constrained by a lack of resources (e.g., right before major software releases). Some also believe that refactoring does not provide immediate benefit unlike new features or bug fixes, and incurs the risk of functionality regression and *increased testing cost*.

Recent empirical studies show contradicting evidence on the benefit of refactoring as well. Ratzinger et al. [23] found that, if the number of refactoring increases in the preceding time period, the number of defects decreases. On the other hand, Weißgerber and Diehl found that a high ratio of refactoring edits is often followed by an increasing ratio of bug reports [30, 31] and incomplete or incorrect refactoring edits cause bugs [9]. Though refactoring is defined as a semantics-preserving code transformation, Murphy-Hill et al. found that developers often interleave refactoring edits with other behavior-modifying edits and apply refactoring manually without using automated refactoring engines [20]. This practice could be error-prone, and thus requires developers to test code after refactoring. In our field study of refactoring at Microsoft, the survey participants also indicated that refactoring comes with

a risk of introducing subtle bugs and functionality regression. When a regression test suite is inadequate, it could prevent developers from initiating refactoring effort, because there is no safety net for checking the correctness of refactoring edits [15].

To understand the relationship between refactoring edits and regression testing, we apply a change impact analysis and a refactoring reconstruction analysis in tandem to version history data. We use an automated refactoring reconstruction tool, RefFinder to identify the types and locations of refactoring edits that occurred between each pair of consecutive releases [22, 14]. We manually inspect the results from RefFinder to filter out false positive refactoring edits. We also use an automated change impact analysis tool, FaultTracer [33]. It takes the old and new versions of a program and a regression test suite as inputs. It then identifies *affected tests*—a subset of tests relevant to the program differences between the two versions and *affecting changes*—a subset of atomic changes relevant to the affected tests. We then examine these two sets of data to investigate how refactoring edits in practice affect regression tests in three large open source projects: *Apache JMeter*, *XML Security Library*, and *Apache ANT*. The following summarizes our study questions and corresponding results.

- **Q1:** Are there adequate tests for refactoring edits in practice? In theory, refactorings are code transformations that do not affect any functional property of the program. However, in practice, we often find that refactorings co-occur with functional changes. A sufficient test coverage of refactoring edits may provide confidence to programmers to initiate refactoring effort. In our study, we find that refactoring edits are not very well tested—only 22% of refactored methods and fields are covered by existing regression tests. Though this coverage of refactoring edits is slightly higher than the test coverage of all changes (20%), there is an insufficient amount of regression tests to ensure the correctness of refactoring edits.

- **Q2:** How many of existing regression tests need to be re-run due to refactoring edits? Because refactoring often introduces a large amount of coordinated edits throughout the system, a large proportion of existing tests may too be affected by the edits and thus must be re-run for the new version. We investigate the proportion of existing regression tests affected by refactoring edits. While refactoring edits constitute only 8% of atomic changes identified by FaultTracer, 38% of affected tests are relevant to those refactoring edits. This result indicates that there is a potential opportunity of saving a regression testing cost, if a regression test selection algorithm can isolate pure refactoring edits from behavior-modifying edits and select tests relevant to only behavior-modifying edits.
- **Q3:** What proportion of failed regression tests are relevant to refactoring edits? Manual application of refactoring could be error-prone and logically inconsistent. To understand the extent of regression errors potentially caused by refactoring edits, we measure how many of failed regression tests exercise the location of refactoring edits. Out of 18083 tests in our subject program, 80 tests failed due to regression faults. 39 out of those 80 failed tests include refactoring edits as affecting changes. We also investigate the types of refactorings relevant to these failed tests. Most of these refactorings were intra-method refactoring edits such as *remove control flag*, *inline temp*, *introduce explaining variable*, etc [8]. We speculate that these refactoring edits are often done manually by programmers without an automated tool. This indicates the needs of automated refactoring validation and (/or) test augmentation techniques that target intra-method refactoring edits.

These results call for new regression test augmentation and selection techniques geared to validating refactoring edits. By disambiguating pure refactorings from *refactoring edits mixed*

with behavior-modifying edits, a regression test selection algorithm could avoid selection of test cases that only exercise behavior-preserving edits. The remainder of this paper is organized as follows. Section 2 summarizes related work. Sections 3 and 4 describe our study approach and the corresponding results with the limitations of our study. Section 6 concludes with the direction of future work.

2 RELATED WORK

2.1 Empirical Studies of Refactoring

Xing and Stroulia found that 70% of structural changes in Eclipse's evolution history are due to refactorings and existing IDEs lack support for complex refactorings [32]. Dig et al. studied the role of refactorings in API evolution, and found that 80% of the changes that break client applications are API-level refactorings [5]. While these studies focus on the frequency and types of refactorings, they do not focus on how refactoring edits impact regression testing. MacCormack et al. [17] defined modularity metrics and use these metrics to study evolution of Mozilla and Linux. They found that the redesign of Mozilla resulted in an architecture that was significantly more modular than that of its predecessor. However, this study merely monitors design structure changes in terms of modularity metrics without identifying refactoring edits.

Kataoka et al. [13] proposed a refactoring evaluation method that compares software before and after refactoring in terms of coupling metrics. Kolb et al. [16] performed a case study on the design and implementation of existing software and found that refactoring improves software with respect to maintainability and reusability. Moser et al. [19] conducted a case

study in an industrial, agile environment and found that refactoring enhances quality and reusability related metrics. Carriere et al.'s case study found the average time taken to resolve tickets decreases after re-architecting the system [3]. Ratzinger et al. found that refactoring related features and defects have an inverse correlation [23]. Unlike these previous studies, this paper focuses on the relationship between refactoring edits and regression testing.

