Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Early Detection of Manual Refactoring Faults

Everton L. G. Alves

Tese submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Engenharia de Software

Dra. Patrícia D. L. Machado

Dr. Tiago Massoni

(Orientadores)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

A474E Alves, Everton Leandro Galdino.
Early detection of manual refactoring faults / Everton Leandro Galdino Alves. – Campina Grande, 2015.
125 f. : il. color.

Tese (Doutorado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.

"Orientação: Prof.ª Dr.ª Patrícia D. L. Machado, Prof. Dr. Tiago Massoni".
Referências.

1. Refactoring. 2. Testing. 3. Prioritization. I. Machado, Patrícia D. L. II. Massoni, Tiago. III. Título.

CDU 004.415.53(043)

"EARLY DETECTION OF MANUAL REFACTORING FAULTS"

EVERTON LEANDRO GALDINO ALVES

TESE APROVADA EM 07/04/2015

[illegible]

PATRICIA DUARTE DE LIMA MACHADO, Ph.D, UFCG
Orientador(a)

Tiago L. Massoni

TIAGO LIMA MASSONI, Dr., UFCG
Orientador(a)

Roberta Coelho

ROBERTA DE SOUZA COELHO, Drª, UFRN
Examinador(a)

PAULO HENRIQUE MONTEIRO BORBA, Ph.D, UFPE
Examinador(a)

[illegible]

FRANKLIN DE SOUZA RAMALHO, Dr., UFCG
Examinador(a)

Rohit gheyi

ROHIT GHEYI, Dr., UFCG
Examinador(a)

CAMPINA GRANDE - PB

Universidade Federal
de Campina Grande

Declaro, para os devidos fins, que participei por videoconferência da apresentação da defesa da Tese de Doutorado de **Everton Leandro Galdino Alves**, entitulada: "EARLY DETECTION OF MANUAL REFACTORING FAULTS", em 07 de Abril de 2015 e considero o trabalho aprovado.

[illegible]

Paulo Henrique Monteiro Borba (UFPE)

Resumo

Um estudo recente mostra que cerca de 90% de todos os refactoramentos são aplicados manualmente. Refatoramentos manuais são mais suscetíveis a erro, uma vez que desenvolvedores tem que coordenar transformações relacionadas e entender relações, muitas vezes complexas, entre arquivos, variáveis e métodos. Neste contexto, suites de regressão são usadas para diminuir as chances de introdução de defeitos durante refatoramentos. Contudo, devido aos altos custos de lidar com suites massivas, existe a necessidade de otimização da execução destas. Técnicas de priorização de casos de teste propõem uma nova ordem de execução, almejando a detecção antecipada de faltas. Entretanto, as técnicas atuais não são projetadas para lidar especificamente com faltas relacionadas a refatoramentos. Neste documento propomos RBA (Refactoring-Based Approach), uma técnica de prioritização voltada para refatoramentos. RBA reordena uma suite existente de acordo com um conjunto de modelos de falta (Refactoring Fault Models - RFMs). Estes abrangem os elementos de código que são geralmente impactados dado um refatoramento. Apesar de ser a técnica de validação de refatoramentos mais usada na prática, em alguns casos, o uso de suites de regressão pode ser inadequado. Suites inadequadas podem impedir desenvolvedores de iniciar uma tarefa de refatoramento dada as chances de introdução de defeitos. A fim de complementar a validação por testes e ajudar na revisão de refatoramentos, nós propomos REFDISTILLER, uma técnica que usa anáise estática para detectar edições de código negligenciadas e edições extra que desviam de um refatoramento padrão e podem vir a mudar o comportamento do software. Ambas abordagens (RBA e REFDISTILLER) focam em sistemas Java/JUnit e em um sub conjunto dos refatoramentos mais comuns. Uma avaliação usando um dataset composto de faltas de refatoramento sutis, e comparando com técnicas de prioritização tradicionais, mostra que RBA melhor prioriza as suites em 71% dos casos, promovendo um melhor agrupamento dos casos de teste em 73% dos casos. REFDISTILLER detecta 97% das faltas do nosso dataset de faltas injetadas. Destas, 24% não são detectadas por suites de teste geradas. Finalmente, em um estudo com projetos open-source, REFDISTILLER detecta 22.1 mais anomalias que as suites de teste, com uma precisão de 94%. Esses resultados mostram que (i) RBA consegue melhorar consideravelmente a priorização durante evoluções perfectivas, melhorando tanto a antecipação da detecção de defeitos, quanto fornecendo mais informação sobre estes antecipadamente; (ii) RefDistiller complementa efetivamente a análise dinâmica por achar novas anomalias e fornecer informações extra que ajudam no debug e correção das faltas.

Abstract

A recent study states that about 90% of all refactorings are done manually. Manual refactoring edits are error prone, as refactoring requires developers to coordinate related transformations and to understand the complex inter-relationship between affected files, variables, and methods. In this context, regression tests suites are often used as safety net for decreasing the chances of introducing behavior changes while refactoring. However, due to the high costs related to handling massive test suites, there is a need for optimizing testing execution. Test case prioritization techniques propose new test execution orders fostering early fault detection. However, existing general-purpose prioritization techniques are not specifically designed for detecting refactoring-related faults. In this work we propose a refactoring-aware strategy – RBA (Refactoring-Based Approach) – for prioritizing regression test case execution. RBA reorders an existing test suite, according to a set of proposed Refactoring Fault Models (RFMs), which comprise impact locations of certain refactorings. Although being the most used refactoring validation strategy in practice, regression suites might be inadequate. Inadequate test suites may prevent developers from initiating or performing refactorings due to the risk of introducing bugs. To complement testing validation and help developers to review refactorings, we propose REFDISTILLER, a static analysis approach for detecting missing and extra edits that deviate from a standard refactoring and thus may affect a program's behavior. Both strategies (RBA and RefDistiller) focus on Java/JUnit systems and on a set of the most common refactoring types. Our evaluation using a data set composed by hard-to-identify refactoring faults shows that RBA improves the position of the first fault-revealing test case in 71% of the suites, also providing a better grouping rate (in 73% of the cases) for test cases in the prioritized sequence, when compared to well-known general purpose techniques. Regarding REFDISTILLER, it detects 97% of all faults from our data set with seeded refactoring faults, of which 24% are not detected by generated test suites. Moreover, in a study with open source projects, REFDISTILLER detects 22.1 times more anomalies than testing, with 94% precision on average. Those results show that (i) RBA can considerably improve prioritization during perfective evolution, both by anticipating fault detection as well as by helping to giving more information about the defects earlier; and (ii)

REFDISTILLER effectively complements dynamic analysis by finding additional anomalies, while providing extra information that help fault debugging/fixing.

Acknowledgments

Here I express my sincere gratitude to everyone that helped me throughout the long journey of this doctorate research.

To God, truthful friend, that is always with me through the good and bad moments. Without Him I would never have the strength to overcome challenges or achieve my goals.

To my family for their love and support. My parents, Edvaldo and Eliene that are my life mentors. Since childhood they have taught me the importance of education and, with life lessons, have showed me how to be a decent human being and good citizen. My brother Emanuel and grandparents Manuel, Maria (*in memoriam*) and Creuza, for their support and care. My girlfriend Joicy for all her partnership, love and patience.

To all my friends, for showing that life cannot be complete without good people to share moments. Here I include all friends that I made during my experience abroad, the Halstead family. It is nice to know that distance is not an obstacle for real friendship.

To professors Patrícia Machado and Tiago Massoni, that provided me the foundation for an academic career. Their guidance, trust, supervision and directions have greatly helped me in all the time of research and writing of this thesis. Their patience and hard work have inspired me to be a better researcher. I could not have imagined having better mentors for my doctorate.

To professor Miryung Kim for her mentoring during my eleven-month internship at The University of Texas. Her lessons and hard work will always inspire me to push myself in order to achieve greatness. Thank you for allowing to have a life changing experience.

To my colleagues Ana Emília, Adriana, Julio, Samuel, Igor, Joao Filipe, Alan, Neto, the other members of the Software Practice Laboratory (SPLab), and the members of the Software Evolution and Analysis Laboratory (SEAL), for their support, collaboration and company during this research.

To the funding agencies Conselho Nacional de Desenvolvimento Científico e Tecnologico (CNPq) and Instituto Nacional de Ciência e Tecnologia para Engenharia de Software (INES) for their financial support throughout this work.

Last but not least, I thank the Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande and its staff for the administrative support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Refactoring edits are code changes performed for improving quality factors of a program while preserving its external behavior [33; 75]. The term *Refactoring* was first coined by Opdyke [86] and later popularized by Fowler [33]. Refactoring edits play an important role during software development. Recent studies have evidenced that nearly 30% of the changes performed during a software development are likely to be refactoring [108]. For instance, in the Extract Method refactoring [33], which is one of the most widely applied refactorings [79], code clones spread throughout several methods of a class can be unified into a single method, then replacing the clones by a call to this new method. In the agile community, the refactoring activity is known to contribute to confine the complexity of a source code and to improve nonfunctional aspects of a software such as decrease coupling and increase cohesion [78]. Fowler [33] lists four advantages that refactoring brings in the context of agile projects: i) it helps developers program faster; ii) it improves design of the software; iii) it makes software easier to understand; and iv) it helps developers find bugs.

Although popular IDEs (e.g., Eclipse[1], NetBeans[2]) have built-in automatic refactoring tools, developers still perform most refactorings manually. Murphy et al. [79] find that about 90% of all refactoring edits are manually applied. Negara et al. [83] agree by showing that expert developers prefer manual refactorings over automated. Usability issues seem to have a negative impact on developers' confidence on those tools [67]. Moreover, recent studies show that incorrect refactorings - unexpectedly changing behavior - are present even in the

[1]https://eclipse.org/

[2]https://netbeans.org/

most used tools [26; 109; 77]. Therefore, manual refactoring remains developers' standard procedure for performing refactoring tasks [63].

The manual application of refactoring edits often leads to the error-proneness of refactoring, despite the original intention of improving software quality and productivity through refactoring. Studies using version histories find that indeed there is a strong correlation between the timing and location of refactorings, and bug fixes [61]. Weibgerber and Diehl find that a high ratio of refactoring is often followed by an increasing ratio of bug reports [119; 61]. In a field study of refactoring at Microsoft, Kim et al. [63] observe that 77% of the survey participants perceive that refactoring comes with a risk of introducing subtle bugs and functionality regression. Moreover, developers find it difficult to ensure correctness of manual refactoring - *"I would like code understanding and visualization tools to help me make sure that my manual refactorings are valid."* (a quote from a professional developer [63]).

The error-proneness of manual refactoring is even more crucial for object-oriented languages where small changes can have major and nonlocal effects due to type hierarchies and dynamic dispatch [102]. Binder [16] lists a set of possible causes for a refactoring fault inclusion: i) a coding mistake inserted by a programer who is responsible for implementing the refactoring; ii) an unexpected interaction between the modified elements; and iii) a side effect introduced by an incorrect communication of elements of the system.

Several approaches have been proposed for ensuring correctness of refactoring edits:

- *Refactoring mechanics.* Fowler [33] proposes a set of mechanics for a group of refactoring types. Those mechanics are micro steps in which small edits are combined with compilation and test checking. However, the overhead of applying each step individually may lead developers to apply several at once, which can easily lead to mistakes;

- *Formal specification of refactorings.* Several studies try to formally specifying refactoring edits and, consequently, eliminate the possibility of fault introduction (e.g., [25; 76; 105; 106]). However, due to the intrinsic complexity of object-oriented languages, these formal solutions have a short applicability in real projects;

- *Refactoring validation tools and methodologies.* Several studies propose strategies for validating refactoring edits. For instance, SafeRefactor [111] validates refactoring edits by leveraging an existing test generation engine and by comparing test results between two versions of a program. It requires the generation of new tests cases for the program under refactoring. Similarly, GhostFactor [39] validates whether certain behavior preserving conditions are satisfied for refactoring edits; however, it is limited to three refactoring types and five types of refactoring faults.

Regression test suites are still the main, in most of the cases the only, strategy for assuring the correctness of manual refactorings in the daily development. A regression test suite [68] is a set of test cases which have already passed when run against previous versions of a *System Under Testing* (SUT) and should still pass when run against its future versions, i.e., given a program P and P' (P after a modification), a regression suite should reveal behavior differences between P and P'. Testing validation has been widely used in the industry context [85]. Moreover, test suites are the main validator artifact used during the continuous architecture adjustments in agile methodologies [78]. Developers often use regression test suites as safety nets to gain confidence that, if all test cases continue to pass after the edits, the software's previous behavior remains preserved. Participants from Kim et al. field study at Microsoft [63] pointed that the lacking of a sufficient regression test suite often prevent developers to initiate a refactoring effort. However, it may be impractical to rerun the whole test suite, and analyze its execution, every time a refactoring edit is performed. This difficulty is crucial for projects that deal with massive test suites and have limited resources. For instance, in the context of our lab[3], there is a Java project which applies several real time features. The execution of this system's JUnit test suite takes over 48 hours. Thus, suppose a recently added refactoring fault is detected by a test case run only in the 48º hour, it could easily pass undetected, or be later detected, if a developer decides not to run the whole test suite due to a time constraint. Techniques that minimize the number of test cases to run and analyze while maintaining their effectiveness are desirable.

Test case prioritization [100] rearranges a test suite aiming to speed up the achievement of certain testing goals (e.g., improve the rate of fault detection). Differing from others strategies designed for reducing regression efforts (e.g., test case selection, test case reduction), prioritization techniques do not discard any test case. The developer/tester decides how much of the prioritized suite needs to be run, according to his needs and/or available resources. However, the top test cases are most likely to enable the achievement of the

[3]http://splab.computacao.ufcg.edu.br/

testing goal. Although prioritization has been widely studied (e.g., [122; 64; 69; 118; 112; 101; 95]), there is no prioritization techniques focusing on detecting refactoring faults. Most prioritization techniques use coverage criteria to accelerate detection of faults. However, testing coverage alone frequently misleads the prioritization when searching for test cases that reveal refactoring problems [5; 4]. Moreover, refactoring edits often lead to the inclusion of faults [61; 119; 63]. Therefore, a prioritization approach for detecting those faults might require more qualified data, and a specific analysis. On the other hand, deep impact analysis approaches may be costly and time-consuming. Finally, refactoring validation by testing alone might be faulty. The effectiveness of a test suite is related to how much effort developers put into creating test cases.

1.1 Problem

Suppose a project with a massive regression test suite. By massive we mean a test suite that could rarely be run entirely with limited resources (*e.g.*, too time consuming or too costly). This suite was created by the project's developers over several software development iterations and not for validating any specific change. Now, suppose a developer decides to manually perform a single refactoring edit in the project's code, and, without noticing, he ends up changing the behavior of the system due to a side effect related to the recent applied edit (refactoring fault). Depending on the developer's expertise and/or system's characteristics, this refactoring fault can be hard to detect/fix, or may even be not detected till later stages. Undetected, or later detected, refactoring faults often entail delayed software deliveries and high costs. Thus, solutions to help developers to detect refactoring faults earlier (before it passes to later phases of the software development), additionally providing information that can be used as starting point to fix those faults, are highly desired by developers [63], specially when refactoring is manually applied.

The following two examples use snippets from open-source projects to instantiate the problem above introduced. Suppose Ann is a developer in the JMock project[4], a library that supports test-driven development of Java code with mock objects ($\approx$ 5KLOC, 445 JUnit test cases). Ann decides to perform a Pull up Field refactoring edit, aiming at reducing

[4]http://www.jmock.org/

repetitive code. Field `myActualItems` should be moved from `ExpectationSet` and `ExpectationList` to `AbstractExpectationCollection` (Figure 1.1). Not familiar with any refactoring tool, she performs this refactoring manually. Trying to be systematic, Ann decides to follow Fowler's pull up field mechanics [33], which comprises this edit into six steps:

1. Inspect all uses of the candidate fields to ensure they are used in the same way;
2. If the fields do not have the same name, rename the fields so that they have the name you want to use for the super class field;
3. Compile and test;
4. Create a new field in the super class;
5. Delete the subclass fields;
6. Compile and test.

Supposing that Ann is having a busy day and, by mistake, she ends up introducing a refactoring fault after neglecting the first step of the pull up field's mechanics. Instead of moving identical fields, Ann moves fields with identical names (`myActualItems`) but with different defining types (`HashSet` and `ArrayList`). Figure 1.1 depicts Ann's changes - code insertion is marked with '$+$', code deletion with '$-$'). As the program uses only common methods from Java Collections API, no compilation error is found, however, a behavior change is introduced. For instance, a method that used to know `myActualItems` as an `HashSet` object, is expecting no repetitive elements in this list, and by calling `myActualItems.remove(Object o)` it expects a specific object to be removed. This behavior might have changed after the refactoring, since method `remove(Object o)` from `ArrayList` removes *the first occurrence* of the specified element from this list.

Aiming at validating her edits, Ann runs existing JMock's test suite. In this running, Ann observes that only 4 out of 445 test cases fail due this fault. The first test case to fail is run after 400 others. Suppose an average execution time of 30 seconds for each test case (e.g., test cases that access database are often costly and time consuming), it would take over 3 hours and a half to detect this problem. Depending on how much time and resources

```java
class AbstractExpectationCollection {
  ...
}
class ExpectationSet extends
    AbstractExpectationCollection {
  HashSet myActualItems = new HashSet();
  ...
}
class ExpectationList extends
    AbstractExpectationCollection {
  ArrayList myActualItems = new ArrayList
    ();
  ...
}
```

(a) Original code.

```java
class AbstractExpectationCollection {
+  HashSet myActualItems = new HashSet();
  ...
}
class ExpectationSet extends
    AbstractExpectationCollection {
-  HashSet myActualItems = new HashSet();
   ...
}
class ExpectationList extends
    AbstractExpectationCollection {
-  ArrayList myActualItems = new ArrayList
    ();
   ...
}
```

(b) Code after Ann's pull up field refactoring. Since the moved field had different types in the subclasses, a behavioral change is introduced after the edit. Because there is no compilation error, Ann did not notice it.

Figure 1.1: An example of a problematic refactoring edit (refactoring + fault) using JMock's code.

are available, Ann could naively stop the testing validation before any fault-detecting test case is run (e.g., after running 50% of the test cases), and have the sense that no fault was introduced, which in fact is a misleading impression.

Trying to accelerate the process of detecting this fault, and possibly help the fault debugging process, Ann priorizatizes JMock's regression suite by using the traditional techniques [99] (Total Statement Coverage, Total Method Coverage, Additional Statement Coverage, Additional Method Coverage and Random Choice). However, no optimal results are found from this process. Table 1.1 shows the position of the first test case that reveals the fault (*F-Measure*) for all prioritized suites. Although all reordered suites detect the fault earlier, the strategy that produced the best result was *Random Choice*. This fact shows us evidence that the general prioritization heuristics may not effective when dealing with refactoring faults. This fact was later confirmed by a series of empirical studies [5; 4]. By investigating the four failed test cases (e.g., Figure 1.2), we notice that they do not directly exercise parts of the code that were modified during refactoring. The fault is revealed by test cases that cover elements indirectly impacted by Ann's changes. By neglecting aspects such as the impact of a change, the coverage-based techniques fail to perform a better prioritization. Moreover, the traditional prioritization techniques often end up spreading the fault-revealing test cases throughout the entire suite, which makes the debugging process harder.

Table 1.1: Position of the first failed test case after prioritization.

Prioritization Technique	**F-Measure**
Original Suite	400
Total Statement Coverage	206
Total Method Coverage	259
Additional Statement Coverage	169
Additional Method Coverage	112
Random Choice	70

```
public void testManyFromIterator {
     Vector expectedItems = new Vector();
     expectedItems.addElement("A");
     expectedItems.addElement("B");
     Vector actualItems = (Vector)expectedItems.clone();
     myExpectation.addExpectedMany(expectedItems.
    iterator());
     myExpectation.addActualMany(actualItems.iterator())
    ;
     myExpectation.verify();
}
```

Figure 1.2: Failed test case due to the problematic pull up field edit.

Although widely used in practice, not always a regression test suite is effective. Rachatasumrit and Kim [94] find that regression test suites may lack coverage of refactored locations. Thus, refactoring testing validation by itself is not always safe. Moreover, developers often have a hard time interpreting testing outcomes (pass/fail) and locating/fixing a fault using only tests, specially if the informative test cases are spread out through a test suite.

Suppose another scenario on which Bob works in the XMLSecurity[5] project, which is a library that provides security for managing XML. Bob performs an Extract Method refac-

[5]http://xml.apache.org/security

```
 1   class Reference {
 2     boolean verify() throws .. {
 3       Element digestValueElem = (Element)
          new Node(0);
 4       digestValueElem = this.
          getChildElementLocalName(..);
 5       byte[] p1 = Base64.decode(
          digestValueElem);
 6       byte[] p2 = this.calculateDigest();
 7       boolean re = MessageDigestAlgorithm.
          isEqual(p1, p2);
 8       if (!re) { .. }
 9       return re;
10     }
11   }
```

(a) Original code.

```
 1   class Reference {
 2     boolean verify() throws .. {
 3       Element digestValueElem = (Element)
          new Node(0);
 4 -     digestValueElem = this.
          getChildElementLocalName(..);
 5 +     initializeDigest(); // This line
          should be digestValueElem =
          initializeDigest();
 6       byte[] p1 = Base64.decode(
          digestValueElem);
 7       byte[] p2 = this.calculateDigest();
 8       boolean re = MessageDigestAlgorithm.
          isEqual(p1, p2);
 9       if (!re) { .. }
10       return re;
11     }
12 +   Element initializeDigest() {
13 +     Element digestValueElem;
14 +     digestValueElem = this.
          getChildElementLocalName(..);
15 +     return digestValueElem;
16 +   }
17   }
```

(b) Extract method refactoring with missing edits.

Figure 1.3: An example of problematic refactoring edits. (a) The original code. (b) Code after Bob's extract method refactoring. Lines 4 is extracted to create a new method `initializeDigest`.

toring manually (Figure 1.3). Bob extracts Line 4 to a new method `initializeDigest` and adds a call at line 5. However, since the extracted statement modifies the status of an object used by subsequent statements from `verify`, variable `digestValueElem` should have been updated with the return value of the new method. Because there is no compilation error, Bob misses the required edit. Moreover, the XMLSecurity's test suite in this case is inefficient, not revealing this fault. Bob needs to review his refactoring, but unless he uses a different approach, he will end up with the false sense of a correct program. For instance, an static analysis, or an automatic tool, could have been applied to check whether Bob's edits include all required variable updates for a successful refactoring.

In this work we intend to minimize problems like to the ones above mentioned. For that, we propose solutions for early detection of refactoring faults. By early detection we mean trying to anticipate behavioral changes accidentally added during a refactoring edit, in order to avoid they go unnoticed after the refactoring changes are consolidated into the program code. Differing from other studies, our work considers a scenario in which a developer has just applied a single refactoring edit manually and wants to gain confidence that it was safe. Other works try to prevent problematic refactorings by checking sets of preconditions. However, in practice developers still perform most of their refactorings manually and validate them by using regression testing [79; 83; 63]. Moreover, even well-known refactoring tools that check hundreds of pre-conditions before allowing refactorings are not 100% safe [26; 109; 77]. Thus, our work contributes to state-of-the-art by providing novel solutions that will help developers to accelerate the detection of potential problems that might affect a program's behavior and indications to help developers to better find/fix those faults.

1.2 Research Strategies and Research Questions

Aiming at addressing the problems discussed in Section 1.1, in this work we propose a set of novel approaches, reporting the empirical studies that based the approaches' development. Based on our experience on testing/performing refactorings, and on the literature regarding refactoring validation and/or how developers perform refactorings, we define three research strategies to base our work: i) the investigation on the relation between testing coverage and a simplified impact analysis based on commonly impacted locations for refactoring fault detection. Several code locations are reported as likely to be impacted when refactoring (*e.g.*, callers of a refactored method) and might be a good starting point for evaluating a suite's effectiveness for detecting refactoring faults; ii) the use of test case prioritization for speeding up the detection of refactoring faults. When working with a massive test suite, it might be impractical to rerun the tests after each refactoring. A refactoring-oriented prioritization algorithm may anticipate the detection without reducing a suite's testing power; and iii) the use of static analysis for complementing testing validation and help refactoring review. Static analysis solutions have been widely used for finding general defects in a software code [128]. As testing validation may be flawed, we intend to provide a static analysis refactoring-oriented solution for complementing testing validation and helping developers to review a refactoring that was recently performed. To guide our investigation we define the following research questions:

RQ1: Is the coverage of the most commonly impacted elements appropriate to evaluate a test suite's capability of detecting refactoring faults?

RQ2: Can we anticipate the detection of refactoring faults of a suite by using data such as commonly impacted locations and test coverage?

RQ3: Can static analysis be an effective complement to regression suites for detecting refactoring faults?

1.3 Contributions

Several contributions are presented throughout this work. First, we propose a refactoring-based prioritization approach (RBA) and a tool that automatizes it (PRIORJ). RBA rearranges test cases aiming at speeding up the detection of refactoring faults. For that, it uses refactoring fault models that relate testing coverage data and an impact analysis based on common locations. Moreover, our prioritization groups refactoring-related test cases close to each other, *i.e.*, in nearest positions in the prioritized test suite. That fact may help developers when debugging refactoring faults, since the developer can focus on a smaller group of test cases that are most likely to give him useful information when trying to locate/fix those faults. Our evaluation on a data set with several refactoring faults show that an implementation of RBA outperforms a set of well-known general purpose prioritization techniques by improving by 71% the position of the first fault-revealing test case, also providing a better grouping rate (73% higher) for test cases in the prioritized sequence.

As second contribution, we propose an approach and tool that complements testing validation, REFDISTILLER. This strategy uses static analysis data for detecting two classes of refactoring faults (missing and extra edits). Our approach uses refactoring templates to detect constituent steps that were neglected by a developer when refactoring. Moreover, it uses a refactoring engine to detect extra edits that deviate from standard refactoring and thus may affect a program's behavior. REFDISTILLER is indicated when a developer needs to review a refactoring recently applied. The implemented tool, besides identifying refactoring faults, facilitates fault debugging by giving "clues" about the type and location of the faults.

By using RefDistiller we identified several refactoring faults that are not detected neither manual or automatic test suites. RefDistiller as an approach and tool were developed in collaboration with professor Miryung Kim from The University of Texas at Austin.

RBA and RefDistiller are complementary approaches. The first should be used when there is a trustworthy massive test suite and a developer need to speed up the detection of a refactoring fault. On the other hand RefDistiller can be used when there is little evidences about a suite's quality or the testing results are not helpful enough. RefDistiller complements the testing validation by pointing out code editions that might have pass unnoticed and/or that require confirmation during a refactoring reviewing process.

Finally, as third contribution, we perform an exploratory study on the use of coverage data of mostly impacted code elements to identify shortcomings in a test suite. This study shows that by verifying the coverage rate of the most impacted code locations, and which ones, we can predict a suite's capacity of revealing refactoring faults. The results of this study motivated the definition of both RBA and RefDistiller. Besides that, those results can be used by developers to help deciding whether a testing validation alone can be trusted or not when validating refactorings. Moreover, the conclusions of this study can work as guidelines for test suite augmentation in case there is a need for suite improvement.

We believe that all solutions proposed in this have their potential maximized when applied after a single refactoring edit. Although both solutions (RBA and RefDistiller) are technically applicable with combined refactorings, the impact of this scenario is yet to be assessed. However, we believe that the combination of refactorings would make RBA select a bigger and less useful subset of test cases to start its prioritization, and might lead RefDistiller to generate a bigger number of false positives. For instance, as RBA uses a set of refactoring type and location rules to discover possible impact methods, if two refactorings affect the same location in different ways, this set might not be accurate when the extraction rules are run sequentially, which might interfere with the prioritization process and results.

1.4 Structure

This introduction chapter presents an overview of our doctorate research. Further details regarding the remainder of our research are discussed in the upcoming chapters

This document is organized as follows. Chapter 2 provides background on program refactoring, regression testing and test case prioritization. Next, we describe an exploratory empirical study on the use of coverage of mostly impacted code elements (Chapter 3). Chapter 4 describes an approach for selecting and prioritizing test cases based on refactoring edits. Chapter 5 introduces a static analysis approach for detecting refactoring faults. In Chapter 6 we present the tools developed for automating the proposed solutions. Chapter 7 presents the related work. Finally, Chapter 8 discusses the final remarks about our work and possible future works.

Chapter 2

Background

This chapter describes concepts used in this document to make it self-contained. First, we introduce the basic ideas about *Refactoring* (Section 2.1). Then, we define and present important aspects of *Regression Testing* (Section 2.2). In Section 2.3, we present an overview of *Test Case Prioritization*, including the most used code-based prioritization techniques. Finally, in Section 2.4 we discuss the main ideas about Change Impact Analysis.

2.1 Program Refactoring

Code edits are very common during software development. Those edits are usually classified as: i) *evolutionary*, changes performed to add or remove software functionalities; or ii) *refactoring*, edits for applying structural improvements to a software, not altering any behavioral features.

Opdyke and Johnson [86] first coined the term refactoring and formally define refactorings such as (1) generalizing an inheritance hierarchy, (2) specializing an inheritance hierarchy, and (3) using aggregations to model the relationships among classes. Opdyke defines a refactoring edit as a program transformation aiming at improving a software quality aspect, such as reusability and maintainability, but preserving its semantics. Fowler [33] assembles a catalog with over ninety refactoring types.

In Agile Methods (*e.g.* XP [15], TDD [14]) refactoring is an integral part of the development process; it is adopted to continuously improve the system architecture and source code readability [78]. The agile community widely accepts that refactoring prevents code rotting.

Quite often, refactoring efforts are performed in an *ad hoc* manner. Fowler's mechanics try to reduce the chaos of performing refactoring by decomposing the edits into micro steps associated with compilation and testing checking. Each refactoring type from Fowler's catalog is presented along with an informal description, a motivation for its use, and its mechanics. Although well-accepted, those mechanics do not prevent developers from introducing refactoring faults. It is known that the outcome of a refactoring is often related to the developer's expertise and knowledge about the system [38].

In the past few years, popular Integrated Development Environments (IDEs), such as Eclipse, IntelliJ, NetBeans, and Visual Studio, include support for automated refactoring. Despite the help that refactoring tools may bring, recent studies have shown that developers preform most refactoring edits manually [79; 63]. Other studies speculate on reasons for this underuse, such as usability issues, unawareness, and lack of trust [80; 116]. Moreover, not even the most well-known tools are 100% safe [109].

2.1.1 Refactoring Example

Fowler's refactoring catalog[1] lists 92 different types of refactoring edits. To exemplify how refactoring tasks are usually performed during software development, consider the code in Figure 2.1-a. After analyze this code, a skilled developer may identify duplicate snippets. Variable `k` is identically defined in `C2` and `C3`, both subclasses of `C1`. Fowler and Beck [33] present *Duplication of Code* as one of the 21 *Bad Smells* that are often refactoring opportunities.

After identifying the problematic part of the code, we need to choose the proper refactoring edit to apply. As the code duplication is related to fields in two subclasses of `C1`, the reasonable edit to be performed is a *Pull Up Field*. In a Pull Up Field edit, one or more fields are moved to a superclass aiming at improving a code's readability and maintainability. Fowler's mechanics for this edit is comprised by six steps:

1. Inspect the declaration of the candidate fields to assert that they are initialized in the same way;

[1]http://refactoring.com/catalog/

```java
public class C1 {
     ...
}
public class C2 extends C1 {
     public int k = 10;
     public int getK(){
          return k;
     }
     ...
}
public class C3 extends C1 {
     public int k = 10;
     public int doubleK(){
          return k * k;
     }
     ...
}
```

(a) Original code.

```java
public class C1 {
+     public int k = 10;
     ...
}
public class C2 extends C1 {
-     public int k = 10;
     public int getK(){
          return k;
     }
     ...
}
public class C3 extends C1 {
-     public int k = 10;
     public int doubleK(){
          return k * k;
     }
     ...
}
```

(b) Code after a pull up field edit.

