The Journal of Systems and Software 86 (2013) 1006–1022

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Comparing approaches to analyze refactoring activity on software repositories

Gustavo Soares[a,*], Rohit Gheyi[a], Emerson Murphy-Hill[b], Brittany Johnson[b]

[a] Department of Computing and Systems, Federal University of Campina Grande, Campina Grande, PB, 58429-900, Brazil
[b] Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA

ARTICLE INFO

Article history:
Received 21 January 2012
Received in revised form 28 August 2012
Accepted 18 October 2012
Available online 16 November 2012

Keywords:
Refactoring
Repository
Manual analysis
Automated analysis

ABSTRACT

Some approaches have been used to investigate evidence on how developers refactor their code, whether refactorings activities may decrease the number of bugs, or improve developers' productivity. However, there are some contradicting evidence in previous studies. For instance, some investigations found evidence that if the number of refactoring changes increases in the preceding time period the number of defects decreases, different from other studies. They have used different approaches to evaluate refactoring activities. Some of them identify committed behavior-preserving transformations in software repositories by using manual analysis, commit messages, or dynamic analysis. Others focus on identifying which refactorings are applied between two programs by using manual inspection or static analysis. In this work, we compare three different approaches based on manual analysis, commit message (Ratzinger's approach) and dynamic analysis (SafeRefactor's approach) to detect whether a pair of versions determines a refactoring, in terms of behavioral preservation. Additionally, we compare two approaches (manual analysis and Ref-Finder) to identify which refactorings are performed in each pair of versions. We perform both comparisons by evaluating their accuracy, precision, and recall in a randomly selected sample of 40 pairs of versions of JHotDraw, and 20 pairs of versions of Apache Common Collections. While the manual analysis presents the best results in both comparisons, it is not as scalable as the automated approaches. Ratzinger's approach is simple and fast, but presents a low recall; differently, SafeRefactor is able to detect most applied refactorings, although limitations in its test generation backend results for some kinds of subjects in low precision values. Ref-Finder presented a low precision and recall in our evaluation.

1. Introduction

Refactoring is the process of changing a software for evolving its design while preserving its behavior (Fowler, 1999). Empirical analysis of refactoring tasks in software projects is important, as conclusions and assumptions about evolution and refactoring still present insufficient supporting data. Research on these subjects certainly benefits from evidence on how developers refactor their code, whether refactorings activities may decrease the number of bugs, improve developers' productivity, or facilitate bug fixes. Besides identifying behavior-preserving transformations, it is also important to detect which refactorings happened between two program versions. For example, the inferred refactorings can help developers understand the modifications made by other developers and can be used to update client applications that are broken due to refactorings in library components (Henkel and Diwan, 2005). Understanding the dynamics of software evolution helps the conception of specific methods and tools.

Open source projects are appropriate and manageable sources of information about software development and evolution. A number of studies have performed such investigations in the context of refactoring (Soares et al., 2011; Murphy-Hill et al., 2009, 2012; Murphy-Hill and Black, 2008; Ratzinger et al., 2008; Ratzinger, 2007; Dig et al., 2006). Murphy-Hill et al. (2009, 2012) manually analyzed 40 pairs of versions from two open source repositories in order to identify refactoring activities. Similarly, Ratzinger et al. (2008) and Ratzinger (2007) proposed an approach to predict refactoring activities, performing static analysis on commit messages. Also Prete et al. (2010) proposed a tool called Ref-Finder that performs static analysis on both source and target versions, in order to discover the application of 63 refactorings presented by Fowler (1999). Finally, Soares et al. (2011) proposed an approach for analyzing refactoring activities in repositories with respect to frequency, granularity, and scope, over entire repositories' history. It is based on SafeRefactor (Soares et al., 2010), a tool that evaluates whether a transformation is behavior-preserving. SafeRef

* Corresponding author.
E-mail addresses: gsoares@dsc.ufcg.edu.br (G. Soares), rohit@dsc.ufcg.edu.br (R. Gheyi), emerson@csc.ncsu.edu (E. Murphy-Hill), bijohnso@csc.ncsu.edu (B. Johnson).

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2012.10.040

ACTOR analyzes a transformation and automatically generates tests for checking whether the behavior was preserved between two versions in the repository.