Because manual refactoring is often tedious and error-prone, modern IDEs provide features to automate the application of refactorings [10, 25]. However, recent research found several limitations of tool-assisted refactorings as well. Daniel et al. found dozens of bugs in the refactoring tools in popular IDEs [4]. Murphy-Hill et al. found that many refactoring tools do a poor job of communicating errors and programmers do not leverage them as effectively as they could [20]. These findings motivate our study to investigate the relationship between the adequacy of regression tests and refactoring edits.

2.2 Automated Refactoring Reconstruction

Our study approach relies on an automated refactoring reconstruction tool to ease the burden of manually finding refactoring edits from source code or recording all refactoring edits from a refactoring engine in an IDE. Prete et al. present a survey of refactoring reconstruction techniques that take two program versions as input and determines the location and types of refactoring edits [14]. For example, Weißgerber and Diehl use a signature-based analysis and clone detection to analyze and rank refactoring candidates [31]. Xing and Stroulia detect refactoring instances by comparing program versions at the design level, which are packages, classes, interfaces, fields, and blocks [32]. This comparison is based on names and structural similarities. Since both techniques do not analyze method bodies, they do not detect intra-

method refactoring edits, such as an *inline temp* refactoring. Dig and Johnson combine a syntactic analysis and a semantic analysis to detect and refine refactoring candidates [6]. It finds similar code fragments using an *Shingles* analysis and reasons about reference relations between them to aid the matching process.

In our study we use RefFinder because it not only detects simple refactoring edits such as renames and moves, but also complex refactoring edits that require an analysis of method bodies or pre-requisite refactoring edits. RefFinder currently supports sixty three refactoring types in the Fowler's catalog [8], showing the most comprehensive coverage among existing techniques [14]. According to Prete et al.'s evaluation using Fowler's refactoring examples, the accuracy of RefFinder ranges about 93% to 98% for precision and 93% to 99% for recall [14]. In our study, we filter out false positive refactoring edits through manual inspection.

2.3 Change Impact Analysis and Regression Test Selection.

Existing regression test selection algorithms take two program versions V_1 and V_2, and a test suite T as input and select $tests \in T$ relevant to the delta between V_1 and V_2. Some algorithms such as DejaVoo [26, 11, 21] construct control flow graphs (CFG) for both versions and simultaneously traverse the two graphs to identify matching CFG nodes, $\{(o_1, n_1), (o_2, n_2), \ldots (o_k, n_k)\}$, whose outgoing edges have different targets. Then the tests that exercised any of $\{o_1, o_2, \ldots o_k\}$ are selected as *affected tests* because the changes to its control flow may lead to different run-time behavior in the new version V_2.

Chianti change impact analysis [24] instead constructs dynamic call graphs, modeling programs at a coarser granularity. It compares the syntax tree of the old and new program

versions and decomposes the edits into atomic changes at a method and field level [24] such as AM for an method addition and CM for method body edits. It reports **affected tests**—a subset of regression tests relevant to edits and **affecting changes**—a subset of changes relevant to the execution of affected tests in the new version. FaultTracer extends Chianti to identify affected tests and relevant affecting changes more accurately [33]. It uses an extended call graph representation to model how individual tests directly read or write to fields. It implements a Tarantula-style fault localization [12] to rank the affecting changes from the change impact analysis. We analyze affected tests and affecting changes from FaultTracer together with the refactoring edits from RefFinder to investigate the relationship between refactoring and regression tests.

2.4 Refactoring Edit Validation.

Schaeffer et al. validate refactoring edits by comparing data and control dependences between two program versions [27]. As opposed to validating refactoring edits, Daniel et al. focus on testing refactoring engines by systematically generating input programs for refactoring transformations [4]. SafeRefactor focuses on validating refactoring edits by leveraging an existing test generation engine and by comparing test results between the old and new program versions [28]. While these projects focus on either validation of refactoring edits or refactoring engines, we study the impact of refactoring edits on regression tests using version history data.

3 STUDY APPROACH

This section presents our study method and subject programs. It provides the background of RefFinder and FaultTracer. We then discuss how we incorporate the refactoring recon-

struction results from RefFinder and the change impact analysis results of FaultTracer.

3.1 Background on RefFinder

To identify refactoring edits, we employ a RefFinder Eclipse plug-in that automatically identifies both simple and complex types of refactoring edits. It extracts logic facts about the structure of both old and new program versions using the Eclipse JDT syntax tree analysis. It identifies program edits that fit the canonical program structure before and after each refactoring type by invoking template logic queries on extracted facts using a Tyruba logic programming engine [29].

Table 1 shows an example of inferring an *introduce explaining variable* using RefFinder. This refactoring simplifies a complicated expression by putting the result of an expression or parts of the expression in a temporary variable with a name that explains the purpose. For example, in the `get` method in `JMeterVariables` class, a return expression in the old version is assigned to variable `val` in the new version. RefFinder uses the syntax tree analysis to extract logic facts from both program versions. A subset of logical change facts corresponding to the source edits is shown in Table 1. Then RefFinder invokes a template logic query corresponding to *introduce explaining variable* to infer the refactoring edits.