Figure 2.1: Example of a Pull Up Field refactoring.

2. If the fields do not have the same name, rename them so that they have the name you want;

3. Compile and test;

4. Create a new field in the super class. If the fields are private, you should declare them as protected so that the subclass can access it;

5. Remove the fields from the subclasses;

6. Compile and test.

In Figure 2.1-b two Pull Up Field edits are performed. Field `k` is pulled from `C2` and `C3` to `C1`. For this example, no other updates are needed, however, in order to preserve the original semantics, a Pull Up Field edit is often combined with other code updates. For instance, statements that call/use the moved field may have to directly reference the super field to avoiding breaking and/or adding of overriding constraints (e.g., by using the keyword `super`).

2.2 Regression Testing

According to Harrold [47], *Regression Testing* is the activity of retesting a software after it has been modified aiming at gaining confidence that: i) newly added and changed code do not interfere with the previous software behavior; and ii) unmodified code does not misbehave because of the modifications. In other words, regression testing is a practical solution used to give to developers/testers more confidence that new modifications do not alter a previous stable behavior of a software [85].

In a regression scenario there are two versions of the SUT: i) the *base* version - an stable version of the SUT whose test suite reflects the system's behavior; and ii) the *delta* version - SUT after modification(s). A regression test suite is composed by regression test cases. Each regression test case has already passed when run against the base version and it is expected to pass when run against the delta version. When a regression test case does not pass after a modification (delta version) a ”regression fault“ is detected.

When this fault is due to a refactoring edit, we call it a *refactoring fault*. Ideally, a refactoring edit should always be behavior-preserving. When those edits are not performed correctly (e.g., due to a missing step, or extra edit) refactoring-related fault is added, and thus the software's previous stable behavior is modified. A refactoring fault is a type of regression fault. It is highly recommended the combination of refactoring edits and regression testing, and it is indeed commonly applied in practice [94; 109].

The regression activity can be costly and hard to manage. Harrold [47] lists a set of sub activities related to regression testing, each one with its respective challenges: test suite maintenance, regression test selection, test suite augmentation, test suite prioritization, test suite minimization, test case manipulation. Due to their high costs, often a subset of those activities are in fact performed. The decision of which activities to perform is usually made according to the testing team's goals and/or resource limitation.

Several studies (e.g., [68; 85; 17; 99]) have discussed the challenges of using regression testing and its cost. For instance, Chittimalli and Harrold [24] affirm that 80% of testing cost is regression testing. Therefore, new strategies have been developed aiming at reducing the regression testing effort (e.g., [47]), including new techniques for test case selection (e.g., [17]), prioritization (e.g., [69]), and suite reduction (e.g., [65]).

2.3 Test Case Prioritization for Regression Suites

As the software evolves, regression test suite tend be massive [68; 85; 99]. Thus, developers/testers often run part of a test suite, according to theirs budget/resources, but trying not to lose much of the original suite's testing power. As an alternative for decreasing the regression testing high costs and make retesting more effective, *Test Case Prioritization* techniques [29] have been proposed. Those techniques reschedule the test cases in a different execution order for satisfying a given test objective (e.g., increase the rate of fault detection). Thus, a good prioritized suite has in its top positions test cases with the highest chances of achieving certain testing goal.

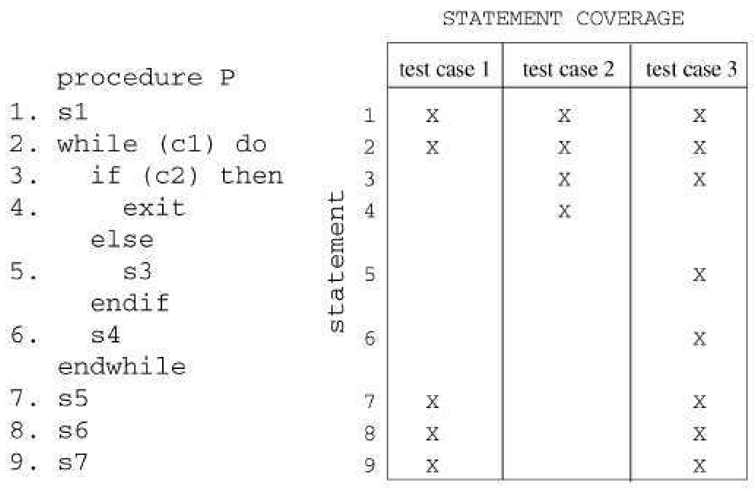
The test case prioritization problem was formally defined by Rothermel [99] as follows: *Given*: T, a test suite; PT, the set of permutations of T, and f, a function from PT to real numbers.
Problem: Find $T' \in PT$ such as $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$
where PT represents the set of possible orderings of T, and f is a function that gets the best possible values when applied to any ordering.

There are several approaches for performing test case prioritization. Singh in [107] categorizes the current prioritization techniques into eight categories: i) *coverage-based* [100; 99; 28]; ii) *modification-based* [122; 64]; iii) *fault-based* [100; 28]; iv) *requirement-based* [113]; v) *history-based* [60; 90]; vi) *genetic-based* [118]; vii) *composite approaches* [71]; and viii) other approaches [101; 95].

In practice, coverage-based prioritization techniques are the most used [107]. This is mainly due to their simplicity and to generate acceptable results in general. Coverage-based prioritization techniques assume the more code elements a test case exercises, the better are the chances of revealing faults.

As follows we present a brief description of the algorithms of the most popular prioritization techniques. The first four are coverage-based techniques, the following is a random technique, and the last one is a modification-based prioritization technique:

- *Total Statement Coverage* (TSC): the prioritized suite's first test case is the one which covers the most SUT statements; the second test case has the second highest statement coverage, and so on;



```
   procedure P
1. s1
2. while (c1) do
3.   if (c2) then
4.     exit
     else
5.     s3
     endif
6.   s4
   endwhile
7. s5
8. s6
9. s7
```

STATEMENT COVERAGE

statement	test case 1	test case 2	test case 3
1	X	X	X
2	X	X	X
3		X	X
4		X	
5			X
6			X
7	X		X
8	X		X
9	X		X

Figure 2.2: (**left-side**) Procedure P; (**right-side**) coverage data.

- *Total Method Coverage* (TMC): the prioritized suite's first test case is the one which covers the most SUT methods; the second test case has the second highest method coverage, and so on;

- *Additional Statement Coverage* (ASC): the prioritized suite's first test case is the one which covers the most SUT statements. The following test case is the one which covers more statements that remain not exercised by the previously chosen test cases. Repeat this process until the prioritized test suite is complete;

- *Additional Method Coverage* (AMC): the prioritized suite's first test case is the one which covers the most SUT methods. The following test case covers more untested methods. Repeat this process until the prioritized test suite is complete;

- *Random* (RD): Technique in which the test cases new order is randomly defined;

- *Changed Blocks* (CB) [114]: first identifies the modified code blocks between two versions of a program (edited parts of the source code), then it reschedules the test cases according to how much of the changed blocks each test case covers.

2.3.1 Prioritization Example

To exemplify how prioritization techniques work, we present the prioritization process of two of the techniques described above, TSC and ASC. Consider procedure P showed in Figure

2.2 (left-hand side). This procedure is exercised by three test cases (Figure 2.2, rigth-hand side). For instance, *Test Case 1* exercise statements *1, 2, 7, 8* and *9*.

According to TSC's prioritization algorithm, the first test case of the prioritized suite is *Test Case 3* because it covers the greatest amount of statements from P (8 statements). The second test case is *Test Case 1* (5 statements). Finally, the prioritized suite is finished with the test case that covers less statements of P, *Test Case 2* (4 statements). Thus, according to the TSC prioritization algorithm, the prioritized test suite is *[Test Case 3, Test Case 1, and Test Case 2]*.

The ASC prioritization process claims that the chances of detecting faults are higher when test cases that cover different parts of the SUT code are run first. Thus, for our example scenario, ASC first selects the test case that covers more statements (*Test Case 3*). The next selected test case is the one that covers more statements that were not covered yet (first *Test Case 2*, and *Test Case 3* in the sequence). Therefore, the final ASC prioritized suite is: *[Test Case 3, Test Case 2, and Test Case 1]*.

2.4 Change Impact Analysis

A software edit often undergoes an impact that goes beyond the entities that are directly modified. Therefore, estimate the set of impacted entities and measure how deep is the impact of a change, might help developers to avoid the high costs of solving problems detected in later development phases, or during software maintenance. Change impact analysis techniques aim at dentifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change [18]. Moreover, this analysis can also be used for predicting the costs of certain change prior its application [49; 50].

Pfleeger and Bohner define impact analysis as the evaluation of the risks associated to certain change, including the measurement of its effects on the project's resources [92]. Besides associate an impact level to a change, impact analysis approaches are often used when there is more than one way of solving an architectural problem. By measuring the impact of each solution to the software's artifacts and resources, developers can better decide which one is the best alternative.

Change impact analysis approaches are often subdivided into two categories: static and dynamic. Static approaches use the program's structure, and/or design elements of a software, such as UML diagrams, to predict the set of impacted entities due to a change (*e.g.*, [96; 3; 21]). Dynamic approaches uses the program's execution data to find relations between entities that might not be captured statically (*e.g.*, [66; 87]. Alternatively, other change impact analysis approaches combine static and dynamic data to provide more accurate results (*e.g.*, [70]).

2.5 Concluding Remarks

In this chapter we cover the basic concepts related to our research. Being aware of those concepts the reader should be capable of fully understand the following chapters of this document. In the next chapter we present an empirical study conducted to investigate the role of testing coverage and commonly impacted locations for detecting refactoring faults. This study's results provide the proper motivation for developing the core solutions proposed in this document.

Chapter 3

Exploratory Empirical Study

In a context where most refactorings are done manually, developers use regression test suites for validating their edit. However, not always test suites are effective for validating refactorings [63]. It seems appropriate to consider that test cases that exercise the elements involved in a refactoring are more likely to uncover faults. For instance, after applying an Extract Method, exercising the changed method, its callers, and callees, tends to be effective in finding any introduced faults. Nevertheless, there is little evidence on which impacted elements are the most important to be tested, and how test coverage is related to this refactoring-based impact analysis. In this chapter, we present an exploratory study [8; 9], performed on three open-source Java projects, that investigate the role of testing coverage and directed impacted locations for detecting refactoring faults, focusing on two of the most common refactoring edits, Extract Method and Move Method [79].

3.1 Motivating Example

Suppose that, after working on several tasks, John, a developer, notices an opportunity of applying an Extract Method edit. Figure 3.1 presents this edit; Lines 5-8 from `Element.m(boolean)` - Figure 3.1-a - are extracted into the `n` method, Figure 3.1-b. As no compilation error is found and the regression test suite passes, John believes his edits are safe. However, in this case the software behavior is undesirably modified. The exception is thrown with the global `x` having its initial state 42; differently, after extracting the method, `x` finishes with value 23 - 42 is only stored to the local `x` before throwing `ex`. This example shows a very subtle refactoring fault that can easily go undetected, specially if the test suite does not complete exercise the `test` method.

Exercising the program elements potentially impacted by a refactoring edit possibly increases the chances to reveal faults. By examining previous research on change impact analysis [97; 126; 102; 77] we can establish that in an Extract Method refactoring edit, the types of elements most likely to be impacted (likely to have its behavior changed due to a bad refactoring) are: i) *the original method* (M): the `n` method is always exercised by calling the `m` method; ii) *the callers of the method under refactoring* (C): methods that call `m()` might be negatively influenced in case they use `m`'s return value, and/or any variable handled by `m`; iii) *the callees of the method under refactoring* (Ce): given methods that `m` calls require as pre-requisite the program to be in a certain state, then `m` must be run according to its previous behavior; and iv) *methods with similar signature to the newly added one* (O): an extracted method may break or introduce overriding/overloading contracts causing a behavior change; for instance, `m` could already be declared within `Element`'s hierarchy.

Although widely adopted to evaluate test suite's quality, *test coverage* alone can mislead developers/testers [52]. More specifically, when dealing with refactoring faults, there is no evidence whether testing coverage is a good quality measure in this context and/or which elements a test suite should be covered to consider it acceptable for validating an edit. Our study aims at analyzing the relationship between impacted elements and test coverage for both the Extract Method and Move Method refactorings, using as experimental units open source Java projects and their JUnit test suites.

3.2 Study on Test Coverage for Impacted Program Elements

We conduct an exploratory study with real open source programs with three goals:
i) investigate whether the refactoring type or the type of fault are factors that influence a suite's capacity of detecting refactoring faults;
ii) investigate whether coverage (method level) of the most commonly impacted elements is appropriate to evaluate a test suite's capability of detecting refactoring faults;

```
 1 class Element{
 2   int m(boolean b){
 3     int x = 42;
 4     try{
 5       if (b){
 6         x = 23;
 7         throw new Exception();
 8       }
 9     }catch(Ex e){
10       return x;
11     }
12     return x;
13   }
14   int test(){
15     return m(true);
16   }
17   ...
18 }
```

(a) Original version.

```
 1   class Element{
 2     int m(boolean b){
 3       int x = 42;
 4       try{
 5 -       if (b){
 6 -         x = 23;
 7 -         throw new Ex();
 8 -       }
 9 +       x = n(b, x);
10       }catch (Ex e){
11         return x;
12       }
13       return x;
14     }
15 +   int n(boolean b, int x) throws
         Exception{
16 +     if (b){
17 +       x = 23;
18 +       throw new Exception();
19 +     }
20 +     return x; }
21     int test(){
22       return m(true);
23     }
24     ...
25   }
```

(b) Code after a problematic Extract Method edit.

Figure 3.1: Extract Method Example. Lines 6-9 are extracted to `n`.

iii) investigate which impacted elements are, when exercised, most likely to reveal refactoring faults.

Our study focuses on two refactoring types, Extract Method (EM) and Move Method (MM) [33]. These are two of the most commonly applied refactorings in Java systems [79]. Those edits are selected to represent edits that involve a single class (Extract Method: part of the code of a method is extracted and placed in a newly added method in the same class) and a pair of classes (Move Method: a method is moved from one class to another). Despite their apparent simplicity, a number of potential issues must be addressed for correctly performing these edits. For instance, the state preservation of variables, fields and parameters being read or written by the statements extracted/moved, the value returned by the extract method, or breaking/adding overriding/overloading constraints. Therefore, often faults are introduced when these refactorings are performed manually [119; 61]. Moreover, not even well-known refactoring tools are free of injecting faults [111].

Although several types of faults could be expected, we narrow down our study to two specific faults: *a statement deletion* (SD) and *a relational operator replacement* (ROR). Nevertheless, as more complex faults can be decomposed into small steps such as statement manipulation, we believe that other faults would present similar results. We also center our investigation on the four classes of commonly impacted source code elements presented in Section 3.1 (M, C, Ce, and O).

3.2.1 Study Setup

We select three open source Java projects as experimental units: XML-Security[1] ($\approx$ 17 KLOC), a library that provides security APIs for manipulating XML documents, such as authorization and encryption; JMock[2] ($\approx$ 5KLOC), a library for testing applications with mock objects; and EasyAccept[3] ($\approx$ 2 KLOC), a tool that helps create and run acceptance tests.

We mine three random versions of each program from their repository history. Each version is associated with a regression test suite that was manually created by its developers for system integration purposes. Then, for each version, we perform five extract method edits each with a single statement deletion, and five with a relational operator replacement. Similarly, ten move methods are performed for each version, five with a single statement delete, and five with a relational operator replacement. Thus, 180 faulty versions are created; 60 for each project, in which 30 are related to extract method faults and 30 to move method faults.

Refactoring Faults. Each of the 180 faulty versions differs from the original only by a single erroneous refactoring edit. As our study deals with two refactoring types and two refactoring faults, we establish a fault seeding process for each combination "refactoring X fault" (Tables 3.1 and 3.2). Each process injects, without compilation errors, a refactoring related mutant. Although dealing only with injected faults, recent studies (e.g [57]) state that

[1]http://xml.apache.org/security

[2]http://jmock.org/

[3]http://easyaccept.sourceforge.net/

mutant/injected faults are valid substitutes for real faults in software testing.

Table 3.1: Seeding fault processes for extract method.

Seeding a *statement deleting* fault in an extract method edit:

Being S the set of classes from a given project version.

1. Randomly select a class C from S;
2. Randomly select a method `m` from C; its body must not be empty;
3. Randomly select a statement `s` from `m`;
4. Extract `s` to a new method, `newMethod`. The new method's name is selected from a pool of names prepared in advance;
5. Insert the fault by removing a random statement `st` from `newMethod`;
6. A compilation error is found, undo the changes and go back to step 1;

Seeding a *relational operator replacement* fault in an extract method edit:

Being S the set of classes from a given project version.

1. Randomly select a class C from S;
2. Randomly select a method `m` from C; its body must have at least a statement that uses a relational operator (==, ! =, >, <, >= or <=);
3. Randomly select a statement `s` from `m` that uses a relational operator;
4. Extract `s` to a new method, `newMethod`. The new method's name is selected from a pool of names prepared in advance;
5. If the extracted method generates a compilation error, undo the extraction and go back to step 1;
6. Insert the fault by replacing the operator from `s` with another one.

The selected faults (statement deletion and relational operator replacement) simulate scenarios in which a developer that performs a refactoring may mistakenly introduce behavior changes. The *statement deletion* fault emulates a scenario in which a developer comments a statement while refactoring, and forgets to uncomment after he is done. Figure 3.2 exemplifies such fault in XML-Security. Line 6 in `generateDigestValues()` should have been extracted to `generate(Reference)`. However, this statement is commented out in the extracted method (Figure 3.2-b), changing the previous behavior without generating compilation errors. Similarly, we also consider scenarios in which the developer does not copy all statements from the original method to the new one.

The *relational operator replacement* fault emulates a situation in which a developer combines a refactoring with other changes. This scenarios goes according to Murphy-Hill et

Table 3.2: Seeding fault processes for move method.

Seeding a *a statement deleting* fault in a move method edit:
Being S the set of classes from a given project version.
1. Randomly select a class C from S;
2. Randomly select a method `m` from C; its body must not be empty;
3. Randomly select a class C_2;
4. Move `m` from C to C_2;
5. Insert the fault by removing a random statement `st` from `newMethod`;
6. A compilation error is found, undo the changes and go back to step 1;

Seeding a *relational operator replacement* fault in a move method edit:
Being S the set of classes from a given project version.
1. Randomly select a class C from S;
2. Randomly select a method `m` from C; its body must have at least a statement that uses a relational operator ($==$, $!=$, $>$, $<$, $>=$ or $<=$);
3. Randomly select a class C_2;
4. Move `m` from C to C_2;
5. If the moving generates a compilation error, undo the moving and go back to step 1;
6. Randomly select a statement `s` from `m` that uses a relational operator;
7. Insert the fault by replacing the operator from `s` with another one.

al.'s [81] that conclude that developers often interleave refactorings with other transformations. Recent studies [11] affirm that *extra edits* are worth revision since they may interfere with the software's previous behavior. Figure 3.3 exemplifies such fault in a move method edit applied to XML-Security. Method `getProviderIsRegisteredAtSecurity` is moved from `JCEMapper` to `Algorithm` along with the replacement of the operator $!=$ for $==$ (line 5). This extra change modifies the behavior of the program without introducing compilation errors.

Metrics. We run each project's test suite on its 3 versions, for all 180 faulty versions. Then, we measure the following properties: M: number of test cases that cover the refactored method; C: number of test cases that cover the callers of the refactored method; Ce: number of test cases covering the callees of the refactored method; and O: number of test cases that cover methods with similar signature of the newly added methods (by similar signature we mean methods with the same name and return type). We also measure how many of the test cases in fact detect the refactoring fault: D_M: number of test cases that cover the refactored method and detect the fault; D_C: number of test cases that cover the caller of the refactored method and detect the fault; D_{Ce}: number of test cases that cover the callees of the refactored

```
 1 class Manifest{
 2  void generateDigestValues() throws...{
 3   if (this.state == MODESIGN) {
 4      for (int i = 0;i < this.references.size();i++){
 5        Reference currentRef = (Reference) this.references.elementAt(i);
 6        currentRef.generateDigestValue();
 7      }
 8      ...
 9    }
10  }
11  ...
12 }
```

(a) original code

```
 1 class Manifest{
 2  void generateDigestValues() throws...{
 3   if (this.state == MODESIGN) {
 4      for (int i = 0;i < this.references.size();i++){
 5        Reference currentRef = (Reference) this.references.elementAt(i);
 6 -       currentRef.generateDigestValue();
 7 +       generate(currentRef);
 8       }
 9       ...
10    }
11  }
12 +  void generate(Reference currentRef) throws...{
13 +       /* Missing statement */
14 +       // currentRef.generateDigestValue();
15 +       }
16  ...
17 }
```

(b) extract method refactoring with missing statement

Figure 3.2: An example of a seeded statement deletion fault when extracting a method. (a) The original code. (b) Code after an extract method refactoring. Lines 6 is extracted to create a new method `generate`. Since the extracted statement is commented out, the behavior of `generateDigestValues()` has changed.

method and detect the fault; and D_O: number of test cases that cover methods with similar signature of the newly added methods detect the fault. For collecting coverage data we use the EclEmma tool[4].

[4]http://www.eclemma.org/

```
class JCEMapper{
  Algorithm algorithm;
  boolean getProviderIsRegisteredAtSecurity(String providerId) {
    java.security.Provider prov =
       java.security.Security.getProvider(providerId);
    if (prov != null) {
      return true;
    }
    return false;
  }
  ...
}
class Algorithm{...}
```

(a) original code

```
class JCEMapper{
  Algorithm algorithm;
  ...
}
class Algorithm{
  boolean getProviderIsRegisteredAtSecurity(String providerId) {
    java.security.Provider prov =
       java.security.Security.getProvider(providerId);
-    if (prov != null) {
+    if (prov == null) {
       return true;
    }
    return false;
  }
}
```

(b) move method refactoring with a relational operator replacement.

Figure 3.3: An example of a seeded move method fault. (a) The original code. (b) Code after a move method refactoring. Method `getProviderIsRegisteredAtSecurity` is moved from `JCEMapper` to `Algorithm`. Since the relational operator is changed, the behavior of program has changed.

Details about artifacts used in our study (subjects, faults and metrics) are available at our website [1].

3.2.2 Results and Discussion

Appendix A presents the detailed tables regarding the results of our study. The largest regression suites are in JMock – the number of test cases in each suite vary according to the collected version. For instance, Version 1 in JMock is supported by 195 test cases, while Version 2 presents 222 tests. Since all versions are subject to a seeded refactoring fault, in all projects more than one version presents no failing test cases, for instance versions "$xv1/f1/em + ds$", "$jv2/f4/mm + ds$" and "$ev1/f1/mm + ror$" (see Appendix A). In every such cases, test cases do not cover either the changed method, its callers, its callees, or methods with similar signatures.

Analyzing the Suite's Fault Detection Capacity

Overall, only 67% of the seeded faults are detected by the projects' test suites. Thus, in 33% of the faults, a successful test suite run would be misleading. The least effective test suite for those faults is in XML-Security – 34 out of 60 faults are missed; EasyAccept misses 16 out of 60. Better results are observed in JMock, with only 10 missing faults. This result evidences that test suites might lack effectiveness when detecting refactoring faults, reinforcing conclusions from previous researches (e.g., [63; 7]). textcolorredHowever, the practice shows that developers do most of their refactoring validations through testing. This fact might be that it is hard to replace the validation power of a quality suite created by expert developers.

We group those results by refactoring and fault type. Moreover, we also analyze each combination "refactoring type X fault type". In all these analysis we observe similar rates of refactoring fault detection. Considering all extract method edits, or all move method edits, the suites detect only 67% of the faults. When grouping by fault type, this rate drops to 61% for *delete statement*, and increases to 72% for *relational operator replacement*.

Based on those results we perform a hypothesis test of proportion in which we compare the proportions regard rate of refactoring fault detection pair to pair. Table 3.3 shows the

Table 3.3: Null and alternative hypothesis regard the proportion tests.

H0	H1
$C_{EM} = C_{MM}$	$C_{EM} \neq C_{MM}$
$C_{DS} = C_{ROR}$	$C_{DS} \neq C_{ROR}$
$C_{EM \wedge DS} = C_{EM \wedge ROR}$	$C_{EM \wedge DS} \neq C_{EM \wedge ROR}$
$C_{MM \wedge DS} = C_{MM \wedge ROR}$	$C_{MM \wedge DS} \neq C_{MM \wedge ROR}$
$C_{EM \wedge DS} = C_{MM \wedge DS}$	$C_{EM \wedge DS} \neq C_{MM \wedge DS}$
$C_{EM \wedge ROR} = C_{MM \wedge ROR}$	$C_{EM \wedge ROR} \neq C_{MM \wedge ROR}$

hypothesis established for this test, in which C refers to a suite's capacity of detecting refactoring faults, EM to an Extract Method edit, MM to a Move Method edit, DS to a Delete Statement fault, and ROR to a Relational Operator Replacement fault. The null hypotheses state that there is no difference on the proportions when comparing by refactoring type (first line), type of fault (line two), and the combinations of type of refactoring and type of fault (lines 3 to 6). With a 95% confidence level, no statistical differences are found to any of the proportion tests. Thus, we can conclude that, for those systems, *the suites' capacity of revealing refactoring faults remains stable, not depending on type of refactoring nor type of fault*. In fact, we can say that around one third of the refactoring faults (33%) goes undetected by the suites. This fact evidences the risks of trusting 100% on general propose test suites alone for validating refactorings. There is a need for creating test cases specialized on checking behavior preservation.

Besides coverage of commonly impacted elements, we believe that other factors might have influenced our results, such as, systems characteristics, general testing coverage, etc. However, this impact remains to be assessed.

Undetected Faults

By observing the coverage data after running our exploratory study, we can then relate the scenarios in which a test suite is not able to detect a refactoring fault (Tables from Appendix A when FTC = 0 - no fault detection) to the coverage of impacted elements (M, C, Ce and O). For that, we investigate test coverage for each type of element in isolation, and also their combination, when the fault is not detected. The results are grouped by type of refactoring (Extract Method and Move Method - Table 3.4-a, second and third columns) and by type of

(a)

	$FTC = 0$				$FTC \neq 0$			
	Type		Fault		Type		Fault	
	EM	MM	DS	ROR	EM	MM	DS	ROR
$M = 0$	60%	53%	54%	60%	88%	95%	91%	92%
$C = 0$	44%	37%	40%	40%	81%	85%	76%	89%
$C_e = 0$	20%	17%	20%	16%	48%	50%	53%	46%
$O = 0$	10%	7%	14%	0%	3%	3%	5%	1%
$(M = 0) \wedge (C = 0)$	43%	30%	37%	36%	80%	83%	74%	88%
$(M = 0) \wedge (C_e = 0)$	20%	17%	20%	16%	48%	47%	51%	45%
$(M = 0) \wedge (O = 0)$	10%	7%	14%	0%	3%	3%	5%	1%
$(C = 0) \wedge (C_e = 0)$	13%	13%	17%	8%	45%	22%	45%	23%
$(C = 0) \wedge (O = 0)$	10%	3%	11%	0%	3%	22%	4%	20%
$(C_e = 0) \wedge (O = 0)$	0%	3%	3%	0%	0%	3%	4%	0%
$(M = 0) \wedge (C = 0) \wedge (C_e = 0)$	13%	13%	17%	8%	43%	20%	43%	21%
$(M = 0) \wedge (C = 0) \wedge (O = 0)$	10%	3%	11%	0%	3%	23%	4%	21%
$(M = 0) \wedge (C_e = 0) \wedge (O = 0)$	0%	3%	3%	0%	0%	3%	4%	0%
$(C = 0) \wedge (C_e = 0) \wedge (O = 0)$	0%	3%	3%	0%	0%	2%	2%	0%
$(M = 0) \wedge (C = 0) \wedge (C_e = 0) \wedge (O = 0)$	0%	3%	3%	0%	0%	2%	2%	0%

(b)

	General	
	$FTC = 0$	$FTC \neq 0$
$M = 0$	57%	92%
$C = 0$	40%	83%
$C_e = 0$	18%	49%
$O = 0$	8%	3%
$(M = 0) \wedge (C = 0)$	37%	82%
$(M = 0) \wedge (C_e = 0)$	18%	47%
$(M = 0) \wedge (O = 0)$	8%	3%
$(C = 0) \wedge (C_e = 0)$	13%	33%
$(C = 0) \wedge (O = 0)$	6%	12%
$(C_e = 0) \wedge (O = 0)$	2%	2%
$(M = 0) \wedge (C = 0) \wedge (C_e = 0)$	13%	32%
$(M = 0) \wedge (C = 0) \wedge (O = 0)$	7%	13%
$(M = 0) \wedge (C_e = 0) \wedge (O = 0)$	2%	2%
$(C = 0) \wedge (C_e = 0) \wedge (O = 0)$	2%	1%
$(M = 0) \wedge (C = 0) \wedge (C_e = 0) \wedge (O = 0)$	2%	1%

Table 3.4: Analysis based on coverage of mainly impacted code elements.

faults (Delete Statement and Relational Operator Replacement - Table 3.4-a, fourth and fifth columns). We also put all results together to have an overview (Table 3.4-b, first column).

Our results show that the lack of test cases calling the method whose body was changed (M) seems to be very relevant – 57% of the unrevealed faults present this property (60% for extracted method, 53% for move method, 54% for delete statement, and 60% for relational operator replacement). Similarly, 40% (EM: 43%, MM: 37%, DS: 40%, ROR: 40%) of the undetected faults do not cover the callers of the changed methods. The changed method and its callers are missed altogether in also 37% (EM: 43%, MM: 30%, DS: 37%, ROR: 36%) of the faults. As expected, these data indicate that chances of detecting refactoring faults are higher if test cases targeting the refactorings main elements – changed method and its callers – are included in the suite. In addition, techniques that generate test cases focusing on the impacted elements should be prioritized in refactoring scenarios; although this is visible to Extract Method and Move Method, other similar refactorings should benefit as well – Extract Class and Pull Up Method are representative examples.

Although less important, other combinations are worth discussing and should be considered by the tester. For instance, in 20% of the cases (EM: 20%, MM: 17%, DS: 20%, ROR: 16%) when a suite cannot detect a fault, there are no tests covering neither the changed method callers nor its callees. Thus, after assuring that both the changed method and its callers are well tested, if the tester has still time and resources available, it is worthy to head efforts in testing the method's callees in order to increase even more the suite's chances of detecting faults.