However, there are some contradicting evidence in previous studies. For instance, recent study found that refactoring may affect software quality negatively (Kim et al., 2012), different from other approaches (Fowler, 1999). As another example, Kim et al. (2011) found that the number of bug fixes increases after API-level refactorings. On the other hand, Ratzinger et al. (2008) found evidence that if the number of refactoring changes increases in the preceding time period, the number of defects decreases. They used different approaches to detect refactoring activities. It is important to evaluate the advantages and disadvantages of different approaches (Murphy-Hill et al., 2009; Kim et al., 2011; Weißgerber and Diehl, 2006; Ratzinger et al., 2008; Soares et al., 2011) to detect refactoring activities according to the subjects analyzed. If an approach misidentifies a version to be a refactoring version, this may have an impact on their conclusions.

In this article, we extend the previous work (Soares et al., 2011) by performing two experiments[1] to compare some approaches for identifying refactoring activities in software repositories. First, we compare Murphy-Hill et al. (2009, 2012) (manual analysis), Ratzinger et al. (2008) (commit message analysis), and SAFEREFACTOR approaches (Soares et al., 2011) with respect to effectiveness in detecting behavior-preserving transformations. Our second experiment compares the Murphy-Hill et al. (2009, 2012) (manual analysis) and REF-FINDER (Prete et al., 2010) (static analysis) approaches with respect to effectiveness in detecting which refactorings happened between two program versions. We perform both comparisons by evaluating 40 randomly sampled software versions of JHotDraw, a framework for development of graphical editors, and 20 randomly sampled software versions of the Apache Common Collections, an API build upon the JDK Collections Framework to provide new interfaces, implementations and utilities.

In our experiments, we found evidence that the manual analysis is the most reliable approach in detecting behavior-preserving transformations and refactorings applied. However it is time-consuming, and may incorrectly evaluate uncompilable programs. Moreover, it depends on evaluators' experience. SAFEREFACTOR's approach automatically detected a number of behavior-preserving transformations. However, it may not detect a number of non-behavior-preserving transformations due to the limitations of its test suite generator. It could not generate a test case exposing behavioral change for some programs that contain a graphical user interface or manipulate files. The commit message approach fails to predict a number of behavior-preserving transformations. Most of them were categorized as non-behavior-preserving transformations. It had low recall (0.16) and only average precision (0.57) in our experiments. It depends on guidelines that must be followed by developers during software development. In our experiment, REF-FINDER identified 24% of these refactorings. Moreover, 65% of the refactorings detected by REF-FINDER were incorrect. It is not simple to identify which refactorings were applied statically based on template matching. Some refactoring templates are similar, and the tool incorrectly identified some of them. So, we have some evidence that the approaches have different results. This may be the reason why some investigations found contradicting evidence.

This article is organized as follows: the following section describes approaches to analyze refactoring activities in software repositories. Section 3 compares manual, commit message and SAFEREFACTOR approaches to identify behavior-preserving transformations. In Section 4, we compare a manual approach and REF-FINDER to detect which refactorings happened between two program versions. Finally, we relate our work to others, and present concluding remarks. Appendix A formalizes some algorithms used to analyze the refactoring granularity and scope.

2. Approaches to identify refactoring activities

Next we give an overview of four approaches for identifying refactoring activities on software repositories: SAFEREFACTOR (Section 2.1), manual analysis (Section 2.2), commit message analysis (Section 2.3), and REF-FINDER (Section 2.4).

2.1. SAFEREFACTOR

In this section, we present an approach based on SAFEREFACTOR that identifies behavior-preserving transformations in open source software repositories. It also classifies a transformation with respect to granularity (a refactoring affects only the internals of a method, for instance, or spans over several methods and classes) and scope (a refactoring spans over a single package or affects multiple packages).

It uses, as input, repository source code and their configuration files – for instance, `build.xml`. Optionally, we can specify an interval of commits we would like to evaluate. Moreover, we can state the time limit used to generate tests. As result, it reports the total number of behavior-preserving transformations, and the granularity and the scope of them. This process consists of three major steps. The first step analyzes selected pairs of consecutive versions and classifies them as non-refactoring or refactoring by using SAFEREFACTOR. Then, we classify the identified refactorings with respect to granularity (Step 2) and scope (Step 3). The whole approach is illustrated in Fig. 1. Next, we show an overview of each step.

2.1.1. Step 1: Detecting behavioral changes

The first step uses SAFEREFACTOR to evaluate whether a transformation is behavior-preserving. SAFEREFACTOR (Soares et al., 2010) evaluates the correctness of each applied transformation. It checks for compilation errors in the resulting program, and reports those errors; if no errors are found, it analyzes the transformation and generates tests for checking behavioral changes.