3.2 Background on FaultTracer

FaultTracer combines the strength of Chianti-style change impact analysis and spectrum-based fault localization to improve the precision of localizing failure-inducing edits [33]. FaultTracer computes all atomic changes and their dependencies by analyzing the abstract syntax tree of the old and new program versions: CM (change method), AM (add method), DM (delete method), AF (add field), DF (delete field), CFI (change instance field initializer),

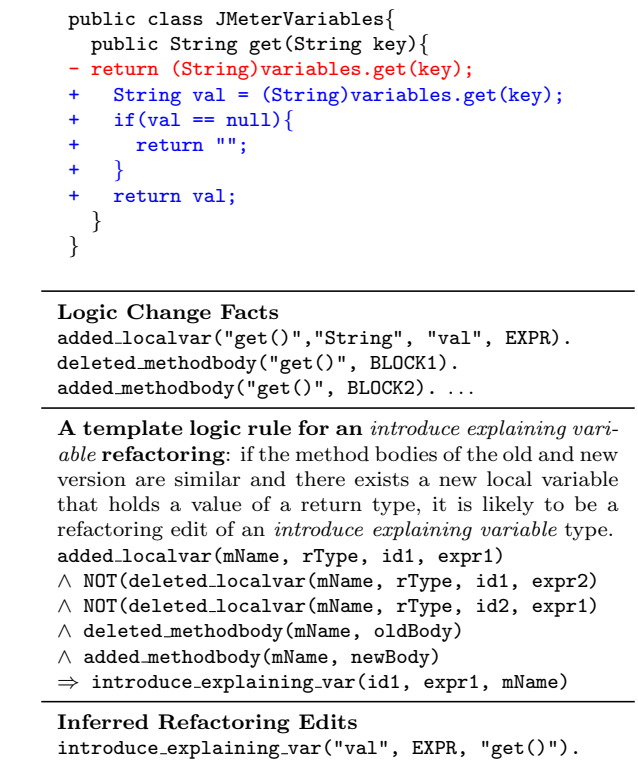

Figure 1: Reconstruction of an *introduce explaining variable* refactoring

CSFI (change static field initializer), LC_m (look up change due to method changes), and LC_f (look up change due to field changes). The dependences among these atomic changes are determined based on the pre-defined rules in [33]. It then creates an extended call graph of each test case using the *ASM byte-code manipulation and analysis framework*. The extended call graph enhances a dynamic call graph representation to additionally include field access information. It then selects *affected tests*, i.e., a set of regression tests in the old version that are relevant to the atomic changes between two program versions and selects *affecting changes*, i.e., a subset of atomic changes relevant to each affected test. We call a set of atomic changes relevant to each failed test as *failure-inducing* changes.

Figure 3 shows an example of a test method that exercises the *introduce explaining*