Detected Faults

In a similar analysis, we investigate the 120 cases in which the refactoring faults are detected by using the program's suite ($FTC \neq 0$). Table 3.4-a sixth to ninth columns and Table 3.4-b second column summarize this analysis by grouping the results by type of refactoring, type of fault and general overview. Again, our results evidence the importance of having tests that exercise the modified method. In 92% (EM: 88%, MM: 95%, DS: 91%, ROR: 92%) of the cases that identified a fault, there is at least one test case that covers the refactored method. Considering callers, this rate is also high - 83% (EM: 81%, MM: 85%, DS: 76%, ROR: 89%), and the combination of M and C presents a rate of 82% (EM: 80%, MM: 83%, DS: 74%, ROR: 88%). Moreover, Figure 3.4 shows the density charts for each impacted elements and their combination. Each chart relates coverage of each impacted elements (M, C, Ce and

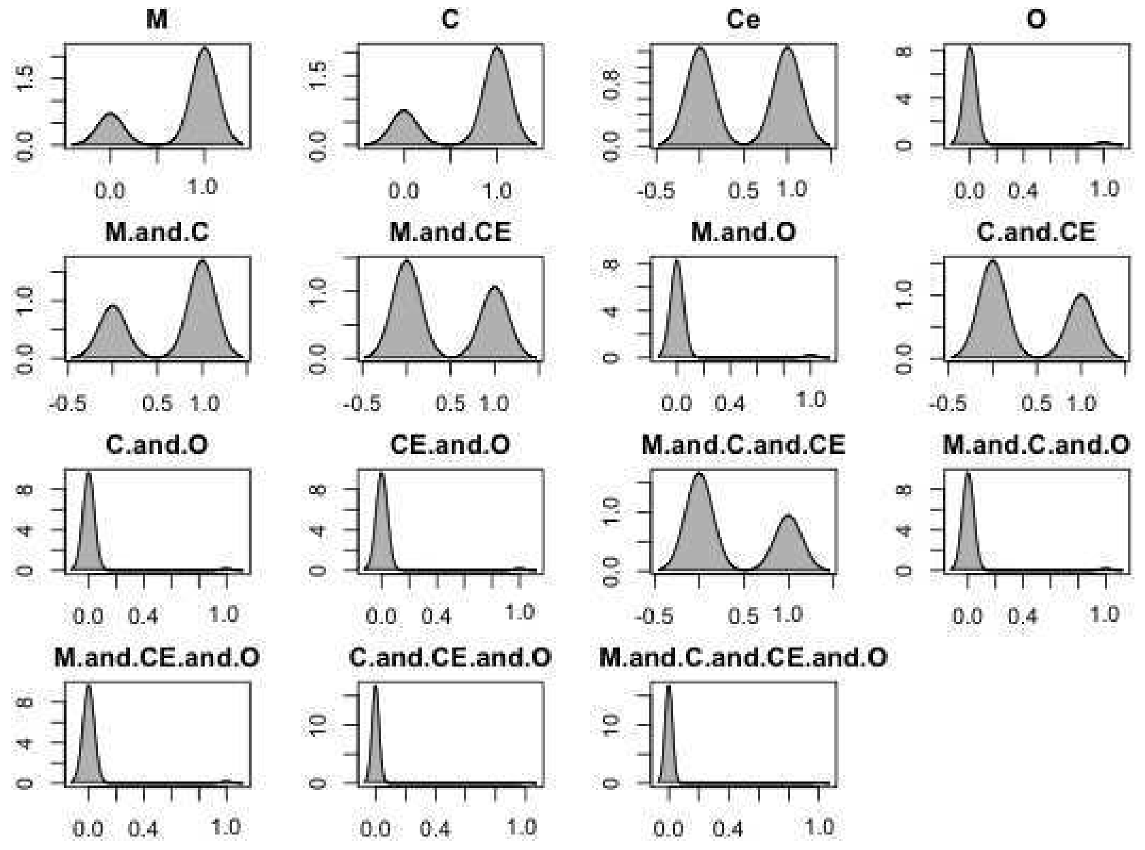


Figure 3.4: Density charts relating fault detection and coverage of impacted elements (M, C, Ce and O).

O) to detection of faults (1 means a fault is detected and 0 it is not detected). The density charts reinforces the numeric results discussed above. The density curves clearly tend to be higher in 1 (fault detected) when there is test coverage of M, C and $M \wedge C$. On the other hand, the density curves tend to be higher in 0 (fault not detected) for the other elements and combinations. Moreover, the correlation coefficients between fault detection and each class of elements (M, C, and $M \wedge C$) are at least three times higher then all the other elements and their combinations. These results corroborate our previous conclusion that these two classes of impacted elements are indeed the most important ones when aiming to validate an refactoring edits. Test suites that have low coverage of those elements might be ineffective.

Our results show that test cases that cover methods with similar signature (O) has very little impact on testing/detecting refactoring faults. However, our fault seeding process was able to emulate those scenarios in only 12 of the 180 cases (7%). Those situations are hard to emulate in a non manual seeding process (we randomly select code elements such method, statements and method name to be part of the refactoring edits). However, considering only those cases, test cases that call those similar methods detected the fault only in one case.

Cases like *"xv3/f1/em+ds"* and *"jv2/f5/em+ds"* (Table in Appendix A) are interesting to mention. In *xv3/f1/em+ds* there are several test cases that cover both elements that our investigation points as worth testing (M and C), and in *"jv2/f5/em+ds"* we have a high number of test cases that exercise one of those classes (76 test cases cover C). In both cases the test suite is unable to detect the fault. Those cases illustrate that, although very common, there is no strong correlation between testing coverage and fault detection, confirming results from other works (e.g. [35; 52]). However, our results show that method coverage is still a good measure for detecting refactoring faults in general. When we look closely to those cases, we see that although covering M and/or C, those tests do not cover 100% of refactored method branches. They also have a very low testing data variability, i.e., the data used as input to the test cases are very similar. Those are factors that can mislead a judgment based on pure method coverage. In Section 3.2.4 we investigate whether the combination of method and branch coverage can help to diminish the uncertainty of those cases.

Based on our results, we can answer the first research question of this work (RQ1), by concluding that in fact test coverage of commonly impacted locations is in general a good measure to predict a suite's capacity on detecting refactoring faults. Also, the lack of testing coverage on the mainly impacted elements can be considered a weakness, and be the starting point to guide developers to augment and improve a suite. Similarly, our study identifies that two classes of impacted elements (the method under refactoring and its callers) must be the first priority when testing an extracted method edit. Test cases that cover those elements are more likely to reveal refactoring faults.

3.2.3 A Coverage-based Model for Predicting Detection of Refactoring Faults

After observing the relationship between the testing coverage of M, C, and/or $M \wedge C$ and refactoring fault detection, we propose a regression model based on our study data. This model summarizes coverage data and refactoring fault detection, aiming to help developer-s/testers to predict the chances of their suite regards its capacity of detecting possible refactoring faults that might have been introduced.

Before building the model we first calculate the correlation coefficients matrix of all class of elements (M, C, Ce, and O) and their combinations in order to visualize which ones should be used in our prediction model. Agreeing with the conclusions discussed in the previous sections, only M, C, and $M \wedge C$ prove to be essential, the other elements and combinations shown to be statistically equivalent to at least one of the selected variables, thus they can be removed from the regression model equation without much loss. To build our prediction model we apply a *Logistic Regression* [48] by using a random sample of 50% of our data set results. For that, we use R[5], an environment for statistical computing. The Logistic Regression is a statistical method used to predict a binary response from at least one categorical predictor. In our case, the predictors refer to the coverage of certain impacted element (0 if the element is not coverage, 1 if it covered).

Table 3.5 shows the coefficients of each predictor from our model and their p-values, while Equation 3.1 presents the proposed predictor model. Our model aims to infer whether a suite is likely to detect refactoring faults. The closer to 1, better the chances of, if there is, a refactoring fault be detected based on the coverage of mostly impacted elements.

To test the fit of our model we use our whole data set. Our model shows to be effective by correctly predicting the detection or non detection of refactoring faults in 80% of the cases. Table 3.6 presents the confusion matrix when we use our model against the reference data regards detection. Therefore, although not general, we believe that our model can work as a tool for helping developers to increase confidence about their suite's capacity of revealing refactoring faults. Moreover, when there is a need, it can be used for driving efforts on deciding how to augment a test suite.

Precision of this model can still be improved with further information such as different coverage analysis or other metrics. In the next section we present a study that motivates further research in this direction. Moreover, this model still needs to be evaluated using different data sets. Our model is limited to the two refactoring types and refactoring faults used in our study. However, we believe that it can also be used when dealing with similar refactorings and faults. For instance, the pull up method refactoring's code edits are comparable to the ones applied when performing a move method (method deletion in a certain class, method

[5]http://www.r-project.org/

addition in another class, etc). Thus, test cases that would validate both types of refactoring seem to work in a similar way.

Table 3.5: Regression statistics

	Coefficient	**p-value**
Intercept	-2.1595	0.000397
M	3.7689	1.79e-05
C	1.8718	0.021557
$M \wedge C$	-2.1054	0.046224

Table 3.6: Model confusion matrix

		Reference	
		0	1
Prediction	0	34	9
	1	27	110

$$D = 1/(1 + e^{-(-2.1595+3.7689M+1.8718C-2.1054M \wedge C)}) \tag{3.1}$$

3.2.4 Variation: Analyzing Branch Coverage

Although our overall results show that, by covering the modified method and its callers, we maximize the chances of detecting faults from both Extract Method and Move Method, some specific faults are not detected, even though test cases cover those element types. For instance, *"xv3/f1/em+ds"* (Appendix A), although there are 36 test cases that cover both the refactored method and its callers, which corresponds to 37% of the test suite, none of them detects the fault. In order to provide a more thorough analysis, we extend the study by selecting the subset of faults for which there is at least one test case covering the changed method and/or its callers ($M \neq 0$ and/or $C \neq 0$). In this scenario, we take a white-box approach to investigate which branches (execution paths) of the original method those test cases cover. Moreover, we establish two ranges to categorize levels of branch coverage: $[0; 25\%]$ as low, and $[75\%; 100\%]$ as high. For instance, in *"ev2/f5/em+ds"*, the single test case that exercises the changed method (`Script.allErrorMessages()`) covers 100% of its branches (high branch coverage for M), while its callers, that are exercised by four test cases, cover 0% of `Script.allErrorMessages()` (low branch coverage for C). Table 3.7 summarizes the results for this analysis.

In 88% of the cases, the fault is detected as long $M \neq 0$ and the branch coverage of the refactored method due to M is high ($\geq 75\%$). Similarly, in 86% of the cases a suite detects a refactoring fault when $C \neq 0$ and those methods exercise more than 75% of the branches.

Moreover, in only 12% and 15% of the cases a high branch coverage is not associated with a fault detection.

When the branch coverage is low ($\leq 25\%$), this rate drops to approximately 50% (for C it drops to 19%). So, for those three projects, a high branch coverage of the refactored method seems to increase the chances of detecting refactoring faults.

To better analyze whether the combination of the coverage of M, C or $M \wedge C$ with a high branch coverage would improve the detection faults, from our results, we observe the proportion of detected faults when the main impacted elements are covered (65%), and the proportion of detected faults when, besides the coverage of those elements, a high branch coverage is found (67%). As we can see the combination of coverage of impacted elements and high branch coverage ended up improving the fault detection by 2%. Although, no statistical difference is found when comparing both strategies in isolation, the combined analysis ended up eliminating several situations in which there were coverage of impacted elements and no fault detection. In some cases, although the mainly impacted elements are covered, the refactoring faults are not detected due to a low branch coverage.

Although helpful when applied in isolation, coverage of M and C alone might be misleading. In some cases, even having tests covering M and C with a high branch coverage ($\geq 75\%$), do not lead to fault detection. For instance, in *ev/v1/ror* 11 test cases cover both M and C with a branch coverage of 100%, however those test cases do not reveal the seeded fault. Looking closely these 11 test cases we can see that although covering all branches there is low variability of test data, i.e., the data used in the test case are quite similar. That is the reason these faults remain undetected. Thus, we can say that our results have shown evidences that by having test cases covering M and C with a high branch coverage we increase the chances of revealing refactoring faults. However, this analysis is not 100% efficient.

3.2.5 Threats to Validity

In terms of *construct validity*, the accuracy of EclEmma for extracting coverage data and the coded R functions for calculating statistical tests might directly affect our study results. However, both tools have been widely used in practice by developers and researchers, which attests that they rely on their results. Moreover, we also manually validated Eclemma's and our functions' results by using limited samples.

Table 3.7: Branch coverage analysis

M, C or $M \wedge C$ are covered + Branch coverage $\geq$ 75%		
	M	C
# of occurrences	85	73
% of detected faults	88	86
% of not detected faults	12	14

M, C or $M \wedge C$ are covered + Branch coverage $\leq$ 25%		
	M	C
# of occurrences	8	16
% of detected faults	50	19
% of not detected faults	50	81

In terms of *internal validity*, we measure only method and branch coverage. Other coverage metrics (e.g., decision coverage) and the testing data might be influential factors to our results. We plan to investigate the impact of those factors as future work.

In terms of *external validity*, our evaluation results certainly do not generalize beyond the three studied projects. However, we tried to minimize this problem by selecting projects with different code and test suite sizes. In projects build in a less controlled environment, it is usually harder to build and maintain an effective test suite. Industrial projects, on the other hand, with tighter QA practices, might present different results. Moreover, all test suites used in our study were manually created to validate the behavior at the system level for integration purpose. Thus, the results be different if considering suites with different styles and purpose, *e.g.*, automatically generated suites, test acceptance suites.

Also, we focused on the Extract Method and Move Method refactorings. Different refactoring edits would present their own singularity, depending on the type of elements impacted by the change. Nevertheless, several of the most applied refactoring edits – such as Pull Up Method or Encapsulate Field [33] – involve creating new methods, and adding new calls to them, thus results might be comparable to those refactorings. Regarding seeded faults, the ones applied to our study represent typical mistakes when performing local manual refactorings. Extract Method and Move Method involve a handful of copy and paste commands on the IDE, and subtle differences between the original and the modified code could induce those kind of errors. Moreover, a recent study [57] affirm that there is a significant correlation between injected faults (mutants) detection and real fault detection.

3.2.6 Study on Binding-related Faults

Reference binding problems are faults that developers often face when refactoring [13]. Refactoring edits usually deal with the moving, renaming and replacement of source code elements. A developer working with systems with several hierarchy levels may have a hard time to make sure that all his edits preserve all method and variable references, not breaking or adding any overriding/overloading constraints. Recent studies have emphasized how easy is to include a binding problem when refactoring, even when using well-known automatic refactoring tools [108; 11].

Aiming to test whether the conclusions of our empirical study are also valid when dealing with this important refactoring fault we run a second exploratory study using the XML-Security project. This system uses class hierarchies in several packages to better distribute the system's behavior and avoid coupling. In our study we manually create four faulty versions of the original XML-Security code by applying a single problematic Move Method in each version. Each move method edit ends up changing the system's original behavior by breaking or adding a method overriding/overloading, and consequently introducing a method binding problem. None of the edits generates compilation errors. Figure 3.5 exemplifies a faulty move method applied to the XML-Security's code. Method `Canonicalizer20010315Excl.engineCanonicalizeXPathNodeSet(Set,-String)` is moved to `Canonicalizer20010315ExclOmitComments`. However, in the output code, the call of `engineCanonicalizeXPathNodeSet(Set,String)` in `Canonicalizer20010315Excl.engineCanonicalizeXPathNodeSet` refers to the method from `CanonicalizerBase` instead of `Canonicalizer20010315ExclOmitComments`. The underlined lines refer to the location where the binding problem can be identified after the problematic move method edit.

All four faults are detected by the XML-Security's test suite. For each faulty version we analyze the test coverage of the mostly impacted elements (moved method - M, callers - C, callees - Ce, methods with similar signature - O). Our analysis shows that, in all four cases, M and C are covered. However, in one case, although covering C with 18 test cases, none of those tests fail due to the fault. It is valid to discuss that in all four cases, there is at least one test that covers O and detects the faults. As expected, this fact shows that this class of elements can be important for detecting binding problems. Moreover, we believe that O can also be helpful when debugging binding refactoring faults, since these elements focus on methods that may lead to the breaking/adding of override/overloading contracts. For instance, considering the binding problem depicted in Figure 3.5, one of the test case that reveals this fault exercises the method `engineCanonicalizeXPathNodeSet(Set)`. By noticing that this test case used to pass before the edit and that it was not directly modified during the refactoring, a developer can infer that the edit may have added or broken an overloading/overriding contract, which in fact happened. Thus, the results of our case study show us evidences that our previous conclusions might be still valid when detecting biding problems. M and C are impacted elements that when covered increase the chances of detecting those faults. However, other impacted elements, mainly O, may also deserve attention.

3.3 Concluding Remarks

This chapter reports a study that investigate the relationship between program elements impacted by the Extract Method or Move Method refactoring and test coverage. Our exploratory studies provide evidences that by exercising two different classes of impacted elements, the modified method and its callers, we are more likely to detect refactoring faults, similar to DS and ROR, related to both Extract Method and Move Method edits. Also, if those tests cover a high number of branches of the original method the chances of detecting those faults are even higher. On the other hand, the lack of testing coverage for those elements may strongly decrease a suite's capability for detecting refactoring faults. Moreover, we derived from our study results a statistical coverage-based model that aims to help developers to predict whether a test suite is likely to detect the refactoring fault.

A few guidelines to developers, obtained from the results of the reported study, seem appropriate. When a developer is performing a refactoring edit, and he mistrusts the test suite as a safety net for avoiding behavioral changes, it seems wise to perform a previous analysis on the test suite. In this analysis, he might identify whether the two main types of

```
class CanonicalizerBase {
  byte[] engineCanonicalizeXPathNodeSet(Set xpathNodeSet, String inclusiveNamespaces)
     ...{
    if (this.xpathNodeSet.size() == 0) {
      return new byte[0]; } ...   }  }
class Canonicalizer20010315Excl extends CanonicalizerBase {
 byte[] engineCanonicalizeXPathNodeSet(Set xpathNodeSet, String inclusiveNamespaces) ...
     {
   try { this.renderedPrefixesForElement = new HashMap();
       return super.engineCanonicalizeXPathNodeSet(xpathNodeSet);
    } finally { ... }  }
 byte[] engineCanonicalizeXPathNodeSet(Set xpathNodeSet) ... {
    return this.engineCanonicalizeXPathNodeSet(xpathNodeSet, "");
 } }
class Canonicalizer20010315ExclOmitComments{ ... }
```

(a) Original version.

```
class CanonicalizerBase {
  byte[] engineCanonicalizeXPathNodeSet(Set xpathNodeSet, String inclusiveNamespaces)
     ... {
    if (this.xpathNodeSet.size() == 0) {
      return new byte[0];} ... } }
class Canonicalizer20010315Excl extends CanonicalizerBase{
- byte[] engineCanonicalizeXPathNodeSet(Set xpathNodeSet, String inclusiveNamespaces)
     ... {
-   try { this.renderedPrefixesForElement = new HashMap();
-       return super.engineCanonicalizeXPathNodeSet(xpathNodeSet);
-    } finally { ... }  }
 byte[] engineCanonicalizeXPathNodeSet(Set xpathNodeSet) ... {
    return this.engineCanonicalizeXPathNodeSet(xpathNodeSet, "");
 } }
class Canonicalizer20010315ExclOmitComments{
+  byte[] engineCanonicalizeXPathNodeSet(Set xpathNodeSet, String inclusiveNamespaces)
     ... {
+   try { this.renderedPrefixesForElement = new HashMap();
+       return super.engineCanonicalizeXPathNodeSet(xpathNodeSet);
+   } finally { ... }  }
}
```

(b) Code with a binding problem due to a Move Method.

Figure 3.5: Binding problem example.

impacted elements (refactored method and callers of the refactored method) are thoroughly exercised. Then, by applying the proposed statistical model he gets a effectiveness level for the test suite. This information can be used to help him to decide whether is safe to perform the refactoring edit, or whether there is room for a test suite improvement. Moreover, by knowing the locations not efficiently tested (e.g., low branch coverage), he may direct efforts to developing tests that improve this deficiency.

Based on those results, in the next chapter, we propose a novel refactoring-based prioritization technique that aims to accelerate the detection of refactoring faults focusing on test cases that cover commonly impacted code locations.

Chapter 4

The Refactoring-Based Approach

The use of regression test suite as safety net when performing refactoring is known to be a common practice in real projects [63]. However, as refactoring edits are often applied in a great number [108], it might be impractical rerun and analyze the execution results of the whole test suite after each refactoring edit. Moreover, a regression test suite run is often costly and time-consuming [24].

Test case prioritization is frequently used to speed up the achievement of certain testing goal. Although general-purpose solutions often produce acceptable results, software engineering specific problems may require specific and/or adaptive solutions [58; 74; 73]. With this fact in mind, we have performed a set of exploratory studies using real open-source projects in which we investigate how six of the most used general-purpose prioritization techniques behave when dealing with refactoring fault detection [5; 4; 7]. Results of these studies show that those techniques perform poorly when placing refactoring fault-revealing test cases – fault revealing test cases were placed in the top of the prioritized suite only in 35% of the cases. Moreover, to the best of our knowledge there is no prioritization technique specialized on refactoring fault detection. In this chapter we present the *Refactoring-Based Approach* (RBA) [7; 6], a technique for prioritizing test cases guided by refactoring edits. This technique reorders test cases assuming that a test case is more likely to detect a refactoring problem if it covers the locality of the edits, and/or the commonly impacted elements; idea that goes according to the conclusions of the study presented in Chapter 3. The RBA approach is currently part of a test case prioritization environment for Java systems, PriorJ [98; 10] (details in Chapter 6).

RBA's Equivalence Definition RBA's equivalence notion establishes that a refactoring transformation is behavior-preserving when all test cases pass before and after the refactoring edits. Otherwise, we consider that a refactoring fault was during refactoring. Equation 4.1 summarizes RBA's equivalence definition:

Being:

- P, the original version of a program;
- P', P after a refactoring;
- T, a test suite for P;
- t, a test case from T;
- $f : \{t \in T, P \vee P'\} \rightarrow \{pass, fail\}$, a function that executes test case t against the program P or its refactored version, P'.

$$\forall t \in T, f(t, P) = pass \wedge f(t, P') = pass \tag{4.1}$$

4.1 Overview

Refactoring-Based Approach (RBA) is a prioritization solution that focuses on rescheduling test cases to speed up the detection of behavioral changes introduced after refactoring. Figure 4.1 gives an overview of RBA - rounded-edge rectangles represent activities, dotted-rounded-edge rectangles represent input or output artifacts, and arrows indicate flow between activities or between an activity and an artifact.

Inputs and Outputs RBA requires inputs that are commonly available when a refactoring task is applied:

- *Two consecutive versions of a program.* The *base* version – a stable version of the program that has its behavior tested by a test suite; and the *delta* version – the version after refactoring edit(s). The behavior of the *delta* version remains untested;
- *A test suite.* A set of test cases that reflects the behavior of the *base* version.

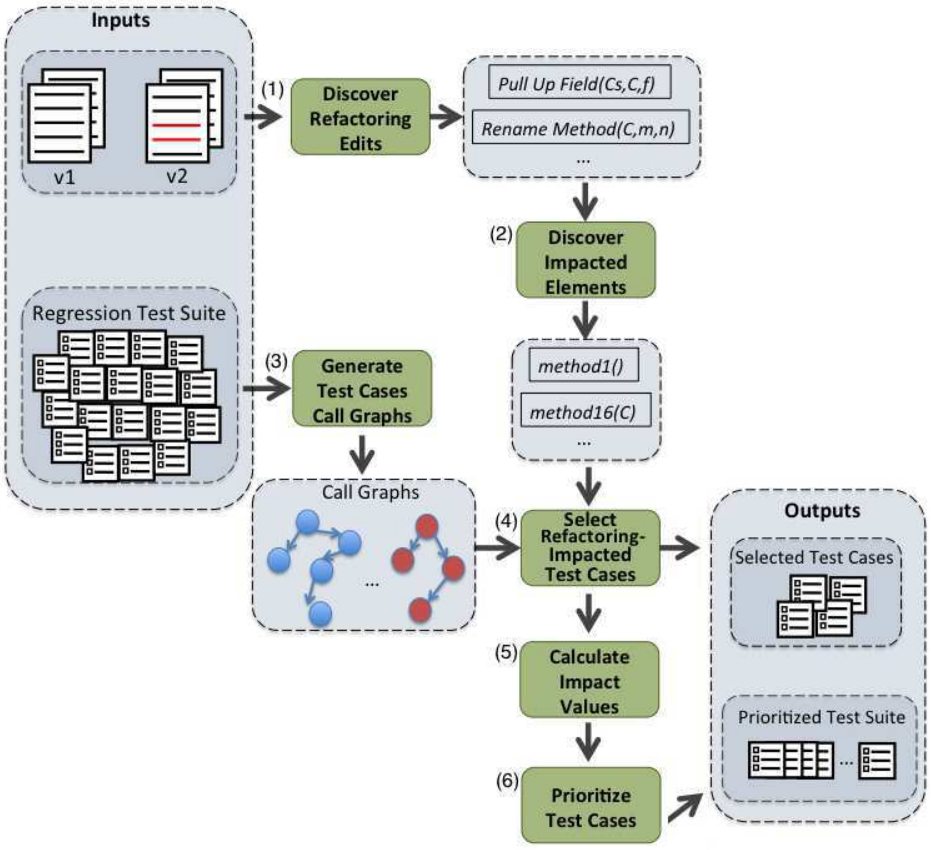


Figure 4.1: RBA overview.

As output, RBA generates two artifacts. The developer/tester must decide whether one or both output artifacts are needed:

- *A selected suite*. A subset of test cases composed only by tests that are related to the performed changes, according to our Refactoring Fault Models (RFM, Section 4.2);
- *A prioritized test suite*. A suite with the exactly same size of the original regression suite, but in a new execution order.

Guiding example In the following sections, we present RBA along with a guiding example. Suppose a developer performs the pull up method edit shown in Figure 4.2. Method `k (int i)` is moved from class `B` to its superclass, `A`. Although simple, this change introduces a behavioral change; method `B.m()` produces different results depending on the version (10 in Figure 4.2-a code and 20 in Figure 4.2-b).

```java
public class A {
  public int k (long i){
   return 10;
  }
  public int x (){
   return 5;
  }
  public int sum (){
   int xRes = x();
   return xRes + k(xRes);
  }
}
public class B extends A {
  public int k (int i){
   return 20;
  }
  public int m(){
   return new A().k(2);
  }
  public int a(){
   return 30;
  }
}
```

(a) Original code.

```java
public class A {
  public int k (long i){
   return 10;
  }
+  public int k (int i){
+   return 20;
+  }
  public int x (){
   return 5;
  }
  public int sum (){
   int xRes = x();
   return xRes + k(xRes);
  }
}
public class B extends A {
  public int m(){
   return new A().k(2);
  }
  public int a(){
   return 30;
  }
}
```

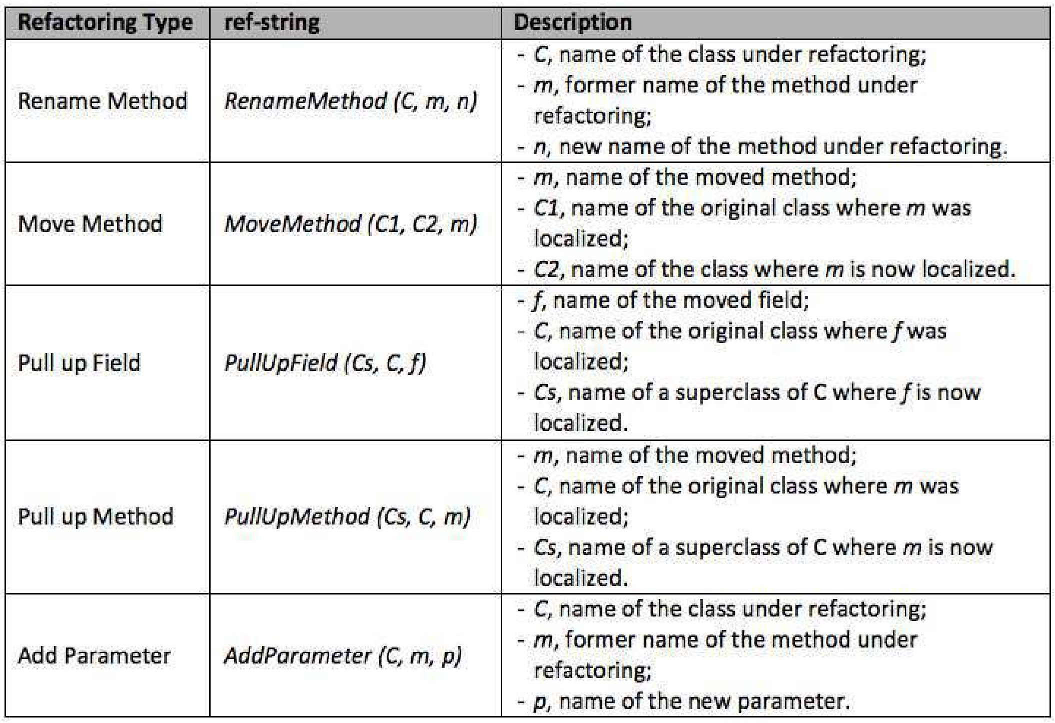
(b) Code after a problematic pull up method refactoring. There is a behavioral change in the *B.m()* method.

```java
public void test1(){
  p1.B b = new p1.B();
  int res = b.m();
  assertEquals (10, res);
}
public void test2(){
  p1.B b = new p1.B();
  int res = b.a();
  assertEquals (30, res);
}
public void test3(){
  p1.B b = new p1.B();
  int res = b.sum();
  assertEquals (15, res);
}
```

(c) JUnit test cases for the target version.

Figure 4.2: An example of a problematic refactoring edit.

Approach Depending on the type of refactoring, a different set of behavioral changes may be introduced to the program. RBA is directed to the refactorings applied. In the first activity (Discover Refactoring Edits, Figure 4.1), the edits applied between the *base* and *delta* versions are identified. Our implementation tool (PRIORJ) reuses a state-of-art refactoring detection tool, Ref-Finder [62]. Ref-Finder is a tool that uses a *template-based refactoring reconstruction* for identifying which refactoring edits were applied between two consecutive versions of a Java program. Although often producing acceptable success rates (precision of 0.79 and recall of 0.95), Ref-Finder may generate false positives [110]. Therefore, RBA can also work with manual refactoring identification methods (e.g., refactoring plans, pair



Refactoring Type	ref-string	Description
Rename Method	*RenameMethod (C, m, n)*	- *C*, name of the class under refactoring; - *m*, former name of the method under refactoring; - *n*, new name of the method under refactoring.
Move Method	*MoveMethod (C1, C2, m)*	- *m*, name of the moved method; - *C1*, name of the original class where *m* was localized; - *C2*, name of the class where *m* is now localized.
Pull up Field	*PullUpField (Cs, C, f)*	- *f*, name of the moved field; - *C*, name of the original class where *f* was localized; - *Cs*, name of a superclass of C where *f* is now localized.
Pull up Method	*PullUpMethod (Cs, C, m)*	- *m*, name of the moved method; - *C*, name of the original class where *m* was localized; - *Cs*, name of a superclass of C where *m* is now localized.
Add Parameter	*AddParameter (C, m, p)*	- *C*, name of the class under refactoring; - *m*, former name of the method under refactoring; - *p*, name of the new parameter.

Table 4.1: ref-strings specification of the five refactoring types supported by RBA.

review comparison [81]). For that, the output of the RBA's first activity is independent of refactoring detection method. The results from Ref-Finder, or any other method, are parsed into *ref-strings*. Each ref-string consists of a string pattern that describes a refactoring edit, its type and location. Their representation evokes a procedure signature; the procedure name represents the refactoring, and parameters are the elements directly involved in the specific edit. Ref-strings for the five refactoring types currently supported by RBA are depicted in Table 4.1 – the ref-string for the edit described in our guiding example is `PullUpMethod(B, A, k(int i))`. The output from this refactoring reconstruction process is a set of ref-strings.