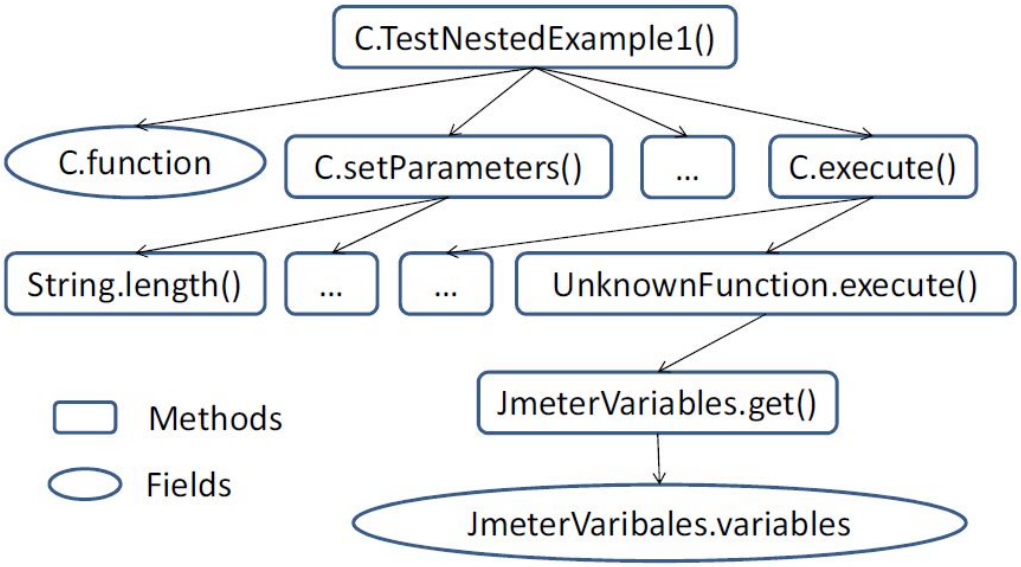

Figure 2: An example extended call graph

variable refactoring edit. Method `testNestedExample1` accesses field `function` of type `CompoundFunction`. Then it calls `setParameters`, which then calls `String.length`. Later `testNestedExample1` calls `execute`, which eventually calls `UnknownFunction.execute`, which then calls `JMeterVariables.get`. Figure 2 shows an extended call graph of `testNestedExample1` that accesses field `variables` which is a part of *introduce explaining variable* refactoring edit from Figure 1. An oval represents a field, and a round rectangle represents a method. The extended call graph represents the trace of method calls and field accesses reachable from a test case under focus. FaultTracer selects this test case as an *affected test* because it exercises the changed method `JMeterVariables.get`. For this test case, the method body change (CM) in `JMeterVariables.get` and all its dependent changes are selected as *affecting changes* as these edits affect the execution of `testNestedExample1`.

```
 CompundFunction.testNestedExample1()

public void testNestedExample1(){
  function.setParameters(expr); ...
  assertEquals("hello worldhellorld",
    function.execute(result, null));...}
```

```
 CompundFunction.setParameters()

public void setParameters(String parameters){
  if (parameters == null || parameters.length() == 0){...}
```

```
 CompundFunction.execute()

public String execute(SampleResult pR, Sampler cS){...
  results.append(item.execute(pR, cS));...}
```

```
 UnknownFunction.execute()

public String execute(SampleResult pR, Sampler cS){
  String ret = getVariables().get(name);...}
```

```
 UnknownFunction.getVariables()

private JMeterVariables getVariables(){
  return varMap.get(Thread.currentThread().getName());
}
```

Figure 3: Test methods that exercise the refactoring edits from Table 1

3.3 Subject Programs

We select three Java programs, *JMeter*, *XMLSecurity*, and *ANT*, from Software Infrastructure Repository (SIR) as our study subjects [7]. Unlike many open source projects, these programs have a large number of regression tests along with evolution history. The size of the programs ranges from 17201 to 80444 LOC (line of code). The number of refactoring edits found by RefFinder in each version ranges from 13 to 222 in *JMeter*, from 19 to 115 in *XMLSecurity*, and from 0 to 249 in *Ant*. The size in LOC and the number of classes, methods and fields are shown in Table 1. It also shows the number of refactoring types, refactoring edits found by RefFinder and validated through our manual inspection, and the number of atomic changes reported by FaultTracer between each consecutive release pair. Table 2 lists the types of refactoring edits, along with the number of instances for each type.

3.4 Manual Inspection Process

Since RefFinder finds false positive refactoring edits—its precision is 0.79 according to Prete et al. [22]—we manually inspect all results from RefFinder to eliminate false positive refactoring edits. We use the following criteria for manual inspection. First, while refactoring is defined as behavior-preserving edits, in practice, refactoring rarely occurs alone without any semantic change [20]. Thus, we include refactoring edits that co-occur with behavior-modifying edits in our data set. We consider the code snippet from *JMeter 4.0* in Figure 4 as an *introduce explaining variable* refactoring edit since `msg` is a new explaining variable for `ex.getMessage()` even though this code also includes a behavior-modifying edit that assigns a different value to `msg` if it is `null`.

Second, while following the guideline and examples from Fowler's refactoring catalog, we do not consider the meaning of identifier names. For an example, in the snippet in Figure 5, `falseCounterString` and `trueCounterString` are introduced as explaining variables for `Integer.toString(vars.getIteration())` and `Integer.toString(counter++)` in method `execute`. Even though the identifier names do not effectively represent their purposes, we consider them as correct refactoring edits.

4 RESULTS

This section describes the result of each study question.

4.1 Are There Adequate Tests for Refactoring Edits in Practice?

If there are sufficient test cases, developers can be more confident about applying refactorings manually as regression errors can be caught by existing tests. On the other hand, if the

```
public void doAction(ActionEvent e){
  ...
  catch(Exception ex){
  + String msg = ex.getMessage();
  + if(msg  == null){
    + msg="Unexpected error - see log for details";
    + log.warn("Unexpected error",ex);
  + }
  - JMeter Utils.reportErrorToUser(ex.getMessage());
  + JMeter Utils.reportErrorToUser(msg);
  }
  ...
}
```

Figure 4: An example refactoring edit of *introduce explaining variable* found in JMeter 3.0-4.0

```
public String execute(SampleResult pR, Sampler cS){
  ...
+ String falseCounterString =
            Integer.toString(counter);
+ String trueCounterString =
            Integer.toString(vars.getIteration());

  if(perThread){
  + return Integer.toString(vars.getIteration());
  + return trueCounterString;
  }else{
  + return Integer.toString(counter++);
  + return falseCounterString;
  }
  ...
}
```

Figure 5: An example refactoring edit of *introduce explaining variable* JMeter 0.0-1.0

regression test suite has insufficient coverage, it may be unsafe to initiate refactoring, because code changes introduced by refactoring may be hard to validate. To investigate whether there are enough regression tests in software projects for developers to safely initiate refactoring, we measure refactoring test coverage and compare this against change test coverage and total test coverage. In Table 3, R is a set of methods and fields that are part of refactoring edits, C is a set of methods and fields that are part of atomic changes identified by FaultTracer, T is a set of methods and fields exercised by existing tests, and A is the total number of

methods and fields in the new version.

Refactoring test coverage is the percentage of refactored methods and fields (R) tested by existing regression tests out of all refactored methods and fields. Change test coverage and total test coverage are defined as $\frac{|(C \cap T)|}{|C|}$ and $\frac{|T|}{|A|}$ respectively. 18083 methods and fields out of 64767 are exercised by regression tests (28% total test coverage). Only 1895 out of 9516 atomic changes are exercised by regression tests, indicating that changes are not well tested by regression tests in the subject programs (20% change test coverage). Similarly, only 160 out of 738 refactored methods and fields are exercised by regression tests, resulting in 22% refactoring test coverage. These results imply that all types edits including refactoring edits are not well tested, and the regression test suite is insufficient for checking the correctness of program changes.