After gathering the refactoring data, in its second activity (Discover Impacted Elements) RBA identifies the elements of the program source code that might be impacted, given an incorrect refactoring. This phase is key to the technique, as the prioritized suite's quality is highly related to the accuracy of the set of possible impacted elements. For that, we propose Refactoring Fault Models (RFM) –algorithms that employ a lightweight static analysis for extracting the method calls potentially affected by that particular refactoring. The RFM concept is detailed in Section 4.2. For each collected *ref-string*, the correspondent RFM is run, and an affected set (*AS*) is built. At the end of this activity the *AS* set contains the method calls from the base version, whose behavior might have been modified by the refactorings. Back to Figure 4.2, the application of the pull up method RFM results in `AS = {A.k(long i), A.sum(), B.k(int i), B.m()}`.

In the activity Generate Test Case Call Graphs, RBA creates a *call graph* [45] for each test case. Those graphs are used to determine the relationship of a test cases to methods from *AS*. Each node in a call graph represents a procedure and each edge `(f,g)` indicates that procedure `f` calls procedure `g`. Call graphs can be either dynamically or statically generated. However, statically generated call graphs may miss accurate information related to subtyping and dynamic dispatch. There are several tools that automatically generate call graphs for different languages, such as Java, Python and C (e.g., PriorJ [98], KCachegrind[1], phpCallGraph[2]). To work with a more thorough data, our implementation of RBA uses dynamic call graphs. Figure 4.2-c shows three JUnit test cases for the example in Figure 4.2-a, while Figure 4.3 shows their respective call graphs created after RBA's third activity. It is important to highlight that when using RBA in a real scenario, a developer would have to execute the whole test suite and generate its call graphs once. These graphs will be reused for future prioritizations. Modifications on a test case would impact on its respective call graph. However, if major modifications are performed in the system's code, its test cases might be highly impacted as well. In this scenario, there is need for major updates in the call graphs.

Next, RBA performs a call graph-based analysis (Select Refactoring-Impacted Test Cases) for selecting the test cases related to the refactoring edits. Although the main goal of RBA is prioritization, we opt to give to the developer/tester the option of working with a smaller set of test cases as well, in case prioritization is not needed. The resulting set encompasses test cases whose call graph contains at least one node that matches elements from *AS*. For example, as the call graphs of `test1` and `test3` include nodes from the *AS* (`A.k(int i)`, `B.m()`, and `B.sum()`), the resulting set is `{test1(), test3()}`. If, due to project constraints, having a complete suite is an issue, test cases absent from this resulting set may be removed from the suite, although, depending on the number of edits or

[1]http://kcachegrind.sourceforge.net/

[2]http://phpcallgraph.sourceforge.net/

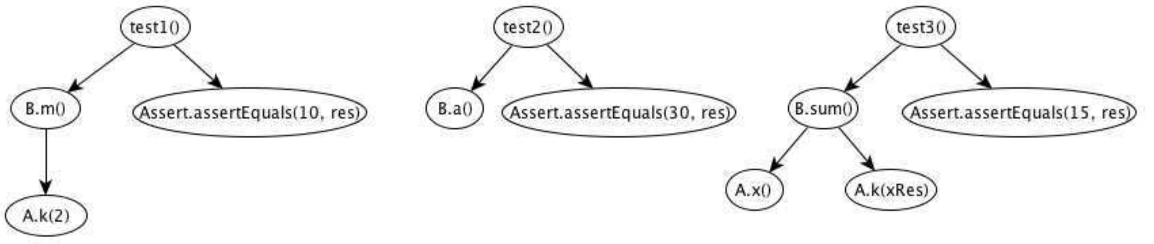

test1()
B.m()
Assert.assertEquals(10, res)
A.k(2)
test2()
B.a()
Assert.assertEquals(30, res)
test3()
B.sum()
Assert.assertEquals(15, res)
A.x()
A.k(xRes)

Figure 4.3: Call graphs of the test cases for the code under refactoring (Figure 4.2-c).

their overall impact, this set may be vast. In this case, prioritization is indispensable.

Prioritization is based on an impact value (*IVAL*) assigned to each test case in the suite during the activity Calculate Impact Values. *IVAL* corresponds to the number of elements from *AS* each test case covers. For instance, both `test1()` and `test3()` include two nodes related to elements from the affected set, so their impact value is the same ($IVAL_{test1} = 2$, and $IVAL_{test3} = 2$). As *test2* does not match any *AS* element, it is associated with an impact value of 0 ($IVAL_{test2} = 0$).

In its last activity (*Prioritize Test Cases*) a new order is proposed by using the calculated *IVALs* as criterion. Our prioritization heuristics assumes test cases that cover more elements from *AS* are more likely to reveal introduced behavioral changes. The test case with the highest *IVAL* is placed on the top of the prioritized test suite, then removed from the comparison for choosing the next test case. Ties with *IVAL* are dealt with simple randomization. Two possible prioritized suites from the example would be: *{test1, test3, test2}* or *{test3, test1, test2}*. Both reordered suites would detect behavioral change in its first test case.

4.2 Refactoring Fault Models – RFM

The refactoring fault models (RFM) extraction rules identify method calls from the SUT that might be affected by a specific refactoring – in particular, calls that are likely to present an altered behavior in the refactored program. The novelty of an approach based on RFMs is that individual characteristics of applied refactorings are taken into consideration for customizing test case prioritization.

The rules that compose the RFMs are related to the most common refactoring faults – a number of those common faults is related to subtle behavioral changes that usually pass unnoticed, even by well-trained developers (e.g., unnoticed overwritten or overloaded methods). We present the RFM in a pseudocode fashion (Sections 4.2.1, 4.2.2 and Appendix B). These RFMs were defined based on guidelines for applying refactorings edits in practice (e.g., Fowler's mechanics[33]), and initiatives for defining formal preconditions for sound refactorings (e.g., [88; 25; 76; 105; 106]).

We propose RFMs for five of the most common refactoring edits in Java systems [79] – rename method, move method, pull up field, pull up method and add parameter. For brevity, in the following subsections we present RFM for two representative refactorings types (rename method and pull up method), and describe them as pseudo-code algorithms. Each RFM builds up the Affected Set (*AS*). We illustrate those RFMs with examples of tricky refactoring problems reported by Soares et al. [109].

The following shows the meaning of the auxiliary functions used in the RFM pseudocode algorithms:

- `searchMethod(m, C)` returns method named `m` from class `C`;
- `searchMethodsWithSameName(m, C)` returns all methods that is or could be part of an overriding/overloading constraint with `m` from `C`;
- `searchMethodCalls(m, C)` returns all callers of method named `m` from `C`;
- `getSubClasses(C)` returns all sub classes of `C`;
- `getSuperClasses(C)` returns all super classes of `C`;
- `getAllClasses()` returns all classes from the current project.

The specification of the remaining three developed RFMs is available in Appendix B. In addition, to give a formal and less ambiguous definition, the specification of all RFM using Metamodeling and ATL rules (Atlas Transformation Language [56] is an OCL-based language) are available in [2].

4.2.1 Rename Method

The RFM for rename method can be represented as the signature `RenameMethod(C,oldName,newName)`, where `C` is the class whose method is to be renamed, `oldName` is the signature of the method that will be renamed, and `newName` is the new signature of the renamed method. The algorithm (with pseudocode listed in Figure 4.4-a) appends to the affected set –*AS*:

- the method formally named `oldName` from `C`;
- all methods in `C` ∪ `subtypes(C)` ∪ `supertypes(C)` whose bodies contain at least one call to `oldName` or `newName`. `subtypes(C)` (and `supertypes(C)` analogously) yields all subclasses that inherit from `C` directly or indirectly;
- if the method `oldName` in `C` is static, all methods, in any class, whose bodies contain a call to this method.

Consider the refactoring depicted in Figure 4.5 – method `n` from `B` is renamed to `k`, generating the target version depicted in Figure 4.5-b. In this scenario, even though a refactoring is desirable, a behavioral change occurred. Method `m` from `B` now returns 0, instead of 1 in the previous version. This change is due to a subtle method overriding that could easily pass without notice in a real complex system. For detecting such fault, a test case must cover calls to method *B.m()*, because through this method, *B.k()* can be invoked, and the modified behavior could be observed. Line 7 of the RFM's pseudocode selects this method as possibly impacted. Also, Lines 6, 11, 17 and 24 identify possible obsolete test cases accessing method *n*, which is absent in the target version.

4.2.2 Pull Up Method

As a RFM, the pull up method refactoring is represented by `PullUpMethod (Cs, C, mName)`, where `Cs` is the class where the method will be moved to, `C` is the original class, and `mName` is the signature of the method to be moved. The pseudocode for the pull up method RFM is listed in Figure 4.4-b – intuitively, *AS* is composed of:

- any method with a similar signature as `m` in `Cs` ∪ `subtypes(Cs)` (`C` ∈ `subtypes(Cs)`);
- all methods in `Cs` ∪ `subtypes(Cs)` whose bodies contain at least one call to the method to be moved;
- if the method to be moved in `C` is static, all methods whose bodies contain a call to this method.

```
RenameMethodRFM (C: class, oldName: String
    , newName: String)
  BEGIN
    AS <- Set<Method> {};
    oldMethod <- searchMethod (oldName, C)
    ;
    AS.add ( oldMethod );
     AS.addAll (searchMethodCalls (oldName
    , C) );
     AS.addAll (searchMethodCalls (newName
    , C) );

     subClasses <- getSubClasses (C);
     FOREACH S in subClasses DO
       AS.addAll (searchMethodCalls (
    oldName, S) );
       AS.addAll (searchMethodCalls (
    newName, S) );
     END FOREACH

     superClasses <- getSuperClasses (C);
     FOREACH Sp in subClasses DO
       AS.addAll (searchMethodCalls (
    oldName, Sp) );
       AS.addAll (searchMethodCalls (
    newName, Sp) );
     END FOREACH

     IF oldMethod is #static THEN
       allClasses <- getAllClasses();
       FOREACH C in allClasses DO
        AS.addAll (searchMethodCalls (
    oldName, C) );
       END FOREACH
     END IF
     return AS;
  END
```

(a) Rename method refactoring fault model.

```
PullUPMethodRFM (Cs: class, C: class,
    mName: String)
  BEGIN
    AS <- Set<Method> {};
     method <- searchMethod (mName, C);
     AS.add ( method );
     method2 <- searchMethod (mName, Cs);
     AS.add ( method2 );
     AS.addAll(searchMethodsWithSameName (
    mName, C));
     AS.addAll(searchMethodsWithSameName (
    mName, Cs));
     AS.addAll (searchMethodsCall (mName,
    C) );
     AS.addAll (searchMethodsCall (mName,
    Cs) );

     subClasses <- getSubClasses (Cs);
     FOREACH S in subClasses DO
       AS.addAll (searchMethodsCall (mName
    , S) );
       AS.addAll(searchMethodsWithSameName
     (mName, S));
     END FOREACH

     IF method is #static THEN
       allClasses <- getAllClasses();
       FOREACH C in allClasses DO
         AS.addAll (searchMethodsCall (
    mName, C) );
       END FOREACH
     END IF
     return AS;
  END
```

(b) Pull up method refactoring fault model.

Figure 4.4: Refactoring fault models algorithms.

In Figure 4.6, method `B.test()`, after the edit, has the invocation resolved to `B.k()`, differently from the source version, in which the call to `k()` is resolved to `A.k()` The RFM selects, among others, the `B.test()` method as possibly impacted. Then, test cases that

```
package p1;
public class A{
  public long k(long a){
     return 1;
  }
}
package p2;
import p1.*;
public class B extends A{
  protected long n(int a){
     return 0;
  }
  public long m(){
     return k(2);
  }
}
```

(a) Original code.

```
package p1;
public class A{
  public long k(long a){
     return 1;
  }
}
package p2;
import p1.*;
public class B extends A{
-   protected long n(int a){
+   protected long k(int a){
     return 0;
  }
  public long m(){
     return k(2);
  }
}
```

(b) Code after a problematic rename method. There is a behavior change when running the *B.m()* method in the target code.

Figure 4.5: Example of a problematic pull up method.

cover this method are more likely to unveil this fault when run against the target version.

4.2.3 Limitations of RFMs

The aim of our proposal is to anticipate the detection of faults when using regression testing; we assume that such verification technique places emphasis on practical constraints over completeness of refactoring fault detection. Therefore, RFMs do not intend to perform a complete change impact analysis; more elaborate static analysis, or even fusing some dynamic analysis, could improve precision of impact determination (e.g., extending the *AS* set), but the cost-benefit ratio is yet to be assessed. Also, our RFM cannot guarantee the detection of all types of behavioral changes; we do not expect RFMs to cover any possible error in applying a particular refactoring. RFMs are proposed as commonly impacted locations, based on established — theoretically-based and/or practice-oriented – literature on refactoring. As such, the proposed approach covers a considerable ground for common refactoring faults. What contributes to this decision is the complexity of the effort to anticipate any possible semantic change within a general-purpose object-oriented language like Java. In fact, the-

```
public class A{
  public int k(){
     return 10;
  }
}
public class B extends A{
  public int test(){
     return k();
  }
}
public class C extends B{
  public int k(){
     return 20;
  }
}
```

(a) Original code.

```
public class A{
  public int k(){
     return 10;
  }
}
public class B extends A{
  public int test(){
     return k();
  }
+  public int k(){
+     return 20;
+  }
}
public class C extends B{
-  public int k(){
-     return 20;
-  }
}
```

(b) Code after a problematic pull up method. There is a behavior change when running the *B.test()* method in the target code.

Figure 4.6: Example of a problematic rename method.

oretical research on defining such completeness property always consider a confined core language [25; 76; 82]. A more comprehensive RFM might lead to excessively large sets of affected methods, possibly, in consequence, decreasing the quality of test case selection and prioritization.

4.3 Evaluation

Assuming test suites that detect refactoring faults, we perform two empirical studies to investigate RBA's effectiveness regarding early detection of behavioral changes : an exploratory study, in which common behavioral changes are seeded into a real Java open-source project; and a set of controlled experiments where subtle behavioral changes, collected from related research studies, are detected with automatic generated test suites. In both studies, we measure how effective RBA is in placing fault-revealing test cases on top positions. Additionally, we also investigate whether the prioritization empowered by RBA is capable of grouping behavioral change-revealing test cases.

4.3.1 Exploratory Study

In this study, we compare RBA to other prioritization techniques in the context of a real Java project.

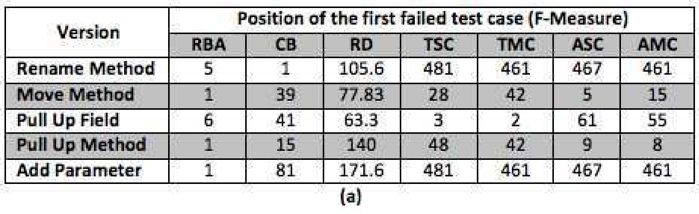
Goal The goal of this study is to observe RBA's prioritization effectiveness when dealing with behavioral changes for a given refactoring edit from the point of view of a tester.

Question and Metrics To guide our investigation, we establish the following research question: *What is the position of the first fault-revealing test case in the ordering defined by each technique?* The metrics chosen to address this question are: i) *F-Measure* [55], number of distinct test cases needed to be run in order to detect the first program failure; and ii) *APFD* [99], that reflects the effectiveness of a test case ordering (Equation 4.3.1 – ranges from 0 to 1. Higher APFD values imply faster fault detection rates).

$$APFD = 1 - \frac{TF_1 + TF_2 + ... + TF_m}{nm} + \frac{1}{2n} \tag{4.2}$$

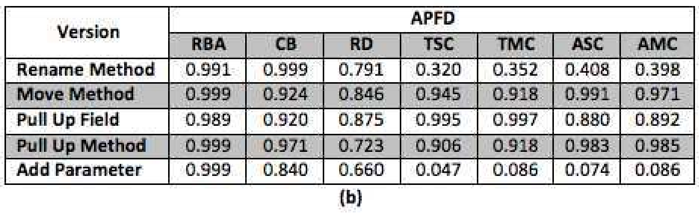
where *n* is the number of test cases, *m* is the number of exposed faults. TF_i is the position of the first test case which reveals the fault *i* in the ordered test cases sequence.

Planning and Design In this study we use the JMock project (described in Section 1.1). We create five faulty versions of JMock's code each one with a single and a distinct refactoring related behavioral change. To each version, a single refactoring fault is seeded. For seeding the faults we follow a similar process as described in Section 1.1, in which a single step from Fowler's mechanics [33] is neglected. For performing the prioritizations, we collect JMock's manually created test suite (445 JUnit test cases in the system integration level). Then, the suite is prioritized using seven prioritization techniques from different categories: i) RBA; ii) four coverage-based [99] (Total Statement Coverage - TSC, Total Method Coverage - TMC, Additional Statement Coverage - ASC, Additional Method Coverage - AMC); iii) a random-based (RD); iv) a modification-based [114] (Change Blocks - CB). Test cases that present compilation errors after the code edits are deleted before prioritization.



Version	Position of the first failed test case (F-Measure)						
	RBA	CB	RD	TSC	TMC	ASC	AMC
Rename Method	5	1	105.6	481	461	467	461
Move Method	1	39	77.83	28	42	5	15
Pull Up Field	6	41	63.3	3	2	61	55
Pull Up Method	1	15	140	48	42	9	8
Add Parameter	1	81	171.6	481	461	467	461

(a)



Version	APFD						
	RBA	CB	RD	TSC	TMC	ASC	AMC
Rename Method	0.991	0.999	0.791	0.320	0.352	0.408	0.398
Move Method	0.999	0.924	0.846	0.945	0.918	0.991	0.971
Pull Up Field	0.989	0.920	0.875	0.995	0.997	0.880	0.892
Pull Up Method	0.999	0.971	0.723	0.906	0.918	0.983	0.985
Add Parameter	0.999	0.840	0.660	0.047	0.086	0.074	0.086

(b)

Table 4.2: Case study results. (a) F-Measure values; (b) APFD values.

Data Analysis and Discussion Tables 4.2-a and 4.2-b show the results of this case study. Those results show that, in most cases, RBA's prioritized suites detect the faults after running just a single test case (F-Measure = 1), which is the best scenario for a prioritization technique. Another aspect to consider is the high stability of RBA when compared to the other techniques. All RBA's APFD results vary in a very tight range [0.989; 0.999], i.e. RBA's orderings detect all behavioral changes very early and in similar positions. This is more evident when we observe the standard deviation of APFD values ($\alpha_{TSC} = 0.431$; $\alpha_{TMC} = 0.409$; $\alpha_{ASC} = 0.408$; $\alpha_{AMC} = 0.404$; $\alpha_{CB} = 0.06$; $\alpha_{RD} = 0.088$; $\alpha_{RBA} = 0.004$). RBA's standard deviation is much lower. Those results give us evidences about the effectiveness of RBA when detecting refactoring faults.

In two cases RBA is outperformed by other techniques, pull up field - TSC/TMC and rename method- CB. In the first case, the behavioral change-revealing test case has a high coverage which ended up favoring the *Total* strategies. In the second case, the CB prioritization strategy is better performs better due to the fact that the introduced rename method behavioral change, by chance, can be detected by tests that directly cover the changed parts of the code, which is the prioritization heuristic applied by CB. But, we can see that even when RBA does not produce the best results, RBA's results are quite close to the best ones.

4.3.2 Study with Subtle Faults

In the second investigation, we perform a set of controlled studies to analyze prioritization techniques when dealing with subtle refactoring faults and generated test suites.

Goal The goal of this investigation is to compare RBA with a set of well-known prioritization techniques on how they speed up the detection of refactoring faults.

Questions and Metrics This study is conducted based on two study questions:
SQ1: *Does RBA detect refactoring faults earlier?*
SQ2: *Does RBA places the behavioral change-revealing test cases in scattered positions?*

For addressing the first question we use the F-Measure metric. Although APFD is the most important metric for evaluating prioritized suites, we opt not to use it in our experiments. As each study deals with a single refactoring edit and a single behavioral change at time, for this configuration, the APFD results would only reflect a relative magnitude of the F-Measure values.

Besides the F-Measure, as we believe that a good prioritization technique would not only place fault-revealing test cases in a suite top positions, but also group them close to each other, we define a new metric, *F-spreading* (Equation 4.3.2) for addressing the second study question. F-spreading measures how spread out the failed test cases are in a suite.

$$F - spreading = (\sum_{i=2}^{m} TF_i - TF_{i-1}) * \frac{1}{N} \tag{4.3}$$

where *N* is the number of test cases; *m* is the number of failed test cases; *TF* is an ordered set containing the positions of the test cases that failed due a behavioral change; and TF_i is the *i*º failed test case in the prioritized suite.

Even when a single behavioral change is introduced, several test cases may fail. But, not always a single test case provides enough information for helping fault localization (debugging). Thus, if the failed test cases are less scattered, its easier to analyze and find relations among them. As higher the F-spreading is, more scattered the behavioral revealing test cases are. Thus, a good prioritization technique generates prioritized suites with low F-measure and F-spreading. For instance, consider two prioritization techniques *T1* and *T2*, and a test suite *S* with 200 test cases in which 5 of them failed due to a refactoring fault. Suppose that, after prioritizing *S* with *T1* and *T2*, we have suites where the failed test cases are placed in the following positions S_{P1}:{1, 30, 40, 75, 100}, and S_{P2}:{1, 10, 11, 15, 30}. The F-spreading value for S_{T1} and S_{T2} are 0.495 and 0.145, respectively. Although both suites are able to detect the refactoring fault early ($F - Measure_{S_{P1}}$ = 1 and $F - Measure_{S_{P2}}$= 1), *T2* rescheduled the failed test cases in a less spread out way, when compared to *T1*. Thus, by using *T2* for this context the tester are likely to have better understanding of the problem which facilitate the fault localization and/or fix.

Planning and Design Due to the difficulty of finding available real systems in which refactoring behavioral changes could be localized through failed test cases (its a common police to do not commit code with failed test cases), and that the current fault-injector strategies (e.g., mutant operators) do not allow us to control the type of faults, we build a data set with 26 refactoring transformations that contain subtle refactoring faults. Those transformations are reported in the literature ([109]) and, in general, cover unexpected introduction/breaking of overriding/overloading constraints during a refactoring. This fault has been reported by Bavota et al. [13] as commonly found in projects after refactoring edits. All transformations are free of compilation errors and depict situations that not even well-known refactoring tools (e.g., Eclipse, NetBeans, JRRT[3] - JastAdd Refactoring Tools) are able to predict. Figure 4.7 shows an example of one of a pull up method transformation used as subjects in our experiments. Method `k (int i)` is pulled up from class `B` to `A`), generating an unexpected behavioral change. Method `B.test()` returns a different result depending on the version used (10, considering the source version, and 20 for the target version). This kind of behavioral change is usually hard to identify through visual inspection and was not anticipated by the JRRT v1 refactoring tool.

To consistently evaluate the prioritization order produced by each technique, for each code transformation, we automatically generate a test suite by using the Randoop tool[4] [89]. Randoop is an automatic unit test generator for Java that generates unit tests using feedback-directed random test generation. This tool can build regression test suites from scratch and has been used in several works (e.g. [111; 109; 51]). A similar strategy is used by SafeRefactor[111], a tool for validating refactorings. However, as the size of the code transformations is quite small (Figure 4.7), and enable the generation of more diversified regression suites, we add a set of extra Java methods to each code transformation. Each extra method is completely independent, not interfering with the execution neither of the original

[3]http://jastadd.org/web/jastaddj/refactoring.php

[4]https://code.google.com/p/randoop/

```
public class A {
     public int k (long i){
          return 10;
     }
}
public class B extends A {
     public int k (int i){
          return 20;
     }
     public int test(){
          return new A().k(2);
     }
}
```

(a) Original code.

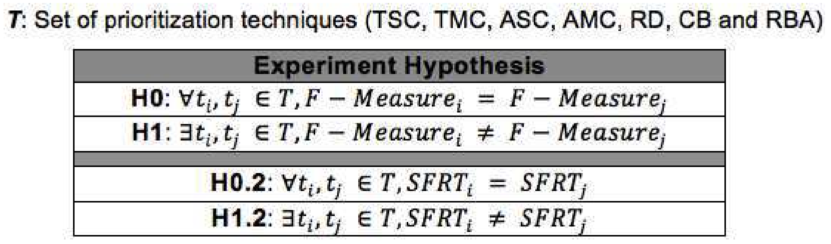
```
public class A {
     public int k (long i){
          return 10;
     }
+     public int k (int i){
+          return 20;
+     }
}
public class B extends A {
-     public int k (int i){
-          return 20;
-     }
     public int test(){
          return new A().k(2);
     }
}
```

(b) Code after a problematic pull up method refactoring. When called, B.test() returns 20 instead of 10 as in the previous version.

Figure 4.7: One of the pull up method transformations used as experiment subjects.

code example methods, nor to any other extra method. The extra methods are available in [2]. On average, for each of the 26 subjects, Randoop generated suites with 3,568 JUnit test cases, with a time limit of 100 seconds, and maximum test size of 5. Although larger time limits would allow Randoop to occasionally generate new test cases that might detect faults, studies [111; 108] have shown that there is little gain regarding testing power when considering test suites generated using big time limits. After a while, Randoop's test generation algorithm exhausts most of its possibilities and starts generating new test cases that are quite similar (in terms of path coverage and data) to the ones already generated. This is even more evident in small-size subjects such as the ones that compose our data set.

To conduct the statistical validation of our experimentation, we postulate two pairs of statistical hypotheses, null and alternative (Table 4.3). The null hypotheses (H0 and H0.2) state that all prioritization techniques under investigation behave similarly, with respect to F-Measure and F-spreading. The alternative hypotheses (H1 and H1.2) state that they are not all similar. We apply the *One-factor-and-several-treatments* experimental design [53] in each experiment of our study (an experiment for each type of refactoring), where the factor is the prioritization technique and the treatments are the seven techniques under investigation



T: Set of prioritization techniques (TSC, TMC, ASC, AMC, RD, CB and RBA)

Experiment Hypothesis
H0: $\forall t_i, t_j \in T, F-Measure_i = F-Measure_j$
H1: $\exists t_i, t_j \in T, F-Measure_i \neq F-Measure_j$
H0.2: $\forall t_i, t_j \in T, SFRT_i = SFRT_j$
H1.2: $\exists t_i, t_j \in T, SFRT_i \neq SFRT_j$

Table 4.3: Experiments Hypothesis.

(TSC, TMC, ASC, AMC, RD, CB, and RBA).

A pilot study was performed to define the statistically required number of replications for each configuration in our experiments. For each configuration (Java transformation x prioritization technique x metric), the number of replications varied from 500 to 1,892, for a precision (*r*) of 2% of the sample mean and significance (α) of 5%.

Operation All prioritization executions are performed using the PriorJ tool (details in Chapter 6). PriorJ is an open-source tool that supports prioritization activities execution for Java/JUnit systems. Additionally, a set of Java classes were developed for translating PriorJ's output artifacts and calculating the needed metrics (*F-Measure* and *F-spreading*). During the execution of this study PriorJ was run using Java 7 in a MacBook Pro Core i5 2.4GHz and 4GB RAM, running Mac OS 10.8.4.

Data Analysis and Discussion First, we perform a normality test with a confidence level of 95% ($\alpha = 0.05$) for each of experimental configurations (26 Java transformations x 7 prioritization techniques). The variability of the data comes from the aleatory aspect of each prioritization algorithm. As the p-values from those tests are all smaller than the significance value, we can say that our samples do not follow the normal distribution and, consequently, a non-parametric test should be used. Since each experimental design has a unique factor with more than two treatments, we applied the *Kruskal-Wallis* test [121]. This test is used to determine whether there are significant differences among the population means. Again, for all cases, the p-values are smaller than the significance level ($\alpha < 0.05$). Thus, all null hypotheses can be rejected, i.e., *the prioritization techniques do not have identical behavior neither considering the F-Measure nor the F-spreading, with 95% confidence level.*

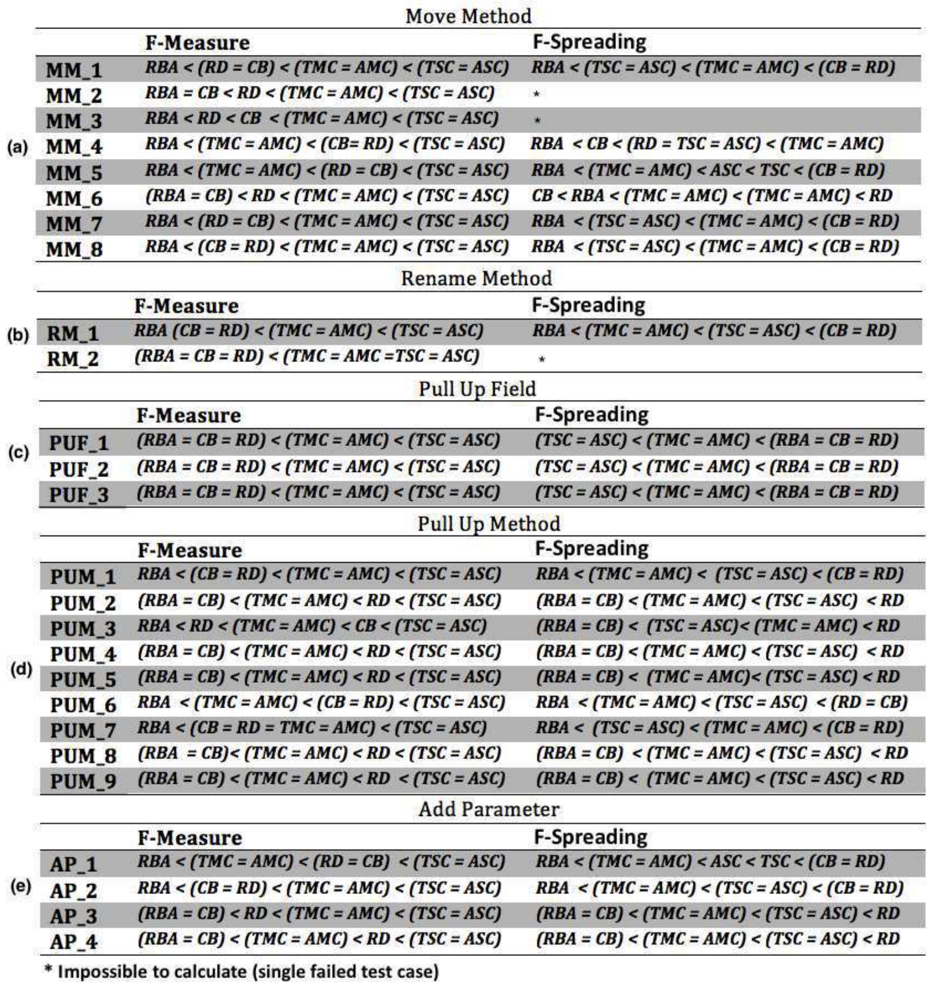
Continuing the data analysis, we plot confidence intervals for each groups of F-Measure and F-spreading results. When we find overlappings, we apply the Mann-Whitney test [121]. The Mann-Whitney test allows us to look to samples in pairs to be able to rank the techniques. After analyzing the position of the confidence intervals and the Mann-Whitney results we rank the techniques with respect to their behavior (Table 4.4). The whole data regard this statistical analysis (normality tests, Kruskal-Wallis tests, and Mann-Whitney tests) are available in [2].

Table 4.4 summarizes the results of our statistical analysis. Each row shows a comparison among the technique results, in which it goes from better to worse results from left to right. For instance, considering only the code transformation related to the first move method behavioral change (MM_1) and the F-Measure analysis, RBA is the technique that has the best performance, i.e. it is the technique that detects this behavioral change earlier. The second best technique for this scenario is either RD or CB, they have statistically identical performances. Then TMC or AMC. Finally, TSC and ASC have the worst performance when detecting this fault.