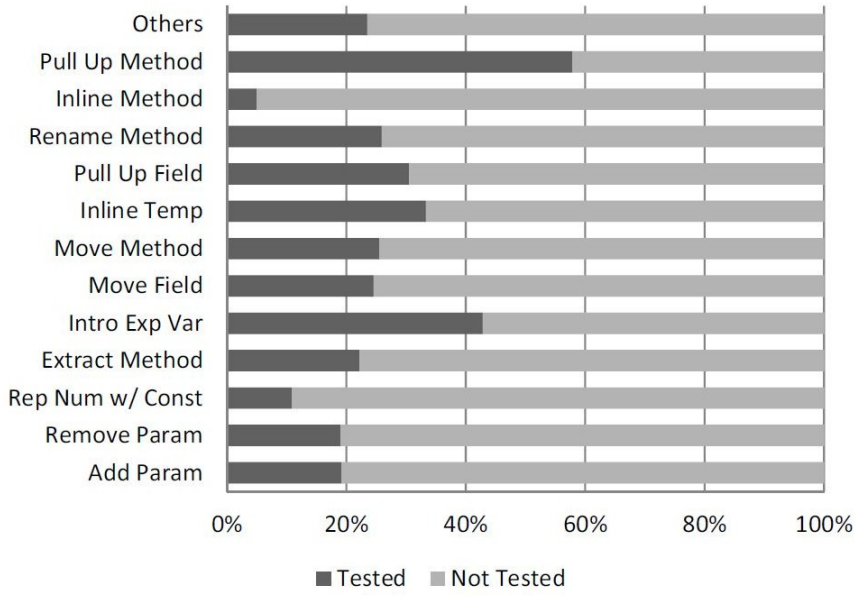

Figure 6: Refactoring test coverage by type

We also measure refactoring test coverage per refactoring type. Two refactoring types with

the lowest test coverage are *replace magic numbers with constant* and *inline method*, both under 10%. The *replace magic numbers with constant* refactoring creates a constant, names it after the meaning, and replaces the number with it for improved readability. The *inline method* refactoring puts the method's body into its caller and removes the method. Since the implementations of these two types of refactoring edits do not often change public APIs, they are relatively easy to implement. Therefore, developers may have not added new tests to validate these types of refactoring.

Only 22% of refactorings are tested

4.2 How Many of Existing Regression Tests are Relevant to Refactoring Edits?

Because refactoring often introduces a large amount of coordinated edits throughout the system, a large proportion of existing tests may too need to be re-run on the new version. We measure a ratio of affected tests that exercise refactoring edits. We use FaultTracer to determine affected tests (AT), each of which exercises at least one atomic change between two program versions. For those affected tests, FaultTracer identifies affecting changes, i.e. the atomic changes that appear on the ECG of the tests and all other changes that are dependent on those atomic changes (AC). We then measure the ratio of affecting tests that exercise at least one refactoring edit location (AT_R), and affecting changes whose location overlaps with at least one refactoring edit (AC_R). AT_R and AC_R both show the ratio of affected tests and affecting changes relevant to refactoring edits.

Table 4 summarizes the results. FaultTracer identifies 1564 regression tests as affected tests, i.e., these regression tests are relevant to the changes between two versions and thus must be re-run on the new version. While the methods and fields that are a part of refactoring

edits constitute only 8% of all atomic changes identified by FaultTracer, 594 out of 1564 affected tests (38% of affected tests) exercise refactoring edits.

> While refactoring edits constitutes only 8% of atomic changes,
> 38% of affected tests are relevant to refactoring edits.

In theory, there is no need of re-running a regression test if the test exercises only pure, behavior-preserving transformations. In the study, there are 12 regression tests that consist of only pure refactoring edits with negligible or no behavioral changes. For example, Figure 7 shows an extended call graph of `testTokenizer` in *Ant 2.0*, which is selected as an affected test. Only `translateCommandline()` was selected as an affecting change, but its code does not have any behavior-modifying edit. Such regression tests represent an added testing cost that should be avoided. There is a potential opportunity of saving regression testing cost, if a test selection algorithm can isolate behavior-modifying edits from behavior-preserving edits and select only the tests exercise behavior-modifying edits.

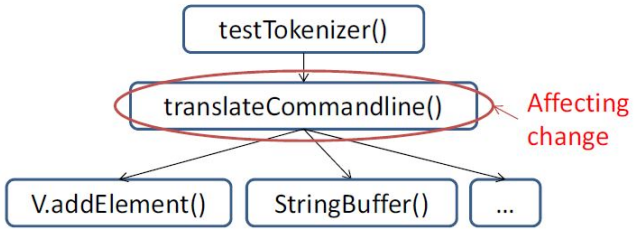

Figure 7: The ECG of `testTokenizer`

4.3 What Proportion of Failure-Inducing Changes Are Relevant to Refactoring?

Manual application of refactoring could be error-prone and logically inconsistent. To understand the extent of regression errors potentially caused by refactoring, we measure how

many of failed regression tests exercise the location of refactoring edits. We define a set of affected tests that succeeded in the old version but failed in the new version as failed affected tests, AT_F. We then measure a subset of AT_F that exercise refactoring edits, which we call AT_{RF}. Similarly, we define a subset of affecting changes for the failed tests as *failure-inducing changes*, AC_F and define a subset of affecting changes for the failed tests that exercise the location of refactoring edits, AC_{RF}.