Our analysis shows RBA's ability on helping the early detection of refactoring faults related to behavioral changes. With respect to fault detection effectiveness (F-Measure results - Table 4.4), we can see that RBA's performance is better than, or at least similar to, the other techniques for every configuration and every refactoring type. Even the change-based technique (CB) does not behave well in several situations (e.g., *MM_4*, *MM_5*), which does not happen to RBA. These facts help us to answer the first study question (SQ1) and conclude that, in the context of our experiments, *RBA is be a better choice for prioritizing test cases aiming at speeding up the detection of refactoring related behavioral changes.*

Regards the scattering of failed test cases, we observe the F-spreading results (Table 4.4, right-hand side). For most of the cases, RBA has the best performance with respect to placing the failed test cases in close positions. Thus, we can answer our second study question (SQ2) by stating that RBA often places behavioral change revealing test cases near to each other. This fact may give to the tester/developer more information earlier to help him during the fault detection/debugging task.

Looking closely to the cases where RBA is outperformed (MM_6, PUF_1, PUF_2, PUF_3), we observe that the selected possible impacted methods are in fact not affected. It happened because some rules from the RFMs apply a name-based investigation, which



Move Method

(a)		F-Measure	F-Spreading
	MM_1	RBA < (RD = CB) < (TMC = AMC) < (TSC = ASC)	RBA < (TSC = ASC) < (TMC = AMC) < (CB = RD)
	MM_2	RBA = CB < RD < (TMC = AMC) < (TSC = ASC)	*
	MM_3	RBA < RD < CB < (TMC = AMC) < (TSC = ASC)	*
	MM_4	RBA < (TMC = AMC) < (CB= RD) < (TSC = ASC)	RBA < CB < (RD = TSC = ASC) < (TMC = AMC)
	MM_5	RBA < (TMC = AMC) < (RD = CB) < (TSC = ASC)	RBA < (TMC = AMC) < ASC < TSC < (CB = RD)
	MM_6	(RBA = CB) < RD < (TMC = AMC) < (TSC = ASC)	CB < RBA < (TMC = AMC) < (TMC = AMC) < RD
	MM_7	RBA < (RD = CB) < (TMC = AMC) < (TSC = ASC)	RBA < (TSC = ASC) < (TMC = AMC) < (CB = RD)
	MM_8	RBA < (CB = RD) < (TMC = AMC) < (TSC = ASC)	RBA < (TSC = ASC) < (TMC = AMC) < (CB = RD)

Rename Method

(b)		F-Measure	F-Spreading
	RM_1	RBA (CB = RD) < (TMC = AMC) < (TSC = ASC)	RBA < (TMC = AMC) < (TSC = ASC) < (CB = RD)
	RM_2	(RBA = CB = RD) < (TMC = AMC =TSC = ASC)	*

Pull Up Field

(c)		F-Measure	F-Spreading
	PUF_1	(RBA = CB = RD) < (TMC = AMC) < (TSC = ASC)	(TSC = ASC) < (TMC = AMC) < (RBA = CB = RD)
	PUF_2	(RBA = CB = RD) < (TMC = AMC) < (TSC = ASC)	(TSC = ASC) < (TMC = AMC) < (RBA = CB = RD)
	PUF_3	(RBA = CB = RD) < (TMC = AMC) < (TSC = ASC)	(TSC = ASC) < (TMC = AMC) < (RBA = CB = RD)

Pull Up Method

(d)		F-Measure	F-Spreading
	PUM_1	RBA < (CB = RD) < (TMC = AMC) < (TSC = ASC)	RBA < (TMC = AMC) < (TSC = ASC) < (CB = RD)
	PUM_2	(RBA = CB) < (TMC = AMC) < RD < (TSC = ASC)	(RBA = CB) < (TMC = AMC) < (TSC = ASC) < RD
	PUM_3	RBA < RD < (TMC = AMC) < CB < (TSC = ASC)	(RBA = CB) < (TSC = ASC)< (TMC = AMC) < RD
	PUM_4	(RBA = CB) < (TMC = AMC) < RD < (TSC = ASC)	(RBA = CB) < (TMC = AMC) < (TSC = ASC) < RD
	PUM_5	(RBA = CB) < (TMC = AMC) < RD < (TSC = ASC)	(RBA = CB) < (TMC = AMC)< (TSC = ASC) < RD
	PUM_6	RBA < (TMC = AMC) < (CB = RD) < (TSC = ASC)	RBA < (TMC = AMC) < (TSC = ASC) < (RD = CB)
	PUM_7	RBA < (CB = RD = TMC = AMC) < (TSC = ASC)	RBA < (TSC = ASC) < (TMC = AMC) < (CB = RD)
	PUM_8	(RBA = CB)< (TMC = AMC) < RD < (TSC = ASC)	(RBA = CB) < (TMC = AMC) < (TSC = ASC) < RD
	PUM_9	(RBA = CB) < (TMC = AMC) < RD < (TSC = ASC)	(RBA = CB) < (TMC = AMC) < (TSC = ASC) < RD

Add Parameter

(e)		F-Measure	F-Spreading
	AP_1	RBA < (TMC = AMC) < (RD = CB) < (TSC = ASC)	RBA < (TMC = AMC) < ASC < TSC < (CB = RD)
	AP_2	RBA < (CB = RD) < (TMC = AMC) < (TSC = ASC)	RBA < (TMC = AMC) < (TSC = ASC) < (CB = RD)
	AP_3	(RBA = CB) < RD < (TMC = AMC) < (TSC = ASC)	(RBA = CB) < (TMC = AMC) < (TSC = ASC) < RD
	AP_4	(RBA = CB) < (TMC = AMC) < RD < (TSC = ASC)	(RBA = CB) < (TMC = AMC) < (TSC = ASC) < RD

* Impossible to calculate (single failed test case)

Table 4.4: Behavior evaluation of prioritization techniques.

shown not to be efficient for those cases. More specifically, when a refactoring involves variable manipulation (e.g., pull up field), this name analysis can be confused by variable names from different scopes. Some of the extra methods had internal variables with the same name of the fields manipulated. Thus, for those cases the name-based analysis failed on selecting only the test cases related to the changes. When we rename those problematic variables in a investigation-scenario we observe a great improvement of the F-spreading results. For instance, PUF_1 's F-spreading went from 0.95 to 0.004. This fact enable us to conjecture that by combining the current fault model rules with a variable scope differentiation and/or a binding checking we can improve even more the quality of our prioritization process.

Numerical Analysis The statistical tests discussed above depict the general behavior of RBA. They showed that statistically RBA is the better option for detecting refactoring faults earlier, when compared to the traditional techniques. Taking into consideration all configurations run during our empirical studies we can numerically visualize this benefit. In our study each prioritization technique was run 3,545 times, generating 3,545 different prioritized suites. After analyzing the values collected through the metrics (F-Measure and F-spreading) we observe that in 71% of the cases RBA generated better results than the traditional techniques w.r.t. F-Measure. Similar results are found when we consider the F-spreading metric, RBA outperformed the other techniques in 73% of the cases. The spreadsheets containing all results from this study are available in [2].

Final Remarks An important aspect to highlight regarding our investigation is that we dealt with two complex factors: i) *very subtle refactoring faults*, faults that even the most well-known refactoring tools (Eclipse, NetBeans, JRRT) are not able to identify; and ii) *large regression test suites*, our suites had an average of 3,568.6 test cases. These factors tend to turn the prioritization process even harder. Even though dealing with those difficulties, RBA produced very good and stable results.

With the combination of an early fault localization (low F-Measures) and low scattering of failed test cases (low F-spreading) we show the contribution that RBA brings to the state-of-art of test case prioritization when dealing with refactoring problems. Thus, we can answer the second research question of our doctorate work (**RQ2**) by stating that indeed is possible to anticipate the detection of refactoring faults by using relating testing coverage and commonly impacted locations.

4.3.3 Threats to Validity

In terms of **conclusion validity**, we discuss about the statistical test used during in our evaluation. To achieve significant conclusions, the number of replications for each experiment was decided according to statistical guidelines. Also, the analyses of the results were performed considering a high confidence level (95%). Finally, we choose the statistical tests based on the type of data collected from our data set (e.g., according to normality tests).

In terms of *internal validity*, a factor that may influence our results is the existence of potential faults in our tool support. The PRIORJ project has a set of unit testing that confirms its accuracy regarding prioritization. Additionally, for controlling this threat, we additionally validated this tool through testing on several examples of JUnit test suites and Java programs.

In terms of **construction validity**, the dependent measures that we have considered, F-Measure and F-spreading, are not the only possible measure for evaluating prioritization effectiveness or spreadness of refactoring-revealing test cases. However, according to the purpose of our investigation, and to configuration of our experiments, we believe that those measures are quite appropriated. Future studies will consider different measures.

In terms of **external validity**, some aspects can be defined as limiting for the generalization of our results: i) the subjects. Due to the fact that the number of subjects are small in some cases (e.g., two code examples for the rename method edit) we cannot say that our subjects represent the whole universe of Java programs and refactoring faults. But, as those transformatios were collected from independent works and reflect tricky faults that not even the most used refactoring tools were able to identify, we believe that they are suitable for the purpose of our investigation based on the assumption that if a prioritization technique is able to accelerate the detection of those hard to find faults it is likely that a similar result will be achieved when dealing with easier ones; and ii) test suite representativeness. We decided to use only automatically generated random regression suites in our second study. Though this practice is not always used in real projects, random testing has been used for a long time and random unit tests has been a great alternative due to the available tool support (e.g., Randoop). Random suites has been used for validating refactorings [108]. Moreover the results of our first study with a real project and a test suite manually created, confirm our conclusions.

4.4 Concluding Remarks

This chapter presents the Refactoring-Based Approach (RBA) for prioritizing test cases. This approach aims to speed up the detection of mistakenly introduced behavioral changes after refactoring. RBA's evaluation studies show statistical evidences that RBA speeds up the detection of refactoring problems when compared to other prioritization techniques, *i.e.*, it places fault revealing test cases in earlier positions of prioritized suites. Moreover, as RBA tends to better group the refactoring fault revealing test cases. Thus, it may give to the developer/tester more information related to those faults earlier, which can be very helpful for debugging.

Despite the fact that, along with coverage-based techniques, RBA demands the complete test suite to be executed at least once for generating the call-graphs, one may argue on what are the actual benefits of test case prioritization when compared to the demanded effort. Firstly, during an edit validation, a test suite may be run several times until no failures are experienced. Secondly, the order proposed by RBA may more effectively group test cases that fail, potentially saving testing/debugging time. Finally, test cases that become obsolete can be identified and possibly discarded.

It is important to highlight that our prioritization solution does not affect a suite's testing potential as no test case is discarded, and the number of test cases to execute is flexibility decided according to the project resources availability. However, not rarely test suites fail to provide an efficient support for validating refactorings. In the next chapter, we propose an static analysis approach that aims to complement the refactoring validation with testing, and to help refactoring reviewing.

Chapter 5

RefDistiller

Although, most developers use regression test suites for validating refactorings, the quality of those suites are highly related to developer's expertise and efforts on building high quality tests. Rachatasumrit and Kim find that regression test suites can be inadequate for covering refactored locations [94]. Moreover, testing results are not always easy to understand. A failing test case may not provide useful information to facilitate the process of locating and/or fixing a found fault.

As refactoring requires developers to coordinate related transformations and understand the complex inter-relationship between affected types, methods, and variables, often, there is a need of reviewing those edits. In this chapter we present REFDISTILLER [11], a refactoring-aware code review approach and tool (Chapter 6) that uses static analysis elements to help developers detect potential behavioral changes (anomalies) in manual refactoring edits. REFDISTILLER detects two types of refactoring anomalies that are important to observe when reviewing a refactoring, *missing* and *extra edits*. Missing edits are required steps that, when missed, often alters the behavior of a stable code. Extra edits are edits that deviates from a pure refactoring transformation. Either anomalies may not be in fact behavior changes, however, since developers often interleave refactorings with extra changes [81], it is important to be aware of them, specially when reviewing refactorings correctness.

Besides automatically identifying potential behavioral changes that could pass unnoticed when refactoring manually, REFDISTILLER innovates by helping developers to localize and fix potential refactoring anomalies by outputting information related to the type of anomaly, its location, and possible clues for the anomaly. Thus, REFDISTILLER can complement current testing-based refactoring validations in two ways: (1) it can identify problems that the test suite may fall short on covering and (2) it can help improve the quality of the developer's test suite by identifying new scenarios that require testing. Finally, REFDISTILLER provides a refactoring separation analysis that can be used during peer code review, highlighting which parts of the code deviate from a pure refactoring version.

REFDISTILLER's Equivalence Definition REFDISTILLER's equivalence notion establishes two scenarios when there is behavior preservation after a refactoring: i) when there is no missing edit: all required constituent steps are find among the manually performed edits; and ii) when there is no extra edits: there is no additional edit in relation to the correct edits performed by an automated refactoring tool. The following definition summarizes REFDISTILLER's equivalence representation:

Being:

- R_e, set of required constituent code editions related to a certain refactoring;
- M_e, set of manually performed code editions after a certain refactoring;
- A_e, set of code editions performed by an automatic refactoring after a certain refactoring;

$$A_e \equiv M_e \land \forall e \in R_e, \exists e' \in M_e | e = e' \tag{5.1}$$

5.1 The RefDistiller Approach

Figure 5.1 gives an overview of our approach. REFDISTILLER detects two types of refactoring anomalies: *missing* and *extra edits*. REFDISTILLER consists of two independent phases: REFCHECKER and REFSEPARATOR. REFCHECKER detects likely omissions of expected refactoring steps; and REFSEPARATOR tries to detect unexpected additional changes made during refactoring. REFDISTILLER takes as input the original version P and the manual refactoring version P_r, and uses RefFinder [62], or any manual refactoring detecting strategy, to infer

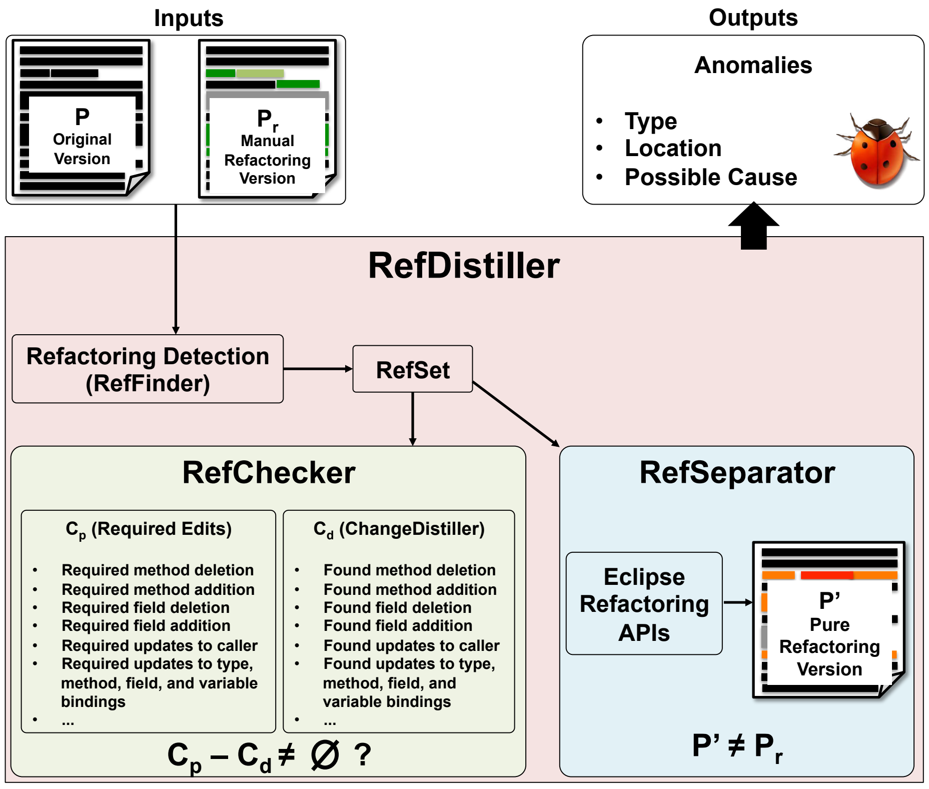

Inputs
P
Original Version
P_r
Manual Refactoring Version
Outputs
Anomalies
• Type
• Location
• Possible Cause
RefDistiller
Refactoring Detection (RefFinder)
RefSet
RefChecker
C_p (Required Edits)
• Required method deletion
• Required method addition
• Required field deletion
• Required field addition
• Required updates to caller
• Required updates to type, method, field, and variable bindings
• ...
C_d (ChangeDistiller)
• Found method deletion
• Found method addition
• Found field deletion
• Found field addition
• Found updates to caller
• Found updates to type, method, field, and variable bindings
• ...
C_p – C_d ≠ Ø ?
RefSeparator
Eclipse Refactoring APIs
P'
Pure Refactoring Version
P' ≠ P_r

Figure 5.1: RefDistiller overview.

the types and locations of potential refactoring edits, which we call *RefSet*. Using this information, REFDISTILLER detects anomalies, *i.e.*, potential errors that might deviate from a behavior preserving refactoring.

5.1.1 Detecting Missing Edits with RefChecker

For each refactoring type, we define template rules to describe required constituent edits. The rules are based on the literature on refactoring definition and formal specifications [33; 104; 88; 25], on the literature on common refactoring faults [13], and on our experience on testing and applying refactorings. For this research, we target the following six refactoring types, as they are some of the most commonly used refactoring types in Java [79]: extract method, inline method, move method, pull up method, push down method, and rename method. Each verifiable refactoring type requires a proper refactoring template. For each detected refactoring edit, REFCHECKER compares a set of expected edits C_p with actual manual edits C_d, generated by ChangeDistiller [30] that extracts source code changes based on tree differencing. ChangeDistiller reports AST change types such as adding a new method, deleting a method, inserting a new statement in a method body, deleting a statement from a method body, updating a statement, updating a method visibility, etc.

For each refactoring type in RefSet, REFCHECKER updates the set of required constituent edits and method and field reference updates, C_p, with the constituents edits based on their predefined template rules. A new required edit is a triple `<Type, Element, Location>` where `Location` represents the specific code element where the edit is expected (e.g., class, method, and field), `Element` represents the code element involved in the edit, and `Type` indicates the type of expected edit.

In order to preserve behavior, it is important to check whether method calls, field accesses, and type declarations of variables are preserved before and after manual refactoring. Therefore, we create a new checking type `Binding Problem` that is not currently supported by ChangeDistiller. REFCHECKER includes a binding object to C_p every time a reference, such as a method call, a variable access, and a field access in the original code differs from its correspondent in the refactored code. Those references are statically associated with the respective class and object scope. Our binding checking strategy compares AST trees in order to verify the preservation of desired references. REFCHECKER verifies if the associations remain the same for modified methods, their callers, and their direct callees in the new version. Similarly, it checks all refactored fields and the field accessors.

REFCHECKER reports the following nine types of warning messages for missing edits:

- *Binding problem:* Detected problematic reference binding X which may have affected method Y;
- *Missing method:* Method X was not found in Class Y;
- *Missing statement update:* There is at least one missing statement to be updated in the method X's body;
- *Missing statement addition:* There is at least one missing statement to be inserted in the method X's body;
- *Missing statement deletion:* There is at least one missing statement to be deleted in the method X's body;
- *Missing type update:* The type associated with field X needs to be updated;
- *Missing renaming:* Method X was not renamed;

- *Not removed method:* Method X should have been removed from Class Y;
- *Visibility problem:* Method X is not visible for one of its callers.

As an example of refactoring template, Table 5.1 presents the template rules for the Extract Method refactoring. The rules focus on (i) correct placements of the newly created method and extracted statements, (ii) reference consistency with respect to its callers, and (iii) scoping consistency with respect to parameters and local variables. We consider an extract method edit a transformation in which part of a method is moved to a newly created method, this new method is placed in the same class of the original method. Thus, extractions to different classes can be decomposed to a combination of a local extracted method and a move method. Rules 1-7 in Table 5.1 define basic steps to check whether an extract a method is successful. Failing to perform any of those steps may lead to behavior changes. Rules 5, 6 and 7 perform a binding check to verify whether the methods and fields still reference the equivalent AST elements after the refactoring edits. This binding verification is important, since simple code modifications may lead to subtle errors in variable and method references. For example, the newly created method could unexpectedly override a method from a higher hierarchy. The rules of our refactoring templates are presented in pseudo code and the following shows the meaning of the auxiliary functions. The rules for the remaining five refactoring types are available in Appendix C.

- `getClass(P, m)` returns the containing class of method `m` to be refactored;
- `getCallers(m)` returns all callers of method `m`;
- `getStatements(m, [beginLine;endLine])` returns the statements from line `beginLine` to line `endLine`. If an empty range (i.e, []) is given, it returns all statements from method `m`;
- `Set{ReferenceElement}:checkBindingProblem(m1, m2)` verifies whether all references to methods or variables are identical between `m1` and `m2`. Returns a set of problematic reference objects;
- `haveDependences(m, stm)` verifies whether the remaining statements in method `m`, after extracting statements `stm` from method `m`, depend on any statement within extracted statements `stm`.

Extract Method (P: original version, P_r: modified version, $m1_o$: method to be refactored, $m2_n$: extracted method, $[startLine, endLine]$: portion to be extracted)		
1	C_p ={}; c_o = **getClass**(P, $m1_o$); C_p = C_p ∪ {<'**Add Functionality**', $m2_n$, c_o>}	Modified version P_r must include a method not existing in original version P.
2	STM_o = **getStatements** ($m1_o$, $[startLine, endLine]$) **IF** (**haveDependences** (P, STM_o)) **THEN** C_p = C_p ∪ {<'**Update Statement**', $m2_n$, $m1_o$>};	If any extracted statements (STM_o) modifies the value of variable(s) used in the rest of the method, the modified method must have a new variable update. The updated variable must be associated to the return value of a calling to the new method.
3	**IF** (**NOT haveDependences** (P, STM_o)) **THEN** C_p = C_p ∪ {<'**Insert Statement**', $m2_n$, $m1_o$>};	If Rule 2 is not applicable, there must be a new statement related to the calling of the the new method in the modified version.
4	**FOREACH** (s **in** STM_o) **DO** C_p = C_p ∪ {<'**Delete Statement**', s, $m1_o$>}; C_p = C_p ∪ {<'**Insert Statement**', s, $m2_n$>};	For each extracted statement, there must be a deleted statement in the original method $m1_o$ and an inserted statement (same statement) in the modified method $m2_n$. The order of inserted statements must be the same as extracted.
5	C = **getCallers**(P, $m1_o$); **FOREACH** (c **in** C) **DO** m = **getMethod**(P_r, c); $BindingProbs$ = **checkBindingProblem**(c, m); **FOREACH** (b **in** $BindingProbs$) **DO** C_p = C_p ∪ {<'**Binding Problem**', b, c>};	All callers of $m1_o$ in the modified version (m from P_r) must preserve all method and variable references from the original version P.
6	C = **getCallers**(P_r, $m2_n$); **FOREACH** (c **in** C) **DO** m = **getMethod**(P, c); $BindingProbs$ = **checkBindingProblem**(c, m); **FOREACH** (b **in** $BindingProbs$) **DO** C_p = C_p ∪ {<'**Binding Problem**', b, c>};	All callers of methods with similar signature (same name but different parameters) of $m2_n$ in the modified version must preserve all method and variable references from the original version P.
7	$unionSet$ = $m1_o$ ∪ $m2_n$; $BindingProbs$ = **checkBindingProblem**($m1_o$, $unionSet$); **FOREACH** (b **in** $BindingProbs$) **DO** C_p = C_p ∪ {<'**Binding Problem**', b, $m1_o$>};	The methods and variable references in $m1_o$ must be the combination of the references from the original method in the new version $m1_o$ and the newly added one $m2_n$, except the references affected by the application of Rules 2 or 3.

Table 5.1: Template rules for extract method refactoring.

5.1.2 Reviewing Extra Edits with RefSeparator

Recent studies [63; 81] show that developers often apply refactoring in the context of bug fixes and feature additions and interleave refactorings with extra changes. Such extra edits are not always errors and could be intentionally made during refactoring. Nevertheless, according to a study at Microsoft [63], developers report that they would like to see semantic changes that deviate from pure refactoring separately.

As shown in Figure 5.1, REFSEPARATOR takes as input the RefSet generated by RefFinder. For each refactoring instance, it automatically applies an equivalent refactoring using Eclipse's refactoring APIs and creates a version with pure refactoring, P'. It compares the generated version against the manual refactoring version, P_r. If the two versions are not identical, it highlights the location of the differences for further revision.

Perform automatic refactorings is very challenging. Despite being the most well known refactoring tool, the Eclipse refactoring engine is not bug free. Several studies [26; 44; 109] found bugs in the Eclipse refactoring engine—the conditions under which refactoring generates compilation errors or behavior changes. Naively using the Eclipse refactoring engine to create a pure refactoring version can lead REFSEPARATOR to false positives. To address this problem, REFSEPARATOR first performs a checking step to avoid non-behavior preserving refactorings. For each refactoring edit, if any of the conditions is satisfied, REFSEPARATOR will not apply the automated refactoring and instead report a warning message to the user. Table 5.2 summarizes the conditions checked by REFSEPARATOR for the push down method refactoring, Eclipse generates unsafe refactoring for those scenarios. The first column indicates the bug number in the Eclipse bug tracker,[1] and the second column shows a brief description of the condition where refactoring creates a compilation error or subtle behavior change. Our technical report [12] describes all conditions that REFSEPARATOR checks for the remaining five refactoring types.

It is important to highlight that REFDISTILLER's reported refactoring faults are anomalies and not always those anomalies lead to real faults. In some cases those anomalies are deviations from the standard refactoring templates and/or automatic refactoring transformation. However, those are problems that often are associated to a bug, and/or require refactoring review to double check the developer intention.

[1]The Eclipse bug tracker—`https://bugs.eclipse.org/bugs/`

Table 5.2: Conditions checked by REFSEPARATOR to avoid push down method refactoring bugs in Eclipse

Bug	*C1: class under refactorings; C2: target class; m: method under refactorings*
320115	Method m is to be pushed down and it directly calls a method that is invisible from the target class.
348278	Method m is to be pushed down and it contains a method call using the keyword *this*.
356698 / 355322	Method m contains a super access to a method that is overridden/overloaded in class $C1$.
290618 / 355324	Method m contains a call to a method that is overridden/overloaded in the target class $C2$.
195003	Method m is to be pushed down, which contains a field access using the keyword, this.
195004	Method m is to be pushed down, whose callee invokes another method by the origin class (e.g., new ClassX().foo().)

5.2 Evaluation

To assess the effectiveness of REFDISTILLER and its capacity of complementing the testing validation, we use two evaluation methods. First, we apply REFDISTILLER to a data set of 100 refactoring transformations with seeded behavior changes (anomalies). Second we apply REFDISTILLER to three real world open source programs. In both studies we compare REFDISTILLER with regression testing suites, our baseline. In both studies, the found faults were manually validated by the authors.

5.2.1 Evaluation using a Data Set

Goal, Question and Metrics We first use a data set of 100 refactoring transformations with seeded refactoring related behavior changes in order to assess the effectiveness of REFDISTILLER on detecting refactoring related anomalies. To guide our study we define the following question: *Is* REFDISTILLER *effective in detecting refactoring anomalies?*

To answer our study question we use two of the most applied measures for evaluating search strategies, *Precision* and *Recall*. Being $TotF$: the set of all injected refactoring faults; TP (true positives): set of corrected identified anomalies; and FP (false positives): set of incorrect identified anomalies:

- $Precision = |TP|/(|TP| + |FP|)$
- $Recall = |TP|/|TotF|$

Planning and Design Each transformation from our data set is a pair (p_1, p_2) of Java programs - p_2 is p_1 after a single refactoring. Both p_1 and p_2 are free of compilation errors and p_2 contains at least one seeded refactoring related behavior change. 39 transformations were automatically identified by prior work by Soares [109]; these are subtle refactoring errors that the state of the art refactoring engines are unable to prevent or detect. The remaining anomalies are created by the authors based on the literature on refactoring errors [42; 39; 26; 111; 13], varying from missing/adding statements in a refactored method to breaking/adding overloading/overriding constraints. The data set covers all six refactoring types and includes both missing edits anomalies (50) and extra edits anomalies (50). Our technical report [12] details the data set, including a description of all anomalies.

We compare REFDISTILLER against the most used and widely used technique for validating refactorings, regression testing. Thus, testing is our baseline fault detection technique. For that, we use an automated feedback-directed random test generation tool, Randoop [89], to create JUnit test suites in for low level integration. Suites created by Randoop have been successfully used to identify refactoring problems [111; 109; 108; 77]. To increase the variability of the generated tests, a set of independent extra methods was added to each transformation. We run Randoop with the time limit of 100 seconds and the maximum test size of 5 statements. On average, we have test suites with 5,332 test cases for each version pair in the data set.

First, we run REFDISTILLER to each version pair in our data set. We collect the results for REFCHECKER and REFSEPARATOR respectively and manually validate their outcomes. Next, we run the regression test suite, which is generated for the original version, and then rerun it for the refactored version. Any differences in the test outcome are then considered a fault detected by regression testing.

Table 5.3: Result of recall and precision

	Recall		Precision
	Missing Edits	Extra Edits	
REFCHECKER	94%	-	75%
REFSEPARATOR	-	98%	98%
REFDISTILLER	96%	98%	84%
Testing	84%	66%	100%

Data Analysis and Discussion Here we present the main results of this study and discuss them regarding improvements of refactoring fault detection.

Recall. We assess how many seeded anomalies can be found by REFDISTILLER. REFDISTILLER detects 97% of all seeded anomalies, outperforming regression testing by 22 faults. Table 5.3 summarizes the results. REFCHECKER detects 47 anomalies out of 50 and REFSEPARATOR detects 49 anomalies out of 50.

REFDISTILLER finds refactoring anomalies that are not easy to identify because they require understanding of complex code structures, e.g., overriding relationships in a deep class hierarchy. For instance, one of the callers of an inlined method is not updated after refactoring, referencing a different method with the same signature from its super class at a higher level. This type of anomaly is not always predicted when using testing, and generates subtle behavior changes that can easily pass unnoticed.

False Negatives. Three anomalies are not detected by REFDISTILLER (only one of them was detected by the test suite). Those are anomalies that require a runtime investigation and/or that change the state of an object/variable due to exception handling. Figure 5.2 exemplifies one of those false negatives. Lines 5-8 from `Element.m` are extracted into method `n`. In the original version, when the exception `e` is thrown at line 7, method `m(boolean)` returns the state of `x` before the `try` block, `x = 42`. However, after extracting the method, when the same exception is thrown, the state of variable `x` is already updated and method `m` returns `23` instead. This missing edit fault is hard to identify using REFCHECKER as our current templates are not capable of capturing the complex changes in the control flows due

```
class Element {
  int m(boolean b) {
    int x = 42;
    try {
      if (b) {
        x = 23;
        throw new Expt();
      }
    } catch (Expt e) {
      return x; }
    return x; }
  int test() {
    return m(true);}
}
```

(a) Original version.

```
  class Element{
    int m(boolean b) {
      int x = 42;
      try {
-       if (b) {
-         x = 23;
-         throw new Expt();
-       }
+       x = n(b, x);
      } catch (Expt e) {
        return x; }
      return x; }
+   int n(boolean b, int x) throws Expt {
+     if (b) {
+       x = 23;
+       throw new Expt();
+     }
+     return x; }
    int test() {
      return m(true); }
  }
```

(b) Code after a problematic extract method edit. Method m returns a different value when exception e is thrown due to a different of variable x's state.