Table 5 summarizes our results. There are 80 failed affected tests, and 39 out of them exercise the location of at least one refactoring edit. This indicates that refactoring edits in practice often involve functional changes and could possibly introduce errors. In some cases, refactoring edits appear on the execution trace of failed tests but they are not the failure causes. For example, `testNestedExample1` in Figure 3 causes a test failure in the new version. A refactored method `get()` occurs in its extended call graph, and thus `testNestedExample1` is considered as a failed affected test. However, refactoring edits are not necessarily a failure cause. As shown in Figure 8, when we trace back the execution of this test, we find that the fault occurs before the refactored method is called. The fault is in `setParameters()`, which is supposed to add an object of type `CompoundFunction` to the head of the linked list. But it adds an object of type `UnknownFunction` instead and then a wrong `execute()` is called, leading to a failure.

A half of failed affected tests include refactoring edits.

Figure 9 shows the percentage of failure-inducing changes out of all refactoring edits for each refactoring type. The refactoring types with the highest percentage of failure-inducing changes are *remove control flag*, *introduce explaining variable*, and *inline temp*. All these involve with changes in local variables and we believe that developers do not have nor use

automated refactoring engines for these types of refactorings and manual application of these refactorings could be potentially error prone.

```
public void testNestedExample1(){
  function.setParameters(expr);
  ...
  assertTrue(function.hasFunction());
  assertTrue(function.hasStatics());
   assertEquals("hello world", <-- Fault occurs
     function.Components.get(0).execute(result, null));
  assertEquals("hellorld",
    function.cComponents.get(1)).execute(result, null));
   assertEquals("hello worldhellorld",
     function.execute(result, null)); <-- Refactoring occurs
  ...
}
```

Figure 8: `testNestedExample1` failure trace

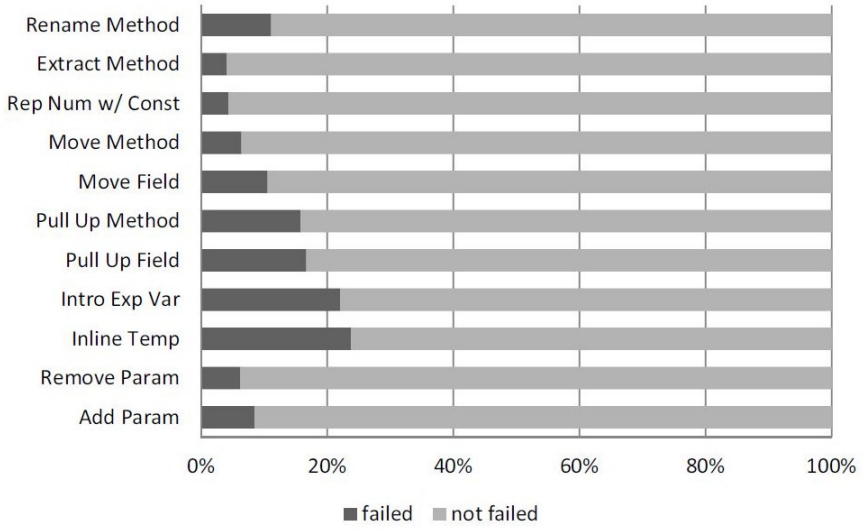

Figure 9: The ratio of failure-inducing changes out of all refactoring edits per type

5 DISCUSSION

While we remove false positive refactoring edits through manual inspection, RefFinder cannot find some refactoring edits (false negatives). In particular, when different refactorings

```
<Refactoring type="Introduce Explaining Variable">
     <package>org.apache.jmeter.threads</package>
     <class>JMeter Variables</class>
     <oldMethod>get</oldMethod>
     <field>val</field>
     <check>CORRECT</check>
     <comment>n/a</comment>
 </Refactoring>
```

Figure 10: An example refactoring edit stored in an XML format

are applied to the same code location, RefFinder generally cannot detect such refactoring instances, because it only compares the old and new versions but does not consider intermediate versions. Though our data set may not include the complete set of refactorings edits in the subject programs, we believe that our study accurately represents the relationship between refactorings and existing tests, because RefFinder has relatively a high recall (0.95) according to Prete et al.'s evaluation [22]).

Some may disagree with our broad definition of refactoring edits—tolerating behavior modifications in counting refactoring edits in the manual inspection process. As we discuss in Section 3, we consult Fowler's catalog of refactoring [8] when manually inspecting refactoring edits. Fowler does not introduce each refactoring type with a formal definition but presents it through examples. Thus, when refactoring edits did not directly conform to Fowler's examples, we subjectively disambiguate individual refactoring instances, as we see appropriate. To mitigate this threat to validity, we make our data set of refactoring edits available in public in an XML format.[1] Each refactoring edit has several attributes: the involved classes, methods, and fields, and the type of refactoring according to Fowler's catalog [8], and our manual inspection result. Figure 10 shows an example of the *introduce explaining variable* instance from an earlier example in the XML format.

[1] `http://users.ece.utexas.edu/~miryung/software.html`

6 CONCLUSION

This paper presents an empirical study of how refactoring affects regression testing using the version histories of Java projects. The study finds that test coverage of refactoring is insufficient, and regression tests are significantly impacted by refactorings edits, though only a small proportion of edits consists of refactoring. Furthermore, failure-inducing changes often include refactoring edits mixed with behavior modifying edits. The results suggest the need of an automated regression test augmentation approach that targets refactoring edits. The results also indicate that there is a potential opportunity of saving regression testing cost, if a test selection algorithm can isolate pure refactoring edits from behavior-modifying edits and select tests relevant to only behavior-modifying edits.

Table 1: Subject Program Statistics

Subject	LoC	#Class	#Method	#Field
JMeter 0.0	31005	313	2501	830
JMeter 1.0	33655	341	2650	894
JMeter 2.0	32948	327	2567	893
JMeter 3.0	37336	383	3103	938
JMeter 4.0	38162	393	3164	935
JMeter 5.0	40695	402	3237	970
XMLSec 0.0	17435	181	1244	140
XMLSec 1.0	18323	194	1300	147
XMLSec 2.0	22863	221	1424	151
XMLSec 3.0	16878	154	1023	129
Ant 0.0	17201	172	1581	748
Ant 1.0	25846	228	2427	1145
Ant 2.0	39733	342	3690	1714
Ant 3.0	38810	342	3696	440
Ant 4.0	61877	532	5430	2520
Ant 5.0	63510	536	5546	2546
Ant 6.0	63578	536	5808	2550
Ant 7.0	80381	649	7186	3212
Ant 8.0	80444	650	7190	3212

Pair	#Type	#Correct	#Change
JMeter 0.0-1.0	10	92	58
JMeter 1.0-2.0	12	67	68
JMeter 2.0-3.0	11	222	72
JMeter 3.0-4.0	9	47	51
JMeter 4.0-5.0	4	13	58
XMLSec 0.0-1.0	8	19	92
XMLSec 1.0-2.0	10	115	53
XMLSec 2.0-3.0	6	27	69
Ant 0.0-1.0	12	199	101
Ant 1.0-2.0	14	249	123
Ant 2.0-3.0	3	4	47
Ant 4.0-5.0	8	53	446
Ant 5.0-6.0	0	0	46
Ant 7.0-8.0	3	6	392

Table 2: Refactoring edits identified by RefFinder and validated by manual inspection

Input program pair	Refactoring types (# instances)
JMeter 0.0-1.0	move method(3), replace magic number with constant(6), introduce explaining variable(8), extract interface(1), extract method(5), replace method with method object(4), add parameter(30), move field(8), extract class(1), remove parameter(26)
JMeter 1.0-2.0	replace exception with test(1), move method(1), replace magic number with constant(6), introduce explaining variable(4), pull up field(16), extract method(7), extract superclass(1), rename method(1), inline temp(3), add parameter(8), move field(16), remove parameter(3)
JMeter 2.0-3.0	inline temp(26), move method(16), replace magic number with constant(18), introduce explaining variable(19), extract method(21), inline method(9), introduce assertion(1), rename method(10), add parameter(52), move field(6), remove parameter(44)
JMeter 3.0-4.0	inline temp(1), move method(1), replace magic number with constant(8), introduce explaining variable(2), extract method(1), replace method with method object(5), add parameter(11), extract class(1), remove parameter(17)
JMeter 4.0-5.0	add parameter(2), replace magic number with constant(4), introduce explaining variable(5), remove parameter(2)
XMLSec 0.0-1.0	move field(4), extract method(2), inline temp(3), add parameter(3), inline method(1), remove parameter(2), introduce explaining variable(3), replace exception with test(1)
XMLSec 1.0-2.0	extract method(2), move method(12), pull up method(12), move field(19), pull up field(19), extract superclass(2), inline temp(1), add parameter(18), remove parameter(27), introduce explaining variable(3)
XMLSec 2.0-3.0	move method(3), replace magic number with constant(2), extract method(2), push down method(1), add parameter(8), remove parameter(11)
Ant 0.0-1.0	inline temp(1), pull up field(1), replace magic number with constant(15), introduce explaining variable(13), move method(5), extract method(13), inline method(1), remove parameter(70), add parameter(71), move field(7), extract class(1), pull up method(1)
Ant 1.0-2.0	inline temp(1), move method(12), replace magic number with constant(56), introduce explaining variable(16), extract interface(1), extract method(6), inline method(1), pull up method(11), remove control flag(3), extract superclass(2), pull up field(9), add parameter(66), move field(9), remove parameter(56)
Ant 2.0-3.0	add parameter(2), introduce explaining variable(1), remove parameter(1)
Ant 4.0-5.0	inline temp(10), replace magic number with constant(4), introduce explaining variable(9), extract method(2), remove control flag(2), rename method(3), add parameter(16), remove parameter(7)
Ant 5.0-6.0	n/a
Ant 7.0-8.0	add parameter(1), introduce explaining variable(4), remove parameter(1)

Table 3: Are there adequate tests for refactoring in practice?

Pair	$\|R\|$	$\|C\|$	$\|T\|$	$\|A\|$	$\|R \cap T\|$	$\|C \cap T\|$
JMeter 0.0-1.0	62	880	606	3544	9	104
JMeter 1.0-2.0	50	563	1007	3460	4	157
JMeter 2.0-3.0	196	1905	1368	4041	42	584
JMeter 3.0-4.0	33	462	1395	4099	1	77
JMeter 4.0-5.0	11	230	1400	4207	2	38
XMLsec 0.0-1.0	9	456	591	1447	7	126
XMLsec 1.0-2.0	34	145	588	1575	24	50
XMLsec 2.0-3.0	17	500	540	1152	6	100
Ant 0.0-1.0	125	1544	674	3572	10	123
Ant 1.0-2.0	146	2273	991	5404	30	326
Ant 2.0-3.0	3	49	995	5414	3	25
Ant 4.0-5.0	47	435	2374	8092	20	150
Ant 5.0-6.0	0	24	2368	8358	0	14
Ant 7.0-8.0	5	50	3186	10402	2	21
Total	738	9516	18083	64767	160	1895

Table 4: How many of existing tests need to be rerun due to refactorings?