Figure 5.2: Missing edit fault: extract method edits.

to incomplete refactoring. Heuristics to consider those scenarios are planned as future work.

RefDistiller detects 22 anomalies that are not found by the generated test suites. Figure 5.3 shows a case where testing fails to detect the faults. Lines 5-6 from `Calc.getVal` are extracted to `extrMeth`, and `if/else` statements are added to the new method (see Figure 5.3(b), underlined code). The extra edit changes the control flow depending on inputs. Although the generated test suite consists of more than 12,000 test cases and has a high testing coverage ($\approx$ 85%), this fault is not detected. This is because test suites are built for the old version, thus it is hard to predict how extra edits may affect pre-existing methods or new ones. Thus, existing test cases are incapable of exploring this new path. To identify the bug from 5.3(b), the developer, or the test generator tool, should have included a test case with a negative input to `getVal` in the old version, which is not the case.

```
class Calc {
  int getVal(int amnt) {
    if (amnt > 10) {
      int x = amnt + 10;
      int z = x + 10;
      return z;
    } else {
      int x = amnt * amnt;
      return x; } }
}
```

(a) Original version.

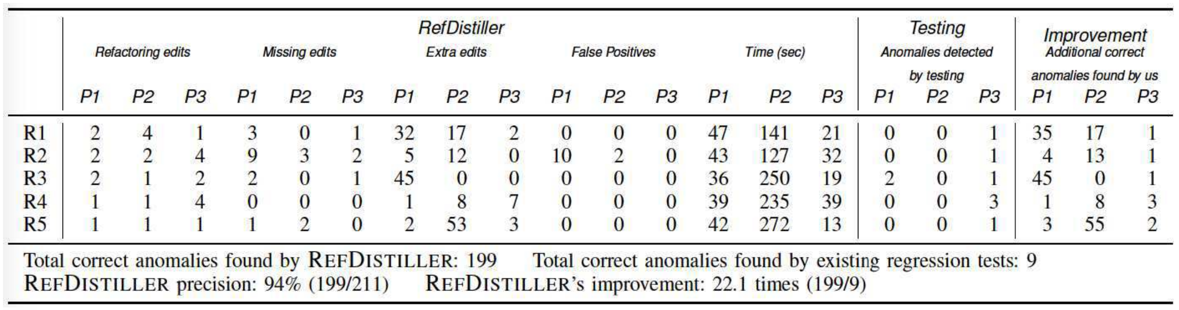
```
  class Calc {
    int getVal(int amnt) {
      if (amnt > 10){
        int x = amnt + 10;
-       int z = x + 10;
-       return z;
+       return extrMeth(x);
      } else {
        int x = amnt * amnt;
        return x; } }
+   int extrMeth(int x) {
+       if (x > 0)
+       return x + 10;
+       else return -1;
+   }
  }
```

(b) Code after extract method refactoring with extra edits underlined.

Figure 5.3: Fault detected by REFDISTILLER but not by testing

Precision. We assess REFDISTILLER precision—how many anomalies it detects are indeed true faults. REFCHECKER finds 63 omission anomalies, of which 47 are correct, resulting in 75% precision. REFSEPARATOR finds 50 extra edit anomalies, of which 49 are correct. REFDISTILLER in total finds 113 faults, of which 96 are correct, resulting in 84% precision.

Most incorrectly detected instances occurred due to incorrect identification of refactoring types by RefFinder. Though there are only one hundred refactorings seeded in the data set, RefFinder identifies 32 additional refactoring instances. For instance, for a single pull up method refactoring, RefFinder reports both move method and pull up. REFCHECKER therefore checks two templates to check for potential refactoring missing edit anomalies, where it finds a false positive with respect to move method. For a push down refactoring with an extra edit, RefFinder reports three refactoring types: move method, extract method, and push down. The edit moved method `calc` from class `A` to subclass `B`, turning the method call to delegation `B.calc`. When REFSEPARATOR applies push down method and move method refactorings, it correctly reports two types of extra edit anomalies. On the other hand, when it applies the false positive extract method refactoring with Eclipse, it generates an incorrect comparison version. REFSEPARATOR then attempts to compare the



	RefDistiller															Testing			Improvement		
	Refactoring edits			Missing edits			Extra edits			False Positives			Time (sec)			Anomalies detected by testing			Additional correct anomalies found by us		
	P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3
R1	2	4	1	3	0	1	32	17	2	0	0	0	47	141	21	0	0	1	35	17	1
R2	2	2	4	9	3	2	5	12	0	10	2	0	43	127	32	0	0	1	4	13	1
R3	2	1	2	2	0	1	45	0	0	0	0	0	36	250	19	2	0	1	45	0	1
R4	1	1	4	0	0	0	1	8	7	0	0	0	39	235	39	0	0	3	1	8	3
R5	1	1	1	1	2	0	2	53	3	0	0	0	42	272	13	0	0	1	3	55	2

Total correct anomalies found by REFDISTILLER: 199 Total correct anomalies found by existing regression tests: 9
REFDISTILLER precision: 94% (199/211) REFDISTILLER's improvement: 22.1 times (199/9)

Table 5.4: Results regarding anomalies detection and Improvement on XML-Security (P1), JMeter (P2), and JMock (P3).

incorrect comparison refactoring version with the manual version and subsequently reports a false positive extra edit anomaly.

5.2.2 Evaluation using Real World Programs

Hoping to extend the investigation on REFDISTILLER's effectiveness, and to assess its applicability to real scenarios, we apply REFDISTILLER to three open source projects: XMLSecurity—a library for providing security functionality for XML such as authorization and encryption, JMeter—a performance analysis application for measuring performance under different load types, and JMock—a library for testing applications with mock objects.[2].

Data Set For XMLSecurity and JMeter, we use a data set from prior work on refactoring change impact analysis [94]. This work identifies real refactoring edits performed throughout several versions of these systems. The subject programs are drawn from Software Infrastructure Repository (SIR) [27]. XMLSecurity is 18 KLOC and JMeter 32 KLOC respectively. As SIR provides release-level changes, to identify a commit with refactoring edits, we mine commit histories and map each refactoring edit to the earliest and closest commit revision. For JMock, we go through its repository history and search for commits with refactoring edits using RefFinder.[3] From this mining, we selected 3 versions (v1-v3). We also create additional four versions by seeding refactoring faults: (v4) incomplete move method

[2]JMeter `http://jmeter.apache.org/`

XMLSecurity `http://jmeter.apache.org/`

JMock `http://jmock.org/`

[3]`https://github.com/jmock-developers/jmock-library/commits`

refactoring with incorrect method call bindings, (v5) extract method method with missing statements, (v6) extract method refactoring with a missing update in its caller, and (v7) inline method refactoring without required deletions. Those anomaly versions are all free of compilation errors and are inspired by the literature on refactoring errors [42; 39; 26; 111; 13]. Each project version used in our study has a test suite manually created by their developers for testing the system at a system level. The size of these suites varies from dozens to hundreds of JUnit test cases.

Analysis and Discussion Table 5.4 shows the number of missing and extra edits detected by REFDISTILLER: XMLSecurity (P1), JMeter (P2), and JMock (P3). Rn is each revision pair.

Running REFDISTILLER takes on average 41.4, 205, and 24.8 seconds for the three projects. We manually inspect individual anomalies to determine correct anomalies and precision is 90%, 97%, and 100% for the three projects respectively.

We compare REFDISTILLER's anomaly detection capability with the fault detection capability of existing JUnit regression tests included in SIR. The faults detected by REFDISTILLER and by testing were later manually inspected. The existing regression test suites are inadequate for covering refactoring edits, they detects only 2, 0, and 7 refactoring faults in the subject programs. In fact, those suites have also a low branch coverage in general (above 50% on average). The results are aligned with the prior finding that only a small portion of refactoring edits are tested by existing regression tests and refactoring mistakes can be easily unnoticed [94].

Because REFDISTILLER applies a static analysis approach, and focuses on the location of refactoring edits, it finds 190 additional correct faults not found by testing. We cannot measure recall because we do not know the total number of all true refactoring faults in these versions. Therefore, we measure the improvement by dividing the total number of REFDISTILLER's true faults by the total number of faults found by testing. The overall improvement is 22.1 times. The results show that REFDISTILLER can effectively complement a testing-based approach for refactoring validation.

False Positives. Although we cannot generalize, in the context of our study, our false positive rate is low. In a few cases, false positives are caused by the combination of missing edits and extra edits at the same method location. For example, REFDISTILLER identifies two extract method edits in the same method body. To obtain the pure refactoring version, REFSEPA-

RATOR subsequently applies two extract method refactorings to two different edit positions. However, the refactoring application to the first edit position affects the second refactoring edits, changing the second edit position. Applying multiple refactorings in the same method could prevent REFDISTILLER from locating extra edits correctly. Moreover, to have a deep understanding on REFSEPARATOR's false positive rate, we would have to ask developers from each project whether a found extra edit is in fact intended or not, which would have been impractical in the context of this study. Therefore, in a real scenario, REFDISTILLER's number of false positives tends to be higher. However, considering a refactoring reviewing scenario, where a developer is reviewing a refactoring that might have been applied by other(s), we consider that any found extra edits are worth to be noticed and must be reviewed.

Hints for Correcting Refactoring Anomalies. When REFDISTILLER reports warning messages about missing and extra edits, it provides *hints* for fixing refactoring anomalies. For example, REFDISTILLER outputs a warning message about missing edits like "REFDISTILLER *detected that a method extract refactoring edit caused a possible binding problem of the reference* `foo()` *in class* `ClassA` *at line 1,114. This may have affected method* `ClassB.bar()`." As another example, REFDISTILLER reports an warning message about extra edits like "REFDISTILLER *detected that a pull up method refactoring edit with extra edits in Z.baz() at line 1411*". We believe that such warning messages could help developers debug refactoring mistakes and even fix then. For extra edits, we display a comparative view in which a pure-refactoring version of the code is put against the manual version with the extra edits highlighted. That way, a developer can review his edits and undone the extra edits if they were not intended. As far as we know, REFDISTILLER is the first tool that helps developers on debugging/fixing refactoring faults by providing this extra information.

Final Remarks With the results obtained with our seeded data set and with the study using the open-source projects we can answer our third research question (**RQ3**) by stating that by using a static analysis strategy we can effectively detect refactoring problems with high rates of recall and precision. In both studies REFDISTILLER outperformed testing, baseline refactoring validation technique, and detected several new refactoring anomalies. Moreover, we believe that REFDISTILLER's hints for fault localization is an useful source of information that tests often lack to provide. Thus, we can state that REFDISTILLER complements the testing validation.

5.3 Threats to Validity

In terms of **construct validity**, the accuracy of RefFinder's refactoring reconstruction directly affects REFDISTILLER's capability in detecting refactoring faults. We refer to Prete et al. [93] for further evaluation data on how varying its similarity threshold affects RefFinder accuracy. When multiple interfering refactorings are seeded in the same version together, REFDISTILLER may find false positives or false negatives. For example, when we seed the extract method and inline method refactorings in the same location, which partially cancel each other, REFDISTILLER finds two separate faults that should not be found—one related to the extract method and the other related to inline method. Moreover, as discussed in the *False Positives* subsection, in practice, REFSEPARATOR's false positive rate tends to be higher. However, we understand that there is a need for confirmation of any extra edit in the refactoring reviewing phase, since those extra edits might interfere with a software's previous stable behavior.

Since it applies an static analysis, REFDISTILLER is currently unable to detect runtime object types precisely and cannot capture control flow changes in the exception handling logic precisely, etc.

REFDISTILLER takes the old and new version snapshots as input and does not leverage edit history during programming sessions (e.g., a sequence of edit operations recorded in Eclipse). Extending and applying REFDISTILLER to edit histories or other richer refactoring data [84; 117] remains as future work.

In terms of **internal validity**, we detect refactoring anomalies where behavior changes occur. Not all identified faults are indeed errors and could be intentional. Nevertheless, paying attention to behavior changes occurred during refactoring is worthwhile.

In terms of **external validity**, our evaluation results do not generalize beyond the three subject programs and our data set. In our evaluation, REFDISTILLER is able to detect anomalies not found by either exiting test suites or auto-generated ones. However, a test suite's refactoring validation power is relative to several factors that might influence its effectiveness, such as coverage level, or variability of data. Thus, the potential gain of using REFDIS-

TILLER, instead of testing, might be relative, however, we believe that the combination of both strategies (testing + REFDISTILLER) are likely to allow developers to better validate/review refactorings. Finally, the injected faults do not cover all possible refactoring related bugs. However, these faults were collected, from previous studies that identified problems in automatic refactoring tools, and/or or inspired by real scenarios experienced by the authors when performing/reviewing refactorings.

5.4 Concluding Remarks

This chapter presents REFDISTILLER, an approach for helping developers to review refactorings. In addition to anomaly detection, REFDISTILLER also provide useful information for guiding developers to locate and fix refactoring related problems. REFDISTILLER ask developers to confirm if extra edits that are included while refactoring and are in fact intentional. The evaluation shows that our static analysis approach can effectively complement testing-based refactoring validation. REFDISTILLER finds several anomalies that existing and generated test suite falls short of covering.

To the best of our knowledge, REFDISTILLER is the first to detect both missing edits and extra edits that deviate from pure refactoring. Its capability can help developers focus their attention to subtle behavior changes with no compilation errors, which require revision since can easily slip unnoticed.

In the next chapter we present the two Eclipse plug-ins developed to enable developers to easily start using the solutions proposed in our work, RBA and REFDISTILLER.

Chapter 6

Tools

How to efficiently validate, locate and fix those faults is a constant concern throughout a developer's daily routine. Both solutions presented in the previous chapters (RBA [98] and REFDISTILLER [11]) aim to address those issues by using artifacts that are often available when refactoring (e.g., test suites, source code). In order to motivate developers to use our solutions and reduce the overhead that their application might bring, we developed two Eclipse plug-ins (PRIORJ and REFDISTILLER). Those tools instantiate our solutions for Java/JUnit systems.

6.1 PriorJ

The PRIORJ tool[1] [98] is an open-source Eclipse plug-in developed in order to allow developers to apply test case prioritization. For that, it embodies a prioritization framework that provides the following features: i) a testing coverage analysis module; ii) a set of seven prioritization algorithms (TSC, TMC, ASC, AMC, RD, CB, RBA); iii) prioritization in a batch mode; iv) generation of executable artifacts to run the new prioritized suites; v) measuring and visualization of quality metrics; and vi) history of prioritization suites and runnings.

By default, PRIORJ includes six of the most widely used prioritization techniques (TSC, TMC, ASC, AMC, Random and Change Blocks) for Java/JUnit systems. Moreover, the RBA technique is also part of PRIORJ. This set of techniques can be easily extended. PRIORJ provides a set of interfaces and modules that enables the inclusion of new prioritization algo-

[1]https://github.com/samueltcsantos/priorjplugin/

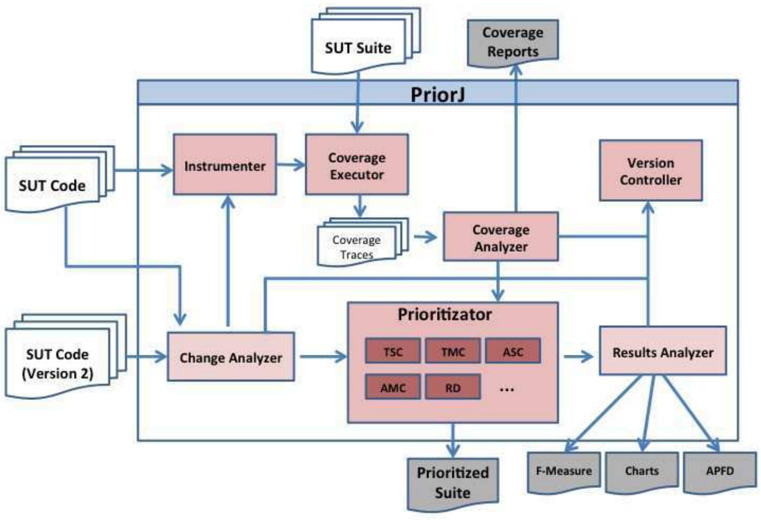


Figure 6.1: PriorJ Architecture.

rithms by implementing a single new Java class that deals only with the test case reordering. Other issues such as project's characteristics capture, the building of runnable Java files that enable the new execution order, and etc, are provided by PRIORJ.

As most prioritization techniques use testing coverage data, PRIORJ implements its own integrated coverage analyzer. For that, it applies an strategy that uses AspectJ [59] files to run JUnit regression test suites and collect coverage traces in both statement and method levels [10]. Those traces are recorded and later used by the prioritization techniques.

Figure 6.1 presents the PRIORJ's architecture. Rectangles represent PRIORJ modules, arrows represent the relationship between modules, and elements out of the PrioJ box with an incoming arrow are PRIORJ's external outputs:

Instrumenter. Module that instruments the SUT code in order to make it ready for a further coverage analysis. For that, it creates a temporary instance of the SUT project and spreads marks throughout the code of the temporary project.

Coverage Executor. Responsible for running the test suite and collect the coverage data. This module produces as output artifact a set of log files with coverage traces.

Coverage Analyzer. The files produced by the Coverage Executor module are read and manipulated in order to generate coverage reports and treat the data for prioritization.

Change Analyzer. Module responsible for identifying, from two versions of a Java program, the code parts that are different. For that, it compares classes at the AST level.

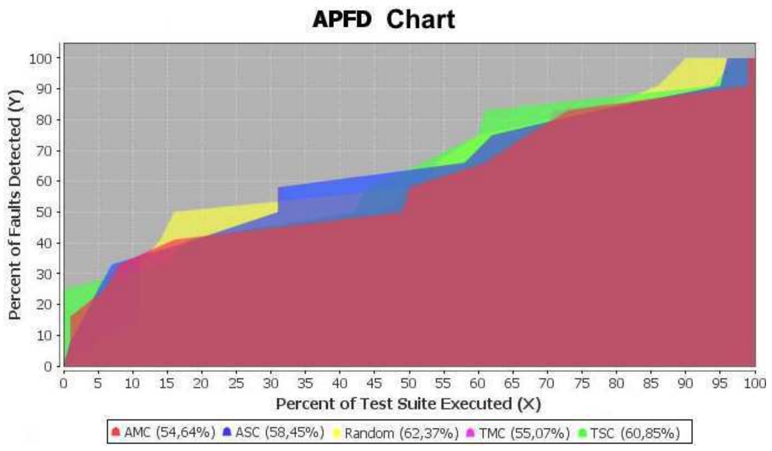


Figure 6.2: PriorJ APFD chart.

Prioritizator. PriorJ's core module. Module that implements the prioritization algorithms/techniques/approaches. Each prioritization technique is represented by a single Java class that implements an interface of methods. As output, this module generates: i) a file with the test cases prioritized sequence; ii) an executable Java class that enables the prioritized suite execution; and iii) the data to be used for calculating the evaluation metrics (module Results Analyzer).
Results Analyzer. Module that calculates quality metrics for prioritized suites (F-Measure, APFD). Moreover, this module is responsible for generating quality charts and for reducing suites according to the user's needs.
Version Controller. Module that implements a version control mechanism. For that, it manages the creation, storage and update of prioritization projects and data.

In [10], we discuss challenges that developers/testers commonly face when using prioritization in practice. Moreover, we depict scenarios to help developers to build and use prioritization tools/environments. PRIORJ implements all those scenarios and generated useful artifacts to help to developers/testers to overcome those challenges. For instance, Figure 6.2 presents a comparative chart in which the APFD results are plotted after running several prioritization techniques in batch mode. Each area presents the collected APFD results for each prioritization technique. By analyzing this chart, a tester can decide which technique is better suited to be used in his project.

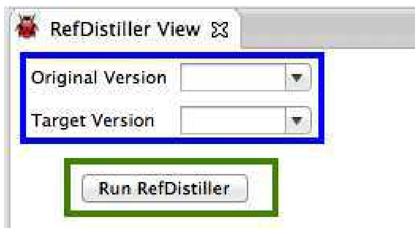


Figure 6.3: RefDistiller's input view

6.2 RefDistiller

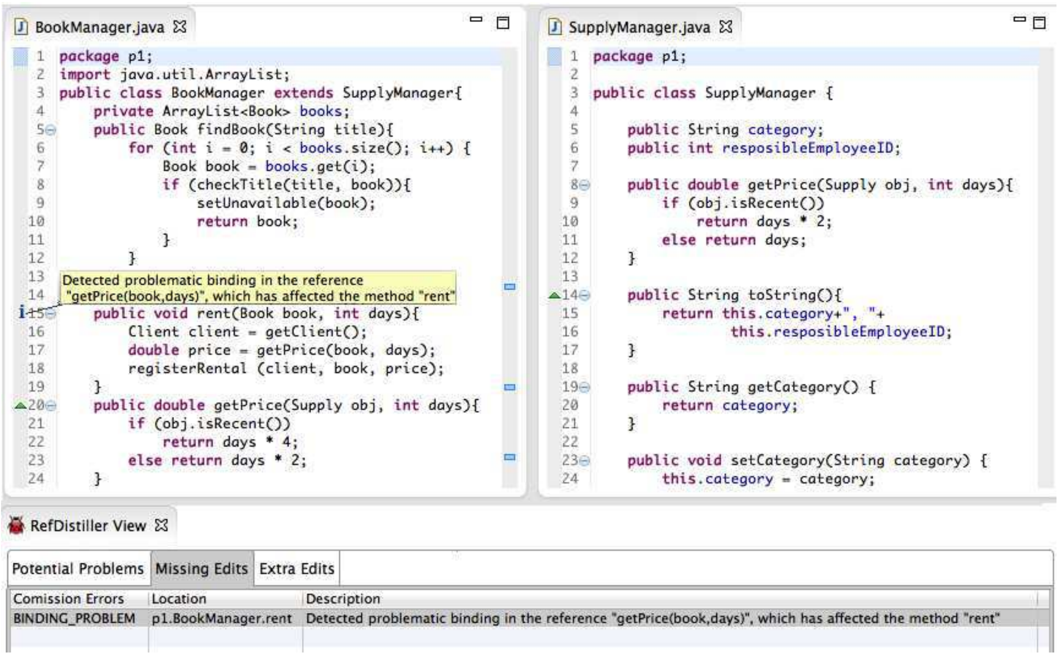
Aiming not only to implement the REFDISTILLER solution (described in Chapter 5), but also to enable the inspection/review of refactoring edits, we developed the REFDISTILLER tool [11]. RefDistiller[2] is a refactoring-aware code review Eclipse plug-in. To detect potential deviations from pure refactoring edits, REFDISTILLER incorporates two key techniques: i) *RefChecker* for detecting missing edits; and ii) *RefSeparator* for detecting extra edits (more details in Chapter 5). Currently, REFDISTILLER supports six of the most common refactoring types for Java programs: move method, extract method, inline method, rename method, pull up method, and push down method.

When running the REFDISTILLER view, a user first selects both versions of a project (Figure 6.3), the original and the target one (after refactoring). After that, a new view is triggered. In the first tab of this view (*Potential Problems*) all potential deviations from pure refactoring edits reported by both RefChecker and RefSeparator. Each line describes the type of a refactoring problem such as `BINDING_PROBLEM`, the location (e.g., method rent from class `p1.BookManager`), and a short description of the problem. By clicking on each line, a developer can review the corresponding Java files, where the potential problems are located.

Missing Edits View. By clicking the Missing Edits tab, the user sees the problems reported from RefChecker. RefChecker reports the nine types of warning messages discussed in Chapter 5.1.1.

By clicking Missing Edits View's each line, the respective Java file is opened and a warning message is tagged at the location of the warning. Figure 6.4 exemplifies RefDistiller's output for a refactoring fault related to a missing edit. In this scenario, a devel-

[2]https://sites.google.com/site/refdistiller/



BookManager.java

```
 1 package p1;
 2 import java.util.ArrayList;
 3 public class BookManager extends SupplyManager{
 4     private ArrayList<Book> books;
 5     public Book findBook(String title){
 6         for (int i = 0; i < books.size(); i++) {
 7             Book book = books.get(i);
 8             if (checkTitle(title, book)){
 9                 setUnavailable(book);
10                 return book;
11             }
12         }
13
14
15     public void rent(Book book, int days){
16         Client client = getClient();
17         double price = getPrice(book, days);
18         registerRental (client, book, price);
19     }
20     public double getPrice(Supply obj, int days){
21         if (obj.isRecent())
22             return days * 4;
23         else return days * 2;
24     }
```

Detected problematic binding in the reference "getPrice(book,days)", which has affected the method "rent"

SupplyManager.java

```
 1 package p1;
 2
 3 public class SupplyManager {
 4
 5     public String category;
 6     public int resposibleEmployeeID;
 7
 8     public double getPrice(Supply obj, int days){
 9         if (obj.isRecent())
10             return days * 2;
11         else return days;
12     }
13
14     public String toString(){
15         return this.category+", "+
16                 this.resposibleEmployeeID;
17     }
18
19     public String getCategory() {
20         return category;
21     }
22
23     public void setCategory(String category) {
24         this.category = category;
```

RefDistiller View

Potential Problems | Missing Edits | Extra Edits

Comission Errors	Location	Description
BINDING_PROBLEM	p1.BookManager.rent	Detected problematic binding in the reference "getPrice(book,days)", which has affected the method "rent"

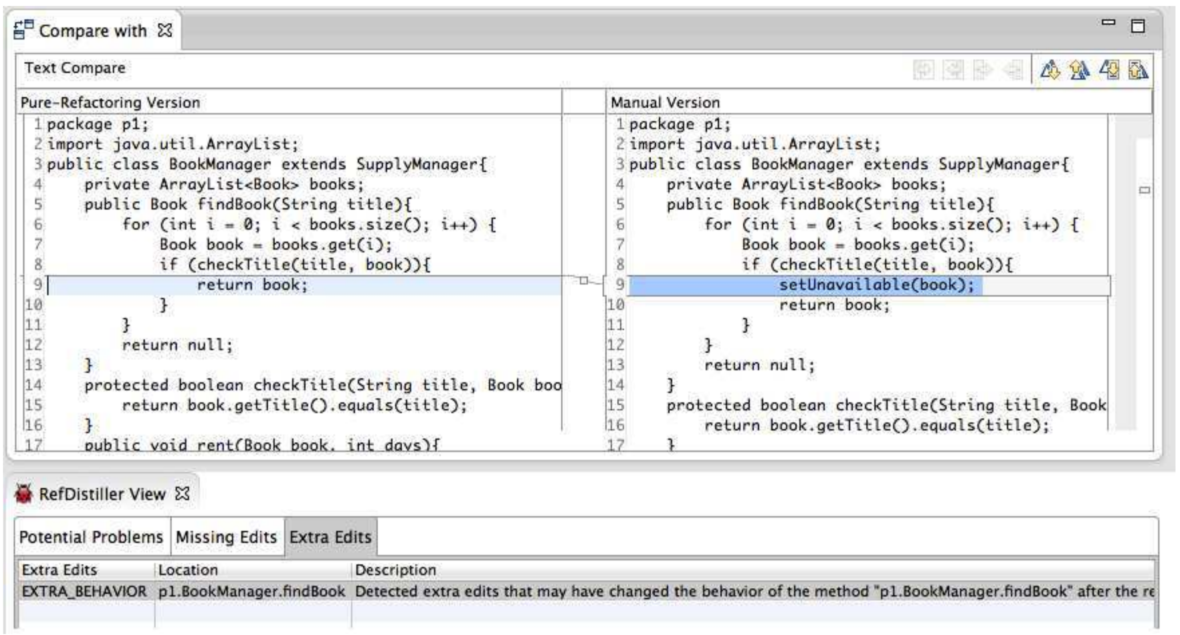
Figure 6.4: Example of how RefDistiller outputs a missing edit fault.

oper can see that a `BINDING_PROBLEM` was added after a refactoring because the call to `getPrice(book,days` in `rent` references different methods when comparing before and after the refactoring (see mark at line 15).

Extra Edits View. By clicking the Extra Edits tab, a user can see the problems reported from RefSeparator. By clicking each line, an Eclipse Compare View is open. In this view the developer can examine the program differences between the manual refactoring version and a pure refactoring version generated by RefDistiller. Figure 6.5 shows a pure refactoring version on the left side and the manual refactored version on the right side. When the two versions are different, RefSeparator reports that the manual version is not pure refactoring and pinpoints the location of extra edits, in this case line 9 calling `setUnavailable`. When a developer sees a warning message, “Detected extra edits that may change the behavior of method `findBook`”, he can check whether he intended to introduce this new behavior.

6.3 Concluding Remarks

This chapter presents the two open source tools designed and implemented during our research, RBA and REFDISTILLER. Those tools are Eclipse plug-ins and can easily be incor-



Pure-Refactoring Version

```
package p1;
import java.util.ArrayList;
public class BookManager extends SupplyManager{
    private ArrayList<Book> books;
    public Book findBook(String title){
        for (int i = 0; i < books.size(); i++) {
            Book book = books.get(i);
            if (checkTitle(title, book)){
                return book;
            }
        }
        return null;
    }
    protected boolean checkTitle(String title, Book boo
        return book.getTitle().equals(title);
    }
    public void rent(Book book, int days){
```

Manual Version

```
package p1;
import java.util.ArrayList;
public class BookManager extends SupplyManager{
    private ArrayList<Book> books;
    public Book findBook(String title){
        for (int i = 0; i < books.size(); i++) {
            Book book = books.get(i);
            if (checkTitle(title, book)){
                setUnavailable(book);
                return book;
            }
        }
        return null;
    }
    protected boolean checkTitle(String title, Book
        return book.getTitle().equals(title);
    }
```

Extra Edits	Location	Description
EXTRA_BEHAVIOR	p1.BookManager.findBook	Detected extra edits that may have changed the behavior of the method "p1.BookManager.findBook" after the re

Figure 6.5: Example of how RefDistiller outputs an extra edit fault.

porated to a developers' daily routine. Those tools evidence the practical character of our work. In the next chapter we present and discuss works at some point related to our research, comparing our solutions to similar ones when adequate.

Chapter 7

Related Work

In this section we discuss research results that are related to ours. We focus our discussion on studies that investigate the role of testing coverage for detecting faults and that evaluate a suite's effectiveness (Section 7.1); studies that propose/evaluate test case prioritization approaches (Section 7.2); studies on change impact analysis for object oriented languages (Section 7.3); and on refactoring and refactoring tools (Section 7.4).

7.1 Testing Coverage and Suite Effectiveness

Long-established texts on software testing (e.g. [91]) recommend the use of coverage to gain confidence that a test suite is effective on detecting faults. Extensive research on this topic, however, presents mixed conclusions. Frank and Iakounenko [34] report the chance of detecting a fault increases sharply with very high coverage rates. While Wong et al. [123] find a suite effectiveness is highly correlated with block coverage, Xia and Lyu [22] observe a moderate correlation between coverage and fault detection, depending on the tests nature. Furthermore, Gligoric et al. [43] state that fault detection effectiveness is moderately correlated with coverage in general, and project size. On the other hand, Frankl and Weiss [35] found a weak correlation between suite coverage and its error-exposing ability. Later work [36] investigate the relationship between coverage and effectiveness for fixed-sized randomly generated test suites, and an improvement in effectiveness with higher coverage is observed (although there is no nonlinear relation). Briand and Pfahl [20] do not find any causal dependency between test coverage and defect coverage with respect to four coverage criteria (block, c-use, decision, p-use), independently of suite size. Inozemtseva and Holmes [52] find there is a low to moderate correlation between coverage and suite effectiveness, and that stronger forms of coverage do not provide better data to measure effectiveness of a suite. The empirical study presented in Chapter 3 differs from all of the above by focusing its investigation on the role of testing coverage to evaluate a suite aiming to identify refactoring faults. Moreover, instead of considering coverage data as it is, our study observe coverage of specific classes of impacted code elements. Our study also concludes that testing coverage indeed can be used for detecting refactoring faults when focusing on specific code locations.

Gargantini et al. [37] propose a framework to evaluate the robustness of a test suite with respect to semantic preserving transformations. For that, they define new coverage metrics that consider the fragility of coverage. Our study applies a different approach by using coverage of commonly impacted element for predicting a suite's capability of detecting a refactoring fault. Moreover, our study's results are extracted based on data from real projects and can be used as reference for helping testers to decide which code locations need more testing when validating a refactoring.

To the best of our knowledge, our exploratory study is the first to investigate the combined use of impacted elements and test coverage to evaluate a suite effectiveness and give warnings regarding suite weaknesses with respect to refactoring fault detection.

7.2 Test Case Prioritization

To the best of our knowledge, RBA is the first prioritization approach dedicated to an early detection of refactoring faults. However, a considerable amount of research focusing on test case prioritization has been done in the past decades.

Singh et al. [107] perform an elaborated literature review mapping the state-of-art of test case prioritization. From an initial amount of 12,977 studies, 106 prioritization techniques are discussed and classified among eight categories: i) *Coverage Based*; ii) *Modification-based*; iii) *Fault-based*; iv) *Requirement-based*; v) *History-based*; vi) *Genetic-based*; vii) *Composite approaches*; and viii) Other approaches. Even though there is a Modification-based category, none of the techniques identified by Singh et al. consider exclusively either refactoring edits or the impact that those edits may generate in the SUT code. Similarly, the systematic mapping study performed by Catal and Mishra [23] did not find any approach in this context. This fact emphasizes the novelty that RBA brings whereas our prioritization approach focuses specifically on early detection of refactoring faults. Moreover, the found modification-based techniques require a model representation of the software code (e.g., finite state machines) for performing their prioritization. Those models are not always available or easy do get. RBA in the other hand uses as prioritization heuristics SUT localities, which, in most of cases, is more practical, and less costly.

Coverage-based prioritization techniques are simple and often used in practice. Techniques from this group base their prioritization on the assumption that test cases that cover more elements (e.g., statements, methods), are most likely to reveal faults. Those techniques have been successfully used for speeding up the detection of faults in general. Rothermel et al. [100] present several coverage-based prioritization techniques and evaluate them according to their ability of improving the rate of fault detection. The authors show that even the last expensive technique significantly improves the suites' rate of fault detection. They also suggest that there might be room for improvement of the traditional test case prioritization techniques. These conclusions are reassured by our empirical studies [7; 5; 4] in which we observed that the general-purpose techniques are not adequate when aiming at an early detection of refactoring faults. However, the combination of testing coverage and impact analysis showed to be effective in this context.

Srivastava and Thiagarajan [114] propose a modification-based prioritization technique that focus on rescheduling regression test cases according to their coverage on modified the code blocks. This technique is used in our empirical studies (labeled as CB). By focusing only on the modified parts of the code, this technique ends up not capturing indirect impacts that a change may have in the code, *i.e.* behavior changes that are not directly associated to the changed code but to their impact (e.g., a method renaming may negatively affect several methods in an inferior hierarchy level). Srivastava and Thiagarajan's technique does not consider those situations when prioritizing. Also, they do not apply any specialization regards to the type of the changes. Thus, these techniques showed to be less effective when dealing with refactoring faults (Chapter 4 and [7]).

Other recent studies have contributed to advance the state-of-art on test case prioritization. In [125] and [46], the authors propose an unified test case prioritization approach that encompasses both the *total* and *additional* strategies. For that, they propose models in by which a spectrum of test case prioritization techniques ranging from a purely total to a purely additional technique. Jeffrey and Gupta [54] present an algorithm that prioritizes test cases based on coverage of statements in relevant slices of the outputs of test cases, and compare their proposed technique with conventional code coverage techniques. Korel et al. [64] propose prioritization techniques based on system models that are associated with code information. Srikanth et al. [112] propose a prioritization approach based on requirements volatility, customer priority, implementation complexity, and fault proneness of the requirements. Walcott et al. [118] present a prioritization technique based on a genetic algorithm to reorder test suites in light of testing time constraints. Park et al. [90] use historical information to estimate the current cost and fault severity for cost-cognizant test case prioritization. Mei et al. [72] propose an static approach for prioritizating JUnit test cases in the absence of coverage information. Sanchez et al. [103] explore the applicability of test case prioritization techniques to software product line testing by proposing five different prioritization criteria based on common metrics of feature models.

7.3 Change Impact Analysis

Both RBA and REFDISTILLER uses a lightweight change impacted analysis that focus on commonly impacted code localities (Refactoring Fault Models and Refactoring Templates). Traditional change impact approaches often use either static or dynamic data, however, the combination of static and dynamic data, although rare, can also be found. RBA's fits in the hybrid group, while REFDISTILLER is a static approach.

Bohner and Arnold [19] measure impact through reachability on call graphs. Although very intuitive, this strategy can be very imprecise as it only tracks methods downstream from the changed method. On the other hand, Law and Rothermel [66] propose *PathImpact*, a dynamic impact analysis strategy based on whole path profiling. From the changed method, *PathImpact* goes back and forth in the execution track in order to determine the impact after the change. One big issues related to any change impact analysis is that, in general, when dealing with a big change (e.g., a structural refactoring) they tend to collect a great number of methods as possible impacted. When relating those impacted method to regression tests, a great number of test cases may have to be selected as possibly impacted. RBA reduces this risk by focusing on the locations that a refactoring edit most commonly affect.

Ren et al. [97; 96] first decompose the differences of two versions of a software into atomic changes (e.g., *add a field*, *delete a field*). Then, they use a test cases call graphs analysis based on rules that identifies the subset of tests, which is potentially affected by the changes. RBA performs a similar strategy for detecting the possible affected test cases by using call graphs. However, RBA's impact analysis is performed by fault models. Ren et al.'s approach does not distinguish the type of edit that is performed. As RBA differentiates the edits by type and prioritizes according to their own specific characteristics and impact, we can say that RBA's output is specialized for refactorings. Furthermore, by working with a different granularity level (refactoring edits, instead of atomic changes), RBA tends to select a smaller group of test cases. This smaller set might not detect the faults in 100% of the cases, however, our results have shown that, in general, RBA produces very good results. Moreover, we believe developers would prefer to work with a smaller set of test cases, specially in context with strict resource constraints.

Zhang et al. [126; 127] extend Ren et al.'s work by improving their impact analysis, and introducing an spectrum analysis that helps the inspection of the changes, ranking them according to their chances of localizing faults. This work evidences the importance of making the fault debugging process easier, a secondary goal of our work. With RBA we intend to avoid the need of this spectrum analysis by efficiently selecting the test cases that are directly related to the common refactoring faults. Thus, the test cases more likely to reveal those problems are placed in the top positions of our prioritized suite and also close to each other.

Wloka et al. [120] propose an approach and tool that uses a specific change impact analysis to guide developers in creating new unit tests. This analysis identifies code changes not covered in the current test suite, and indicate if tests are missed for exercising those changes. If a problem is found, a developer can choose to add or extend a test to cover the problematic locations. Similar to other work, and different from ours, this approach works with a change impact analysis with finer granularity (an edit is decomposed into atomic changes), which might generate a bigger impact set than the one collected by our RFM. Moreover, this work goes towards test augmentation, which is not the focus of our work. We focus on project with massive test suites in which there is a need to speed up fault detection.

Mongiovi et al. [77] propose SafeRefactorImpact, an extension of the SafeRefactor tool [111] that includes an impact analysis step. SafeRefactorImpact detects non-behavior-preserving transformations in both object and aspect-oriented programs. For that, it decomposes an edit into small-grained transformations and analyses the impact of each one, composing a set of impacted methods. Then, similar to SafeRefactor, it uses the Randoop tool to generate random test cases that are run against two version of the program. SafeRefactorImpact's and RBA's refactoring fault models (RFM) are similar in the sense that both focus on code locations that are commonly impacted in a refactoring edit. However, our solution differs by the granularity of the applied impact analysis. For instance, by focusing on the impact of a refactoring edit as a whole, a RFM enable the selection of a smaller impacted set, and consequently a smaller number of test cases are selected and used for prioritization. Moreover, RBA is designed for accelerating the detection of refactoring faults in a sound test suite, while SafeRefactorImpact generates new tests for that focus on detection those faults. Thus, we believe that both approaches, as well as SafeRefactor, can be used in combination. Finnaly, when comparing REFDISTILLER to SafeRefactorImpact, we can see that RefDistiller's refactoring templates have different goal and approach, while SafeRefactorImpact identifies impacted elements that are used for guiding a testing generation process, we aim to verify whether the steps for successful refactoring are indeed performed. Finally, REFDISTILLER goes a step further than tools that use testing validation by pinpointing the location and cause of anomaly, such as *"Detected problematic binding in the reference `getStorageData(value)`, which may have affected the method `Store.updateExpiration()` - line 9"*.

7.4 Refactoring, Refactoring Validation and Tools

Refactoring edits are very common. Xing and Stroulia [124] find that 70% of structural changes developing the Eclipse IDE are refactorings. Those edits are mostly manually performed [79]. Negara et al.'s study [83] finds that experts still prefer manual refactorings than automated. However, refactoring is an error prone activity. Kim et al. [61] find there is correlation between the timing and location of refactorings and bug fixes. Weissgerber and Diehl [119] find that a high ratio of refactoring is often followed by an increasing number of bug reports. Most of the participants of Kim et al.'s field study [63] recognize that refactoring activities increase the changes of bug introduction.

Several works have proposed formal ways of formulating refactorings and reason about their correctness (*e.g. [88; 25; 76; 105; 106]*. For instance, Mens et al. [76] formulate refactorings as graph transformations and formalize certain aspects of a program's behavior as properties of the graph representing it. Schafer et al. [106] proposes a new representation of Java programs (JL) and a set of algorithms for translating Java programs into JL and vice versa. Thus, it is possible to formulate refactorings at the level of JL and to rely that translations are free of naming and accessibility issues. Although these formal verification ensure safe refactorings, due to their intrinsic complexity, they end up having short applicability in real projects. However, those work have inspired us greatly when defining the impact rules of RBA's refactoring fault models, and REFDISTILLER's refactoring templates.

Regression testing remains the most used strategy for checking refactoring correctness. However, Rachatasumrit and Kim [94] find that test suites are often inadequate and developers may hesitate to initiate or perform refactoring tasks due to inadequate test coverage [63]. Soares et al. [111] design and implement SafeRefactor that uses randomly generated test suites for detecting refactoring anomalies. It uses the Randoop tool to generate test cases for the impacted methods. REFDISTILLER' evaluation evidences that even tool generated tests can be inadequate. Using a SafeRefactor like testing validation we find that about 25% of the refactoring anomalies are not identified by using generated test suites, even with a long time limit for test generation (100 seconds).

GhostFactor [40] checks correctness of manual refactoring, similar to REFCHECKER. However, unlike REFSEPARATOR, GhostFactor does not have any capability to isolate potential behavior changes from pure refactoring by running an equivalent automated refactoring. GhostFactor detects missing edits only, while REFDISTILLER detects both missing and extra edits. Though we did not run GhostFactor for comparison, according to GhostFactor's algorithm, it supports only 3 refactoring types and detects only potential missing edits. Therefore, GhostFactor can detect 5 missing edits in our open source project data set, as opposed to 24 missing edits and 187 extra edits found by REFDISTILLER. Furthermore, GhostFactor can detect 3 missing edits from the transformations collected from Soares' work [109], as opposed to 34 correct missing edits found by REFDISTILLER.

Ge et al. [41] propose a refactoring-aware code review tool. This tool helps code reviewers examine pure refactorings in isolation from extra edits. Like REFDISTILLER, Ge et al. leverage Eclipse refactoring APIs to separate pure refactorings. REFDISTILLER goes a step further by extending Eclipse refactoring APIs to prevent unsafe refactoring by checking bug conditions. This allows REFDISTILLER to apply automated refactoring in a safe manner when isolating pure refactoring. In addition, REFDISTILLER also detect missing edits (incomplete refactorings) with concrete warning messages about how to fix the anomalies, such as *"Detected problematic binding in the reference `getStorageData(value)`"*.

Tsantalis and Chatzigeorgiou [115; 31] propose an approach and tool (JDeodorant) for identifying refactoring opportunities. They also introduces a set of rules regarding the preservation of existing dependences. Different from REFCHECKER's refactoring templates, JDeodorant's rules are designed to predict whether a possible future refactoring may change the behavior. On the other hand, REFCHECKER's templates check whether performed are according to expected when refactoring.

Ge et al. [38] and Foster at al. [32] detect manual refactoring, remind a developer that automated refactoring is available, and complete it automatically. While these refactoring completions tools leverage Eclipse refactoring APIs, REFDISTILLER differs from these by finding anomalies as opposed to auto-completing refactorings.

Chapter 8

Concluding Remarks

During software development and maintenance developers often face refactoring tasks. Those tasks are mostly manually executed, which may end up with the introduction of problems in a previous stable code. Detect and fix refactoring faults are usually mentioned as great challenges by developers [63]. Regression testing suites are commonly used for validating refactorings. Those suites work as safety nets in order to increase the confidence that a software's behavior remains untouched after refactoring. However, regression suites by itself may fall short when test cases are not designed for validation purposes. Moreover, large test suites may be ineffective as a delayed fault detection may not be acceptable due to resource limitations. For instance, in the context of our research laboratory there is a project with a test suite that takes around two days to execute. This doctorate work presents a novel test case prioritization technique (RBA [7]) that speeds up the detection of refactoring faults by using regression test suites. Moreover, this approach places near to each other test cases that often give useful information about a refactoring fault. We also propose a static analysis refactoring reviewing technique, REFDISTILLER [11]. This approach complements testing validation by searching for missing steps and extra edits that deviates from a pure refactoring transformation. As both solutions are complementary a developer might use one or both depending on the available resources and/or whether he trusts his test suite effectiveness.

Besides proposing RBA and REFDISTILLER, we performed an investigation on the role of testing coverage combined with a location based change impact analysis for detecting refactoring faults [8]. This study indicates that by focusing on a set of commonly impacted locations, a test suite are more likely to effectively detect refactoring faults. This conclusion motivated the development of the Refactoring Fault Models and Refactoring Templates, bases of RBA's prioritization algorithm and REFDISTILLER' fault detection process. Moreover, the study's conclusions can work as guidelines to developers to help them to evaluate theirs suite effectiveness and/or to guide them on detecting which code locations require more testing efforts.

The contributions of our work help developers to locate and fix refactoring faults. Our evaluations have shown that both RBA and REFDISTILLER improve the state of art in this sense by providing to developers useful information regarding fault location. RBA, in general, places refactoring fault revealing test cases near to each other. Analyze and understand a testing execution is not always easy. By grouping the refactoring related test cases a developer can focus his efforts on the test cases nearby and have a better understanding of the problem. RefDistiller, not only detects missing and extra edits, but pinpoints the possible location of the problem along with a description of the fault. This extra information can be used as starting point in order to find and fix the problem.

We believe that the proposed solutions can be integrated in order to provide to developers a better support when dealing with refactorings validation and/or revision. For instance, suppose a developer performs a refactoring manually and, although no compilation error is found, he decides to review it for double checking its correctness. Although the project he is working on has a test suite with thousands of test cases, he does not know if it is trustworthy for validating the recently performed edits. Moreover, even if the test suite is effective, running it entirely would be too costly and time consuming. To have an evaluation regarding the test suite effectiveness, the developer could use the results from the empirical study presented in Chapter 3 and check if there are test cases covering commonly affected code locations (the refactored method and its callers) combined with a high rate of branch coverage. If there are no test cases that fulfill these two requirements, the developer could guide efforts on augmenting the test suite in order to make it more efficient for validating the current and future refactorings. After making sure the test suite is effective for refactoring validation, the developer might use RBA for speeding up fault detection. Thus, any detectable refactoring fault would be early detected by the prioritized suite, saving time and resources. Moreover, the test cases that are most related to the edits are likely to be near to each other. When a refactoring fault is found by testing, but the test cases do not provide much help for locating the fault, or when the test suite does not detect any fault and the developer wants to continue his investigation, REFDISTILLER should be used. REFDISTILLER will perform a different analysis, searching for missing and extra edits. The developer can use REFDISTILLER's outputs for locating the fault detected by testing and/or for detecting new faults that might go undetected by testing. Thus, REFDISTILLER's outputs can also provide useful information that can be used to improve the project's test suite.

One of our main goals while conducting this doctorate work was to provide to developers the proper support for using the proposed solutions. For that, two Eclipse plug-ins were developed (Chapter 6): PriorJ [98], a prioritization tool for Java/JUnit systems that, besides other features, includes RBA as built-in prioritization technique; and RefDistiller [11], a tool that detects missing and extra edits in Java programs, and information to help to review a refactoring. The statistical model extracted from our study results (Section 3.2.3) enables the ease evaluation of a suite's capacity for validating a refactoring.

The challenges faced when developing PRIORJ forced us to provide new solutions that can be applied when developing/using prioritization in practice. We did not discuss these challenges and solutions in this documents, however, details about this side research can be seen in [10].

Although our evaluations show very promising results, our solutions still have limitations and could be improved. Thus, we can list several improvements planned as future work: i) refine our the model for evaluating a suite's effectiveness by investigating whether system's characteristics (*e.g.*, coupling and method cyclomatic complexity) are influential factors. We believe that by introducing new factors, the evaluative capacity of our model could be improved; ii) refine the refactoring fault models. The empirical studies on RBA's effectiveness indicate that we could generate a more accurate impacted set if the extraction rules were less name-based; iii) define fault models and refactoring templates for new refactoring types. Since our solutions are related to the specific edits of each refactoring, new fault models and refactoring templates related to new refactorings would amplify their use; iv) investigate how RBA and RefDistiller would behave when more than one refactoring is applied at once. To maximize their results and/or avoid false positives, we suggest that the current versions of our solutions should be used when a single refactoring is performed at time, which might not be the real refactoring scenario in practice. We plan to investigate the impact of combined refactorings and adapt our solutions to minimize this limitation; v) use RBA and/or REFDISTILLER's outputs to generate new test cases that could improve a suite's effectiveness for detecting future refactoring bugs. Both solutions provide interesting information that could feed a test generation algorithm. The new test cases would be incorporated to the previous regression suite to improve its capacity of validating future refactorings; and vi) develop a process that combines our solutions (suite evaluation + RBA + REFDISTILLER). This process would, for instance, define when and how to use each solution during the software development/maintenance phases. Moreover, we plan to evaluate the real gain of using RBA and REFDISTILLER, and their tools, in industrial projects, regarding their applicability and usability.

Bibliography

[1] Exploratory study website. `https://sites.google.com/site/coverageexperiment/`. Accessed: 2015-07-03.

[2] Rba website. `https://sites.google.com/a/computacao.ufcg.edu.br/rba/`. Accessed: 2015-07-03.

[3] M. Acharya and B. Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the 33rd international conference on software engineering*, pages 746–755. ACM, 2011.

[4] E.L.G. Alves. Investigating test case prioritization techniques for refactoring activities validation: Evaluating suite characterists. Technical Report SPLab-2012-002, Software Practices Laboratory, UFCG, August 2012. http://splab.computacao.ufcg.edu.br/technical-reports/.

[5] E.L.G. Alves. Investigating test case prioritization techniques for refactoring activities validation: Evaluating the behavior. Technical Report SPLab-2012-001, Software Practices Laboratory, UFCG, May 2012. http://splab.computacao.ufcg.edu.br/technical-reports/.

[6] E.L.G. Alves, P.D.L. Machado, and T Massoni. Prioritizing test cases for early detection of refactoring faults. In *Journal of Software: Testing, Verification and Reliability (Under final revision)*, pages –, 2015.

[7] E.L.G. Alves, P.D.L. Machado, T. Massoni, and S.T.C. Santos. A refactoring-based approach for test case selection and prioritization. In *Automation of Software Test (AST), 2013 8th International Workshop on*, pages 93–99. IEEE, 2013.

[8] E.L.G. Alves, T Massoni, and P.D.L. Machado. Test coverage and impact analysis for detecting refactoring faults: A study on the extract method refactoring. In *30th ACM/SIGAPP Symposium On Applied Computing - SAC 2015*, pages –, 2015.

[9] E.L.G. Alves, T Massoni, and P.D.L. Machado. Test coverage of impacted code elements for detecting refactoring faults, an exploratory study. In *The Journal of Systems and Software (Under revision)*, pages –, 2015.

[10] E.L.G. Alves, T.C.S. Santos, P.D.L. Machado, and Massoni T. Test case prioritization by using priorj. *Proceedings of 7th Brazilian Workshop on Systematic and Automated Software Testing (SAST, 2013)*, 2013.

[11] E.L.G. Alves, M. Song, and M. Kim. Refdistiller: A refactoring aware code review tool for inspecting manual refactoring edits. In *Proceedings of The 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Research Demonstration Track*. ACM, 2014.

[12] E.L.G. Alves, M. Song, M. Kim, P.D.L. Machado, and T. Massoni. Experiment Artifacts to RefDistiller: Detecting Anomalies in Manual Refactoring Edits. Technical report, The University of Texas at Austin, 2014. `http://users.ece.utexas.edu/~miryung/Publications/RefDistiller-TR.pdf`.

[13] G. Bavota, B. Carluccio, A. Lucia, M. Penta, R. Oliveto, and O. Strollo. When does a refactoring induce bugs? an empirical study. In *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*, pages 104–113. IEEE, 2012.

[14] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[15] K. Beck and C. Andres. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2004.

[16] R. V Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[17] D. Binkley. Reducing the cost of regression testing by semantics guided test case selection. In *Software Maintenance, 1995. Proceedings., International Conference on*, pages 251–260. IEEE, 1995.

[18] S.A. Bohner. Software change impact analysis. 1996.

[19] S.A. Bohner and R.S. Arnold. *Software Change Impact Analysis*, chapter An Introduction to Software Change Impact Analysis, pages 1–26. 1996.

[20] L. Briand and D. Pfahl. Using simulation for assessing the real impact of test coverage on defect coverage. In *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on*, pages 148–157. IEEE, 1999.

[21] L.C. Briand, Y. Labiche, and L. O'sullivan. Impact analysis and change management of uml models. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 256–265. IEEE, 2003.

[22] X. Cai and M.R. Lyu. The effect of code coverage on fault detection under different testing profiles. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–7. ACM, 2005.

[23] C. Catal and D. Mishra. Test case prioritization: a systematic mapping study. *Software Quality Journal*, 21(3):445–478, 2013.

[24] P. K. Chittimalli and M. J. Harrold. Recomputing coverage information to assist regression testing. *Software Engineering, IEEE Transactions on*, 35(4):452–469, 2009.

[25] M. Cornélio, A. Cavalcanti, and A. Sampaio. Sound refactorings. *Science of Computer Programming*, 75(3):106–133, 2010.

[26] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *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. ACM, 2007.

[27] H. Do, S. Elbaum, and G. Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *Empirical Software*

Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on, pages 60–70. IEEE, 2004.

[28] S. Elbaum, A.G. Malishevsky, and G. Rothermel. *Prioritizing test cases for regression testing*, volume 25. ACM, 2000.

[29] S. Elbaum, A.G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *Software Engineering, IEEE Transactions on*, 28(2):159–182, 2002.

[30] B. Fluri, M. Wursch, M. Pinzger, and H.C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *Software Engineering, IEEE Transactions on*, 33(11):725–743, 2007.

[31] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. Identification and application of extract class refactorings in object-oriented systems. *Journal of Systems and Software*, 85(10):2241–2260, 2012.

[32] S.R. Foster, W.G. Griswold, and S. Lerner. Witchdoctor: Ide support for real-time auto-completion of refactorings. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 222–232. IEEE Press, 2012.

[33] M Fowler, K Beck, J Brant, W Opdyke, and D Roberts. Refactoring: Improving the design of existing programs, 1999.

[34] P.G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *ACM SIGSOFT Software Engineering Notes*, volume 23, pages 153–162. ACM, 1998.

[35] P.G. Frankl and S.N. Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proceedings of the symposium on Testing, analysis, and verification*, pages 154–164. ACM, 1991.

[36] P.G. Frankl, S.N. Weiss, and C. Hu. All-uses vs mutation testing: an experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3):235–253, 1997.

[37] A. Gargantini, M. Guarnieri, and E. Magri. Extending coverage criteria by evaluating their robustness to code structure changes. In *Testing Software and Systems*, pages 168–183. Springer, 2012.

[38] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 211–221. IEEE, 2012.

[39] X. Ge and E. Murphy-Hill. Manual refactoring changes with automated refactoring validation. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1095–1105. ACM, 2014.

[40] X. Ge and E. Murphy-Hill. Manual refactoring changes with automated refactoring validation. In *Software Engineering (ICSE 2014) 36th International Conference on*. IEEE, 2014.

[41] X. Ge, S. Sarkar, and E. Murphy-Hill. Towards refactoring-aware code review. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 99–102. ACM, 2014.

[42] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov. Systematic testing of refactoring engines on real software projects. In *ECOOP 2013 - Object-Oriented Programming*, pages 629–653. Springer, 2013.

[43] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. Amin Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 302–313. ACM, 2013.

[44] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in udita. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 225–234. ACM, 2010.

[45] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. *ACM SIGPLAN Notices*, 32(10):108–124, 1997.

[46] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei. A unified test case prioritization approach. *ACM Trans. Softw. Eng. Methodol.*, 24(2):10:1–10:31, December 2014.

[47] M.J. Harrold and A. Orso. Retesting software during development and maintenance. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 99–108. IEEE, 2008.

[48] D.W. Hosmer, S. Lemeshow, and R.X. Sturdivant. *Introduction to the logistic regression model*. Wiley Online Library, 2000.

[49] L. Huang and Y. Song. Precise dynamic impact analysis with dependency analysis for object-oriented programs. In *Software Engineering Research, Management & Applications, 2007. SERA 2007. 5th ACIS International Conference on*, pages 374–384. IEEE, 2007.

[50] S. Ibrahim, N.B. Idris, M. Munro, and A. Deraman. A software traceability validation for change impact analysis of object oriented software. In *Software Engineering Research and Practice*, pages 453–459, 2006.

[51] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Automated Software Engineering, 2008. 23rd IEEE/ACM International Conference on*, pages 297–306. IEEE, 2008.

[52] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *International Conference on Software Engineering*, pages 435–445, 2014.

[53] R. Jain. *The art of computer systems performance analysis*, volume 182. John Wiley & Sons Chichester, 1991.

[54] D. Jeffrey and R. Gupta. Test case prioritization using relevant slices. In *Computer Software and Applications Conference, 2006. COMPSAC'06. 30th Annual International*, volume 1, pages 411–420. IEEE, 2006.

[55] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse. Adaptive random test case prioritization. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 233–244. IEEE, 2009.

[56] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. Atl: a qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720. ACM, 2006.

[57] R. Just, D. Jalali, L. Inozemtseva, M.D Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*. ACM, 2014.

[58] S.A Kazarlis, A.G. Bakirtzis, and V. Petridis. A genetic algorithm solution to the unit commitment problem. *Power Systems, IEEE Transactions on*, 11(1):83–92, 1996.

[59] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An overview of aspectj. In *ECOOP 2001 - Object-Oriented Programming*, pages 327–354. Springer, 2001.

[60] J.M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 119–129. IEEE, 2002.

[61] M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 151–160. ACM, 2011.

[62] 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*, pages 371–372. ACM, 2010.

[63] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 50. ACM, 2012.

[64] B. Korel, L.H. Tahat, and M. Harman. Test prioritization using system models. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 559–568. IEEE, 2005.

[65] B. Korel, L.H. Tahat, and B. Vaysburg. Model based regression test reduction using dependence analysis. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 214–223. IEEE, 2002.

[66] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 308–318. IEEE, 2003.

[67] Y.Y. Lee, N. Chen, and R.E Johnson. Drag-and-drop refactoring: intuitive and efficient program transformation. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 23–32. IEEE Press, 2013.

[68] H. K. N. Leung and L. White. Insights into regression testing [software testing]. In *Software Maintenance, 1989., Proceedings., Conference on*, pages 60–69. IEEE, 1989.

[69] Z. Li, M. Harman, and R.M. Hierons. Search algorithms for regression test case prioritization. *Software Engineering, IEEE Transactions on*, 33(4):225–237, 2007.

[70] M.C.O Maia, R.A. Bittencourt, J.C.A. de Figueiredo, and D.D.S. Guerrero. The hybrid technique for object-oriented software change impact analysis. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 252–255. IEEE, 2010.

[71] A.G. Malishevsky, J.R Ruthruff, G. Rothermel, and S. Elbaum. Cost-cognizant test case prioritization. *Department of Computer Science and Engineering, University of Nebraska-Lincoln, Techical Report*, 2006.

[72] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel. A static approach to prioritizing junit test cases. *Software Engineering, IEEE Transactions on*, 38(6):1258–1275, 2012.

[73] L. Mei, W.K. Chan, T.H. Tse, and R.G. Merkel. Xml-manipulating test case prioritization for xml-manipulating services. *Journal of Systems and Software*, 84(4):603–619, 2011.

[74] L. Mei, Z. Zhang, W. K. Chan, and T. H. Tse. Test case prioritization for regression testing of service-oriented business applications. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 901–910, New York, NY, USA, 2009. ACM.

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

[76] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[77] M. Mongiovi, R. Gheyi, G. Soares, L. Teixeira, and P. Borba. Making refactoring safer through impact analysis. *Science of Computer Programming*, 93:39–64, 2014.

[78] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi. A case study on the impact of refactoring on quality and productivity in an agile team. In *Balancing Agility and Formalism in Software Engineering*, pages 252–266. Springer, 2008.

[79] G.C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the elipse ide? *Software, IEEE*, 23(4):76–83, 2006.

[80] E. Murphy-Hill and A. P. Black. Breaking the barriers to successful refactoring. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 421–430. IEEE, 2008.

[81] E. Murphy-Hill, C. Parnin, and A.P Black. How we refactor, and how we know it. *Software Engineering, IEEE Transactions on*, 38(1):5–18, 2012.

[82] D.A. Naumann, A. Sampaio, and L. Silva. Refactoring and representation independence for class hierarchies. pages 60–97, 2012.

[83] S. Negara, N. Chen, M. Vakilian, R. E Johnson, and D. Dig. A comparative study of manual and automated refactorings. In *ECOOP 2013–Object-Oriented Programming*, pages 552–576. Springer, 2013.

[84] S. Negara, M. Vakilian, N. Chen, R. E Johnson, and D. Dig. Is it dangerous to use version control histories to study source code evolution? In *ECOOP 2012–Object-Oriented Programming*, pages 79–103. Springer, 2012.

[85] A. K. Onoma, W. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Communications of the ACM*, 41(5):81–86, 1998.

[86] W.F. Opdyke. *Refactoring: A program restructuring aid in designing object-oriented application frameworks*. PhD thesis, PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[87] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M.J. Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the 26th International Conference on Software Engineering*, pages 491–500. IEEE Computer Society, 2004.

[88] J. L Overbey, M. J. Fotzler, A. J. Kasza, and R. E. Johnson. A collection of refactoring specifications for fortran 95. In *ACM SIGPLAN Fortran Forum*, volume 29, pages 11–25. ACM, 2010.

[89] C. Pacheco and M.D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816. ACM, 2007.