Pair	$\|AT\|$	$\|AT_R\|$	$\|AC_R\|$	$\|R\|$	$\|C\|$
JMeter 0.0-1.0	55	22	7	62	880
JMeter 1.0-2.0	66	28	3	50	563
JMeter 2.0-3.0	55	53	58	196	1905
JMeter 3.0-4.0	51	5	1	33	462
JMeter 4.0-5.0	57	12	1	11	230
XMLsec 0.0-1.0	68	46	7	9	456
XMLsec 1.0-2.0	53	53	24	34	145
XMLsec 2.0-3.0	59	34	4	17	500
Ant 0.0-1.0	101	79	7	125	1544
Ant 1.0-2.0	123	115	26	146	2273
Ant 2.0-3.0	47	11	3	3	49
Ant 4.0-5.0	429	126	44	57	435
Ant 5.0-6.0	34	0	0	0	24
Ant 7.0-8.0	366	10	5	5	50
Total	1564	594	190	738	9516

Table 5: What proportion of failure-inducing changes are relevant to refactoring?

Pair	$\|R\|$	$\|C\|$	$\|AT_F\|$	$\|AT_{RF}\|$	$\|AC_F\|$	$\|AC_{RF}\|$
JMeter 0.0-1.0	62	880	16	11	41	2
JMeter 1.0-2.0	50	563	3	3	2	1
JMeter 2.0-3.0	196	1905	0	0	0	0
JMeter 3.0-4.0	33	462	0	0	0	0
JMeter 4.0-5.0	11	230	0	0	0	0
XMLsec 0.0-1.0	9	456	0	0	0	0
XMLsec 1.0-2.0	34	145	5	5	12	7
XMLsec 2.0-3.0	17	500	0	0	0	0
Ant 0.0-1.0	125	1544	6	5	21	2
Ant 1.0-2.0	146	2273	4	3	63	5
Ant 2.0-3.0	3	49	14	4	12	1
Ant 4.0-5.0	57	435	17	7	458	44
Ant 5.0-6.0	0	24	0	0	0	0
Ant 7.0-8.0	5	50	15	1	53	5
Total	738	9516	80	39	662	67

References

[1] K. Beck. *extreme Programming explained, embrace change.* Addison-Wesley Professional, 2000.

[2] L. A. Belady and M. Lehman. A Model of Large Program Development. *IBM Systems Journal*, 15(3):225–252, 1976.

[3] J. Carriere, R. Kazman, and I. Ozkaya. A cost-benefit framework for making architectural decisions in a business context. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 149–157, New York, NY, USA, 2010. ACM.

[4] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 185–194, New York, NY, USA, 2007. ACM.

[5] D. Dig and R. Johnson. The role of refactorings in api evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[6] D. Dig and R. Johnson. Automated detection of refactorings in evolving components. In *ECOOP '06: Proceedings of European Conference on Object-Oriented Programming*, pages 404–428. Springer, 2006.

[7] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, 2005.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 2000.

[9] C. Görg and P. Weißgerber. Error detection by refactoring reconstruction. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[10] W. G. Griswold. *Program Restructuring as an Aid to Software Maintenance.* PhD thesis, University of Washington, 1991.

[11] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 312–326, New York, NY, USA, 2001. ACM.

[12] M. J. H. James A. Jones and J. Stasko. Visualization of test information to assist fault localization. In *ICSE '05: Proceeding of the International Conference on Software Engineering*, page 477, Orlando, Florida, USA, 2002. ACM Press.

[13] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 576 – 585, 2002.

[14] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 371–372, New York, NY, USA, 2010. ACM.

[15] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring benefits and challenges. In *Submitted to FSE 2012 (Under Review)*, 2012.

[16] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. Refactoring a legacy component for reuse in a software product line: a case study: Practice articles. *J. Softw. Maint. Evol.*, 18:109–132, March 2006.

[17] A. MacCormack, J. Rusnak, and C. Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. volume 52, pages 1015–1030, 2006.

[18] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[19] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi. Does refactoring improve reusability? In *ICSR*, pages 287–297, 2006.

[20] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 287–297, Washington, DC, USA, 2009. IEEE Computer Society.

[21] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth International Symposium on Foundations of Software Engineering*, pages 241–251, New York, NY, USA, 2004. ACM.

[22] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1 –10, Sept. 2010.

[23] J. Ratzinger, T. Sigmund, and H. C. Gall. On the relation of refactorings and software defect prediction. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 35–38, New York, NY, USA, 2008. ACM.

[24] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 432–448, New York, NY, USA, 2004. ACM.

[25] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[26] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, 1997.

[27] M. Schaefer and O. de Moor. Specifying and implementing refactorings. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 286–301, New York, NY, USA, 2010. ACM.

[28] G. Soares, R. Gheyi, D. Serey, and T. Massoni. Making program refactoring safer. *Software, IEEE*, 27(4):52 –57, july-aug. 2010.

[29] K. D. Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 1998.

[30] P. Weißgerber and S. Diehl. Are refactorings less error-prone than other changes? In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 112–118, New York, NY, USA, 2006. ACM.

[31] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.

[32] Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 54–65, New York, NY, USA, 2005. ACM.

[33] L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *ICSM*, pages 23–32. IEEE, 2011.