[90] H. Park, H. Ryu, and J. Baik. Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing. In *Secure System Integration and Reliability Improvement, 2008. SSIRI'08. Second International Conference on*, pages 39–46. IEEE, 2008.

[91] W.E. Perry. *Effective Methods for Software Testing: Includes Complete Guidelines, Checklists, and Templates*. John Wiley & Sons, 2006.

[92] S.L. Pfleeger and S.A Bohner. A framework for software maintenance metrics. In *Software Maintenance, 1990, Proceedings., Conference on*, pages 320–327. IEEE, 1990.

[93] 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. IEEE, 2010.

[94] N. Rachatasumrit and M. Kim. An empirical investigation into the impact of refactoring on regression testing. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 357–366. IEEE, 2012.

[95] M.K. Ramanathan, M. Koyuturk, A. Grama, and S. Jagannathan. Phalanx: a graph-theoretic framework for test case prioritization. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 667–673, New York, NY, USA, 2008. ACM.

[96] X. Ren, B.G. Ryder, M. Stoerzer, and F. Tip. Chianti: a change impact analysis tool for java programs. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 664–665. IEEE, 2005.

[97] X. Ren, F. Shah, F. Tip, B.G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *ACM Sigplan Notices*, volume 39, pages 432–448. ACM, 2004.

[98] J.H Rocha, E.L.G Alves, and P.D.L. Machado. Priorj - priorizacao automatica de casos de teste junit. *Proceedings of Third Brazilian Conference on Software: Theory and Practice (CBSoft) - Tools Section*, 4:43–50, 2012.

[99] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929–948, 2001.

[100] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold. Test case prioritization: An empirical study. In *Software Maintenance (ICSM'99). 1999. Proceedings. IEEE International Conference on*, pages 179–188. IEEE, 1999.

[101] M.J. Rummel. Towards the prioritization of regression test suites with data flow information. In *In Proceedings of the 20th Symposium on Applied Computing*. ACM Press, 2005.

[102] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53. ACM, 2001.

[103] A.B. Sánchez, S. Segura, and A. Ruiz-Cortés. A comparison of test case prioritization criteria for software product lines. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, pages 41–50. IEEE, 2014.

[104] M. Schaefer and O. De Moor. Specifying and implementing refactorings. In *ACM Sigplan Notices*, volume 45, pages 286–301. ACM, 2010.

[105] M. Schäfer, T. Ekman, and O. De Moor. Sound and extensible renaming for java. *ACM Sigplan Notices*, 43(10):277–294, 2008.

[106] M. Schafer, A. Thies, F. Steimann, and F. Tip. A comprehensive approach to naming and accessibility in refactoring java programs. *Software Engineering, IEEE Transactions on*, 38(6):1233–1257, 2012.

[107] Y. Singh, A. Kaur, B. Suri, and S. Singhal. Systematic literature review on regression test prioritization techniques. *Special Issue: Advances in Network Systems Guest Editors: Andrzej Chojnacki*, page 379.

[108] G. Soares, B. Catao, C. Varjao, S. Aguiar, R. Gheyi, and T. Massoni. Analyzing refactorings on software repositories. In *Software Engineering (SBES), 2011 25th Brazilian Symposium on*, pages 164–173. IEEE, 2011.

[109] G. Soares, R. Gheyi, and T. Massoni. Automated behavioral testing of refactoring engines. *Software Engineering, IEEE Transactions on*, 39(2):147–162, 2013.

[110] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson. Comparing approaches to analyze refactoring activity on software repositories. *Journal of Systems and Software*, 86(4):1006–1022, 2013.

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

[112] H. Srikanth, L. Williams, and J. Osborne. System test case prioritization of new and regression test cases. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, pages 10 pp.–, 2005.

[113] H. Srikanth, L. Williams, and J. Osborne. System test case prioritization of new and regression test cases. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, pages 10 pp.–, 2005.

[114] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 97–106. ACM, 2002.

[115] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *Software Engineering, IEEE Transactions on*, 35(3):347–367, 2009.

[116] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 233–243. IEEE, 2012.

[117] M Vakilian, N. Chen, S. Negara, B.A. Rajkumar, R. Zilouchian Moghaddam, and R.E. Johnson. The need for richer refactoring usage data. In *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*, pages 31–38. ACM, 2011.

[118] K.R. Walcott, M.L. Soffa, G.M. Kapfhammer, and R.S. Roos. Timeaware test suite prioritization. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 1–12. ACM, 2006.

[119] P. Weißgerber and S. Diehl. Are refactorings less error-prone than other changes? In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 112–118. ACM, 2006.

[120] J. Wloka, E. Hoest, and B.G. Ryder. Tool support for change-centric test development. *Software, IEEE*, 27(3):66–71, 2010.

[121] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer, 2012.

[122] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*, pages 264–274. IEEE, 1997.

[123] W.E. Wong, J.R. Horgan, S. London, and A.P. Mathur. Effect of test set size and block coverage on the fault detection effectiveness. In *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, pages 230–238. IEEE, 1994.

[124] Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65. ACM, 2005.

[125] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 192–201. IEEE, 2013.

[126] L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 23–32. IEEE, 2011.

[127] L. Zhang, M. Kim, and S. Khurshid. Faulttracer: A change impact and regression fault analysis tool for evolving java programs. 2012.

[128] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P Hudepohl, and M.A. Vouk. On the value of static analysis for fault detection in software. *Software Engineering, IEEE Transactions on*, 32(4):240–253, 2006.

Appendix A

Exploratory Study Results

This Appendix section presents the tables (Tables A.1 A.2 A.3 A.4) containing the detailed results of our exploratory study.

Each faulty version is represented according to the pattern *"project version / fault version / type of fault"*. For instance *xv1/f1/em+ds* means XML-Security version 1, first created faulty version in which there is an extract method edit with a deleted statement fault.

Table A.1: Study Results for the versions with an extracted method and a delete statement fault. An x indicates that the code element does not exist in this case (e.g. a method that does not have callers). ID: version/seeded fault; TC: number of test cases; FTC: percentage of failed test cases; D_M: test cases M that failed; D_C: test cases from C that failed; D_{Ce}: test cases from Ce that failed; D_O: test cases from O that failed.

XML-Security											**JMock**										
ID	**TC**	**FTC**	**M**	D_M	**C**	D_C	**Ce**	D_{Ce}	**O**	D_o	**ID**	**TC**	**FTC**	**M**	D_M	**C**	D_C	**Ce**	D_{Ce}	**O**	D_o
xv1/f1/em+ds	88	0	0	0	0	0	✗	✗	✗	✗	jv1/f1/em+ds	195	1	74	2	0	0	76	2	✗	✗
xv1/f2/em+ds	88	0	0	0	0	0	0	0	✗	✗	jv1/f2/em+ds	195	1	62	3	60	3	0	0	✗	✗
xv1/f3/em+ds	88	3	0	0	0	0	3	2	✗	✗	jv1/f3/em+ds	195	3	10	7	62	0	12	7	✗	✗
xv1/f4/em+ds	88	0	0	0	0	0	0	0	✗	✗	jv1/f4/em+ds	195	0	58	0	72	0	103	0	✗	✗
xv1/f5/em+ds	88	17	10	10	10	10	0	0	✗	✗	jv1/f5/em+ds	195	0	0	0	83	0	✗	✗	✗	✗
xv2/f1/em+ds	101	0	0	0	0	0	30	0	0	0	jv2/f1/em+ds	222	0.4	57	3	57	3	53	3	✗	✗
xv2/f2/em+ds	101	0	0	0	0	0	✗	✗	✗	✗	jv2/f2/em+ds	222	0.4	3	0	6	0	✗	✗	✗	✗
xv2/f3/em+ds	101	3	3	3	3	3	53	1	✗	✗	jv2/f3/em+ds	222	0.01	1	1	1	1	✗	✗	✗	✗
xv2/f4/em+ds	101	0	0	0	0	0	7	0	0	0	jv2/f4/em+ds	222	0	0	0	0	0	0	0	✗	✗
xv2/f5/em+ds	101	4	26	3	26	3	✗	✗	✗	✗	jv2/f5/em+ds	222	0.01	3	0	76	2	2	0	✗	✗
xv3/f1/em+ds	103	0	36	0	36	0	0	0	✗	✗	jv3/f1/em+ds	202	2	5	5	5	5	37	3	✗	✗
xv3/f2/em+ds	103	13	10	9	10	9	36	12	✗	✗	jv3/f2/em+ds	202	1	16	2	7	1	✗	✗	✗	✗
xv3/f3/em+ds	103	0	0	0	✗	✗	26	0	✗	✗	jv3/f3/em+ds	202	1	3	2	1	1	✗	✗	✗	✗
xv3/f4/em+ds	103	0	0	0	0	0	29	0	0	0	jv3/f4/em+ds	202	0	15	0	15	0	✗	✗	✗	✗
xv3/f5/em+ds	103	3	5	3	17	3	22	3	✗	✗	jv3/f5/em+ds	202	3	2	2	✗	✗	3	1	✗	✗

EasyAccept										
ID	**TC**	**FTC**	**M**	D_M	**C**	D_C	**Ce**	D_{Ce}	**O**	D_o
ev1/f1/em+ds	65	0	3	0	11	0	29	0	✗	✗
ev1/f2/em+ds	65	6	12	4	✗	✗	✗	✗	✗	✗
ev1/f3/em+ds	65	1.5	11	1	11	1	9	0	✗	✗
ev1/f4/em+ds	65	6	7	4	23	4	35	4	✗	✗
ev1/f5/em+ds	65	0	1	0	23	0	36	0	✗	✗
ev2/f1/em+ds	67	0	4	0	4	0	31	0	✗	✗
ev2/f2/em+ds	67	0	2	0	25	0	31	0	✗	✗
ev2/f3/em+ds	67	0	27	0	4	0	37	0	✗	✗
ev2/f4/em+ds	67	1.5	6	1	6	1	26	1	✗	✗
ev2/f5/em+ds	67	1.5	1	1	4	0	26	1	✗	✗
ev3/f1/em+ds	74	9	10	7	✗	✗	✗	✗	✗	✗
ev3/f2/em+ds	74	2	44	2	44	2	✗	✗	7	2
ev3/f3/em+ds	74	8	26	6	25	6	36	6	✗	✗
ev3/f4/em+ds	74	5	6	4	6	4	✗	✗	✗	✗
ev3/f5/em+ds	74	0	2	0	12	0	✗	✗	✗	✗

Table A.2: Study Results for the versions with an extracted method and a relational operator replacement fault.

XML-Security											JMock										
ID	**TC**	**FTC**	**M**	D_M	**C**	D_C	**Ce**	D_{Ce}	**O**	D_o	**ID**	**TC**	**FTC**	**M**	D_M	**C**	D_C	**Ce**	D_{Ce}	**O**	D_o
xv1/f1/em+ror	88	3	3	3	3	3	0	0	16	3	jv1/f1/em+ror	195	0.5	0	0	3	1	✗	✗	✗	✗
xv1/f2/em+ror	88	0	0	0	✗	✗	0	0	✗	✗	jv1/f2/em+ror	195	5	4	4	4	4	✗	✗	✗	✗
xv1/f3/em+ror	88	0	0	0	0	0	✗	✗	✗	✗	jv1/f3/em+ror	195	2	6	6	59	6	8	6	✗	✗
xv1/f4/em+ror	88	0	0	0	0	0	0	0	✗	✗	jv1/f4/em+ror	195	2	5	4	2	1	5	4	✗	✗
xv1/f5/em+ror	88	14	19	12	19	12	26	12	✗	✗	jv1/f5/em+ror	195	2	3	2	7	2	✗	✗	✗	✗
xv2/f1/em+ror	101	0	0	0	11	0	✗	✗	✗	✗	jv2/f1/em+ror	222	0.4	0	0	2	0	✗	✗	✗	✗
xv2/f2/em+ror	101	1	19	1	19	1	✗	✗	✗	✗	jv2/f2/em+ror	222	21	47	46	47	46	✗	✗	✗	✗
xv2/f3/em+ror	101	0	0	0	0	0	✗	✗	✗	✗	jv2/f3/em+ror	222	41	91	91	91	91	✗	✗	✗	✗
xv2/f4/em+ror	101	1	1	1	1	1	1	1	✗	✗	jv2/f4/em+ror	222	37	91	85	0	0	99	85	✗	✗
xv2/f5/em+ror	101	3	3	3	3	3	10	3	0	0	jv2/f5/em+ror	222	54	0	0	9	4	✗	✗	✗	✗
xv3/f1/em+ror	103	0	3	0	3	0	✗	✗	✗	✗	jv3/f1/em+ror	202	1	0	0	1	0	4	0	✗	✗
xv3/f2/em+ror	103	6	7	5	54	6	✗	✗	✗	✗	jv3/f2/em+ror	202	4	45	8	42	4	45	8	✗	✗
xv3/f3/em+ror	103	0	0	0	0	0	✗	✗	✗	✗	jv3/f3/em+ror	202	1	30	0	✗	✗	34	0	✗	✗
xv3/f4/em+ror	103	0	0	0	0	0	28	0	✗	✗	jv3/f4/em+ror	202	3	0	0	0	0	✗	✗	✗	✗
xv3/f5/em+ror	103	1	1	1	1	1	31	1	✗	✗	jv3/f5/em+ror	202	1	10	2	1	0	7	2	✗	✗

EasyAccept										
ID	**TC**	**FTC**	**M**	D_M	**C**	D_C	**Ce**	D_{Ce}	**O**	D_o
ev1/f1/em+ror	65	0	11	0	✗	✗	35	0	✗	✗
ev1/f2/em+ror	65	7	11	7	11	7	9	6	✗	✗
ev1/f3/em+ror	65	1	1	1	23	0	35	1	✗	✗
ev1/f4/em+ror	65	1	1	1	23	1	36	1	✗	✗
ev1/f5/em+ror	65	5	43	3	43	3	✗	✗	✗	✗
ev2/f1/em+ror	67	6	5	4	0	0	✗	✗	✗	✗
ev2/f2/em+ror	67	0	2	0	2	0	2	0	✗	✗
ev2/f3/em+ror	67	1	1	1	25	0	37	1	✗	✗
ev2/f4/em+ror	67	7	30	5	30	5	37	5	✗	✗
ev2/f5/em+ror	67	4	46	3	46	3	✗	✗	✗	✗
ev3/f1/em+ror	74	12	10	9	10	7	✗	✗	0	0
ev3/f2/em+ror	74	9	9	7	46	7	✗	✗	✗	✗
ev3/f3/em+ror	74	0	0	0	45	0	✗	✗	✗	✗
ev3/f4/em+ror	74	1	55	1	54	0	65	1	✗	✗
ev3/f5/em+ror	74	5	14	4	14	4	✗	✗	✗	✗

Table A.3: Study Results for the versions with a move method and a delete statement fault.

XML-Security											JMock										
ID	**TC**	**FTC**	**M**	D_M	**C**	D_C	**Ce**	D_{Ce}	**O**	D_o	**ID**	**TC**	**FTC**	**M**	D_M	**C**	D_C	**Ce**	D_{Ce}	**O**	D_o
xv1/f1/mm+ds	88	0	0	0	✗	✗	✗	✗	✗	✗	jv1/f1/mm+ds	195	1	8	2	✗	✗	8	2	✗	✗
xv1/f2/mm+ds	88	11	33	10	✗	✗	✗	✗	✗	✗	jv1/f2/mm+ds	195	3	26	6	25	5	0	0	✗	✗
xv1/f3/mm+ds	88	0	0	0	0	0	0	0	✗	✗	jv1/f3/mm+ds	195	0	1	0	3	0	✗	✗	✗	✗
xv1/f4/mm+ds	88	0	23	0	23	0	✗	✗	✗	✗	jv1/f4/mm+ds	195	1	64	2	66	2	68	2	✗	✗
xv1/f5/mm+ds	88	0	0	0	0	0	0	0	✗	✗	jv1/f5/mm+ds	195	5	54	9	54	9	✗	✗	✗	✗
xv2/f1/mm+ds	101	10	0	0	11	8	✗	✗	✗	✗	jv2/f1/mm+ds	222	0.4	7	1	7	1	✗	✗	✗	✗
xv2/f2/mm+ds	101	10	0	0	11	8	10	10	✗	✗	jv2/f2/mm+ds	222	1	4	3	7	3	✗	✗	✗	✗
xv2/f3/mm+ds	101	0	0	0	0	0	0	0	✗	✗	jv2/f3/mm+ds	222	0.4	3	1	8	1	✗	✗	✗	✗
xv2/f4/mm+ds	101	0	0	0	0	0	10	0	✗	✗	jv2/f4/mm+ds	222	0	0	0	0	0	✗	✗	✗	✗
xv2/f5/mm+ds	101	5	1	1	13	1	23	1	21	1	jv2/f5/mm+ds	222	1	14	3	11	3	✗	✗	✗	✗
xv3/f1/mm+ds	103	1	26	1	26	1	35	1	✗	✗	jv3/f1/mm+ds	202	0	3	0	3	0	3	0	✗	✗
xv3/f2/mm+ds	103	28	26	26	26	26	26	26	✗	✗	jv3/f2/mm+ds	202	1	38	0	0	0	39	0	✗	✗
xv3/f3/mm+ds	103	0	26	0	26	0	49	0	✗	✗	jv3/f3/mm+ds	202	0	34	0	0	0	36	0	✗	✗
xv3/f4/mm+ds	103	0	0	0	13	0	3	0	✗	✗	jv3/f4/mm+ds	202	3	0	0	0	0	0	0	✗	✗
xv3/f5/mm+ds	103	3	3	3	0	0	3	3	3	3	jv3/f5/mm+ds	202	0.5	10	1	5	0	7	1	✗	✗

EasyAccept										
ID	**TC**	**FTC**	**M**	D_M	**C**	D_C	**Ce**	D_{Ce}	**O**	D_o
ev1/f1/mm+ds	65	0	0	0	11	0	29	0	✗	✗
ev1/f2/mm+ds	65	5	11	3	11	3	9	1	✗	✗
ev1/f3/mm+ds	65	1	1	1	23	0	9	1	✗	✗
ev1/f4/mm+ds	65	1	1	1	1	1	2	1	✗	✗
ev1/f5/mm+ds	65	0	1	0	23	0	29	0	✗	✗
ev2/f1/mm+ds	67	7	6	5	0	0	46	5	✗	✗
ev2/f2/mm+ds	67	13	46	9	46	9	46	9	✗	✗
ev2/f3/mm+ds	67	1	58	1	58	1	✗	✗	✗	✗
ev2/f4/mm+ds	67	7	21	5	20	4	31	5	✗	✗
ev2/f5/mm+ds	67	16	38	11	38	11	38	11	✗	✗
ev3/f1/mm+ds	74	0	35	0	58	0	35	0	✗	✗
ev3/f2/mm+ds	74	1	1	1	1	1	✗	✗	✗	✗
ev3/f3/mm+ds	74	0	0	0	✗	✗	✗	✗	0	0
ev3/f4/mm+ds	74	9	10	7	✗	✗	✗	✗	✗	✗
ev3/f5/mm+ds	74	9	46	7	46	7	46	7	✗	✗

Table A.4: Study Results for the versions with a move method and a relational operator replacement fault.

XML-Security											JMock										
ID	**TC**	**FTC**	**M**	D_M	**C**	D_C	**Ce**	D_{Ce}	**O**	D_o	**ID**	**TC**	**FTC**	**M**	D_M	**C**	D_C	**Ce**	D_{Ce}	**O**	D_o
xv1/f1/mm+ror	88	0	10	0	10	0	✗	✗	✗	✗	jv1/f1/mm+ror	195	2	5	4	5	4	✗	✗	✗	✗
xv1/f2/mm+ror	88	11	10	10	✗	✗	✗	✗	✗	✗	jv1/f2/mm+ror	195	0.5	6	1	7	1	7	1	✗	✗
xv1/f3/mm+ror	88	0	6	0	0	0	7	0	✗	✗	jv1/f3/mm+ror	195	0	3	0	7	0	✗	✗	✗	✗
xv1/f4/mm+ror	88	1	1	1	1	1	1	1	✗	✗	jv1/f4/mm+ror	195	29	62	57	60	57	60	57	✗	✗
xv1/f5/mm+ror	88	0	0	0	3	0	0	0	✗	✗	jv1/f5/mm+ror	195	18	36	36	62	36	✗	✗	✗	✗
xv2/f1/mm+ror	101	0	0	0	7	0	10	0	✗	✗	jv3/f1/mm+ror	222	7	15	13	29	13	✗	✗	✗	✗
xv2/f2/mm+ror	101	6	6	6	7	6	7	6	0	0	jv3/f2/mm+ror	222	35	90	77	89	77	✗	✗	✗	✗
xv2/f3/mm+ror	101	0	0	0	0	0	11	0	✗	✗	jv3/f3/mm+ror	222	1	13	2	5	0	0	0	✗	✗
xv2/f4/mm+ror	101	12	6	6	6	6	✗	✗	✗	✗	jv3/f4/mm+ror	222	1	13	3	5	0	0	0	✗	✗
xv2/f5/mm+ror	101	0	0	0	0	0	0	0	✗	✗	jv3/f5/mm+ror	222	0	7	0	7	0	13	0	✗	✗
xv3/f1/mm+ror	103	1	1	1	27	1	✗	✗	✗	✗	jv2/f1/mm+ror	202	8	39	13	38	13	✗	✗	✗	✗
xv3/f2/mm+ror	103	3	39	3	39	3	40	1	✗	✗	jv2/f2/mm+ror	202	0.5	10	1	1	0	7	0	✗	✗
xv3/f3/mm+ror	103	0	0	0	16	0	✗	✗	✗	✗	jv2/f3/mm+ror	202	1	7	2	0	0	✗	✗	✗	✗
xv3/f4/mm+ror	103	0	0	0	0	0	✗	✗	✗	✗	jv2/f4/mm+ror	202	3	7	7	43	7	✗	✗	✗	✗
xv3/f5/mm+ror	103	0	0	0	0	0	✗	✗	✗	✗	jv2/f5/mm+ror	202	1	4	3	1	1	4	3	✗	✗

EasyAccept										
ID	**TC**	**FTC**	**M**	D_M	**C**	D_C	**Ce**	D_{Ce}	**O**	D_o
ev1/f1/mm+ror	65	0	11	0	11	0	9	0	✗	✗
ev1/f2/mm+ror	65	6	5	4	11	4	✗	✗	✗	✗
ev1/f3/mm+ror	65	1	1	1	1	1	2	1	✗	✗
ev1/f4/mm+ror	65	0	1	0	1	0	2	0	✗	✗
ev1/f5/mm+ror	65	0	1	0	1	0	2	0	✗	✗
ev2/f1/mm+ror	67	24	19	16	22	16	✗	✗	✗	✗
ev2/f2/mm+ror	67	33	27	22	7	3	✗	✗	✗	✗
ev2/f3/mm+ror	67	36	30	24	30	24	37	24	✗	✗
ev2/f4/mm+ror	67	28	20	19	20	19	38	19	✗	✗
ev2/f5/mm+ror	67	12	46	8	46	8	46	8	✗	✗
ev3/f1/mm+ror	74	4	54	3	54	3	✗	✗	✗	✗
ev3/f2/mm+ror	74	1	1	1	1	1	✗	✗	✗	✗
ev3/f3/mm+ror	74	12	10	9	10	9	✗	✗	0	0
ev3/f4/mm+ror	74	20	46	15	46	15	46	15	✗	✗
ev3/f5/mm+ror	74	8	28	6	27	6	38	9	✗	✗

Appendix B

Refactoring Fault Models

This Appendix section presents the refactoring fault models for the remaining refactoring types: move method, pull up field and add parameter.

```
MoveMethodRFM (C1: class, C2: class, m: Method)
     BEGIN
          AS <- Set {};
          AS.add ( m );
        // get methods with similar signature
          AS.addAll ( searchMethodsWithSameName (C1, m.name));
        AS.addAll (searchMethodCalls (m, C1) );
        // get methods that access or modify the same fields and/or variables used by m
          AS.addAll (searchMethodsThatAccessOrModifySameFields (C1, m) );
          // get methods that have method call with parameters related to any field/
    variable modified by m
        AS.addAll (searchMethodCallsThatUseParametersfromM (C1, m);

        AS.addAll (searchMethodCalls (m, C2) );
          AS.addAll (searchMethodsThatAccessOrModifySameFields (C2, m) );
          AS.addAll (searchMethodCallsThatUseParametersfromM (C2, m);

          subClasses <- getSubClasses (C1);
          FOREACH S in subClasses DO
               AS.addAll (searchMethodCalls (m, S) );
               AS.addAll (searchMethodsThatAccessOrModifySameFields (m, S) );

        subClasses <- getSubClasses (C2);
          FOREACH S in subClasses DO
               AS.addAll (searchMethodCalls (m, S) );
               AS.addAll (searchMethodsThatAccessOrModifySameFields (m, S) );
          END FOREACH

          superClasses <- getSuperClasses (C1);
        FOREACH Sp in subClasses DO
             AS.addAll (searchMethodsWithSameName (Sp, m.name) );
        END FOREACH
      superClasses <- getSuperClasses (C2);
        FOREACH Sp in subClasses DO
             AS.addAll (searchMethodsWithSameName (Sp, m.name) );
        END FOREACH

        IF m is #static THEN
             allClasses <- getAllClasses();
             FOREACH C in allClasses DO
                  AS.addAll (searchMethodCalls (oldName, C) );
                              AS.addAll (searchMethodsThatAccessOrModifySameFields (m,
   C) );
             END FOREACH
        END IF
        return AS;
   END
```

Move method refactoring fault model.

```
PullUPFieldRFM (Cs: class, C: class, f: field)
   BEGIN
        AS <- Set {};
        methsC <- searchMethodsThatAccessOrModifyField (C, f)
      AS.addAll (methsC);
      AS.addAll (searchMethodsThatAccessOrModifyVariable (Cs, f));

        FOREACH m in methsC DO
             AS.addAll (searchMethodsWithSameName (m.name, Cs) );
        END FOREACH

      subClasses <- getSubClasses (Cs);
        FOREACH S in subClasses DO
             AS.addAll (searchMethodsThatAccessOrModifyVariable (S, f) );
             FOREACH m in methsC DO
                 AS.addAll (searchMethodsWithSameName (m.name, S) );
                END FOREACH
        END FOREACH
        return AS;
   END
```

Pull up field refactoring fault model.

```
AddParameterRFM (C: class, m: method, p: parameter)
     BEGIN
          AS <- Set {};
          AS.add ( m );
          AS.addAll (searchMethodsWithSameName (m.name, C) );

          subClasses <- getSubClasses (C);
          FOREACH S in subClasses DO
               AS.addAll (searchMethodCalls (m.name, S) );
          END FOREACH

          IF oldMethod is #static THEN
               allClasses <- getAllClasses();
               FOREACH C in allClasses DO
                    AS.addAll (searchMethodCalls (m.name, C) );
               END FOREACH
          END IF
          return AS;
     END
```

Add parameter refactoring fault model.

Appendix C

Remaining Refactoring Templates

REFCHECKER uses templates to detect missing edits in manual refactorings that might lead to behavior changes. Tables C.1 and C.1 present the template rules that REFCHECKER checks with brief descriptions. The rules are presented in a pseudo code manner. The following auxiliary functions are defined in order to simplify the rules presentation:

- `getClass(P, m)` returns the containing class of method m to be refactored.
- `getCallers(m)` returns all callers of m.
- `isAssociatedWithAField(m)` verifies whether the method m accesses any field declared in the same class.
- `checkBindingProblem (m1, m2)` verifies whether all method and variable references are identical between `m1` and `m2`.
- `verifyAccessibilityChange (m1, m2)` verifies whether method `m2` is visible to method `m1`.
- `haveDependences(m, stms)` verifies whether the remaining statements in method m after the extraction of `stmts` are dependent on any statements within extracted code `stms`.
- `getStatements(m, [beginLine;endLine])` returns the statements that are in between the range of lines specified from beginLine to endLine. In case of an empty range, it returns all statements from m.

Table C.1: **RefChecker**'s refactoring template rules (Part 1).

Move Method (P: original version, P_r: modified version, $m1_o$: method to be refactored, $m2_n$: newly added method)		

1	`$C_p$ ={}; $c_o$ = getClass($P$, $m1_o$);`
	`$C_p$ = $C_p$ ∪ {<'Remove Functionality', $c_o$>}`
		There must be a method deleting in the original version P.

2	`$c_o$ = getClass($P_r$, $m1_o$);`
	`$C_p$ = $C_p$ ∪ {<'Add Functionality', $c_o$>}`
		There must be a new method in the modified version P_r.

3	`$C$ = getCallers($P$, $m1_o$);`
	`FOREACH ($c$ in $C$) DO`
	`  IF (isAssociatedWithField($m1_o$)) THEN`
	`    $C_p$ = $C_p$ ∪ {<'Change Attribute Type', $c$>};`
	`  ELSE $C_p$ = $C_p$ ∪ {<'Update Statement', $c$>};`
		For all callers of $m1_o$, if the method call is associated to a field, there must be an attribute type change in P, and a statement update otherwise.

4	`$C$ = getCallers($P$, $m1_o$);`
	`FOREACH ($c$ in $C$) DO`
	`  $m$ = getMethod($P_r$, $c$);`
	`  IF (checkBindingProblem($c$, $m$)) THEN`
	`    $C_p$ = $C_p$ ∪ {<'Binding Problem', c>};`
		All callers of $m1_o$ in the modified version (m from P_r) must preserve all method and variable references from the original version P.

5	`IF (checkBindingProblem($m1_o$, $m2_n$)) THEN`
	`  $C_p$ = $C_p$ ∪ {<'Binding Problem', $m1_o$>};`
		All method and variables references in $m1_o$ must remain the same in the modified version P_r.

6	`$C$ = getCallers($P$, $m1_o$);`
	`FOREACH ($c$ in $C$) DO`
	`  $m$ = getMethod($P_r$, $c$);`
	`  IF (verifyAccessibilityChange ($c$, $m$)) THEN`
	`    $C_p$ = $C_p$ ∪ {<'Change Visibility', $m$>};`
		Added method $m2_n$ must be visible to the callers of the removed method $m1_o$.

Pull Up Method: rules 1, 2, 4, and 5		
Push Down Method: rules 1, 2, 4, and 5		

Table C.2: **RefChecker**'s refactoring template rules (Part 2).

Inline Method (P: original code, P_r: modified version, $m1_o$: method to be inlined)		
13	C_p ={}; c_o = getClass(P, $m1_o$); C_p = C_p ∪ {<'Remove Functionality', c_o>}	There must be a deleted method in the modified version P_r.
14	STM_o = getStatements ($m1_o$, []); C = getCallers(P, $m1_o$); **FOREACH** (c **in** C) **DO** $m2_n$ = getMethod(P_r, c); **IF** (isNotVoid (c)) **THEN** C_p = C_p ∪ {<'Update Statement', c>}; FOREACH (s in STM_o) DO C_p = C_p ∪ {<'Insert Statement', s, c>};	If the inlined method $m1_o$ has a return type, there must be an updated statement in each of its callers. Also, for all callers, there must exist a sequence of inserted statement inlined from $m1_o$.
-	Check rule 4	See 4

Rename Method (P: original code, P_r: modified version, $m1_o$: method to be renamed)		
15	C_p ={}; c_o = getClass(P, $m1_o$); C_p = C_p ∪ {<'Rename Method', c_o>}	There must be a method in P_r that had its signature changed when compared to the original version P.
16	C = getCallers(P, $m1_o$); **FOREACH** (c **in** C) **DO** C_p = C_p ∪ {<'Update Statement', c>};	For all callers to $m1_o$, there must be an updated statement in modified version P_r.