The process is composed of five sequential steps for each refactoring application under test (Fig. 2). It receives as input two versions of the program, and outputs whether the transformation changes behavior. First, a static analysis automatically identifies methods in common (they have exactly the same modifier, return type, qualified name, parameters types and exceptions thrown in source and target programs) in both the source and target programs (Step 1). Step 2 aims at generating unit tests for methods identified in Step 1. It uses Randoop (Robinson et al., 2011; Pacheco et al., 2007) to automatically produce tests. Randoop randomly generates unit tests for classes within a time limit; a unit test typically consists of a sequence of method and constructor invocations that creates and mutates objects with random values, plus an assertion. In Step 3, SAFEREFACTOR runs the generated test suite on the source program. Next, it runs the same test suite on the target program (Step 4). If a test passes in one of the programs and fails in the other one, SAFEREFACTOR detects a behavioral change and reports to the user (Step 5). Otherwise, the programmer can have more confidence that the transformation does not introduce behavioral changes.

To illustrate SAFEREFACTOR's approach to detect behavioral changes, consider class `A` and its subclass `B` as illustrated in Listing 1. `A` declares the `k` method, and `B` declares methods `k`, `m`, and `test`. The latter yields 1. Suppose we want to apply the Pull Up Method refactoring to move `m` from `B` to `A`. This method contains a reference to `A.k` using the `super` access. The use of either Eclipse JDT 3.7 or

[1] All experimental data are available at: http://www.dsc.ufcg.edu.br/ spg/jss_experiments.html.

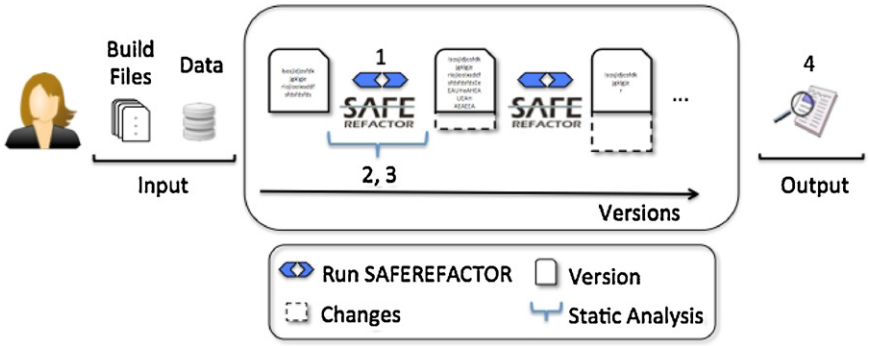

Fig. 1. An approach for detecting behavior-preserving transformations on software repositories. 1. The tool runs SafeRefactor over all consecutive pairs of versions and identifies the refactorings; 2. It analyzes refactorings with respect to granularity; 3. It analyzes refactorings with respect to scope; 4. The tool reports the analysis results.

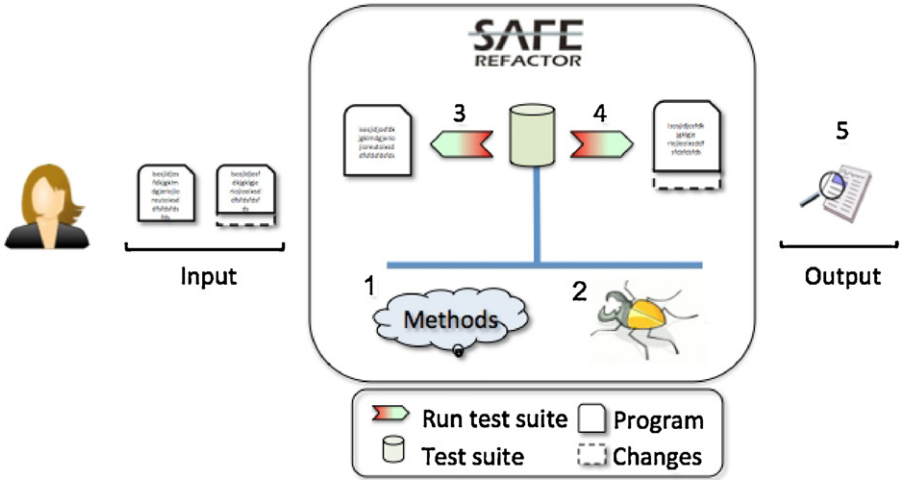

Fig. 2. SafeRefactor major steps: 1. It identifies common methods in the source and target programs; 2. The tool generates unit tests using Randoop; 3. It runs the test suite on the source program; 4. The tool runs the test suite on the target program; 5. The tool shows whether the transformation changes behavior.

JastAdd Refactoring Tools (JRRT) (Schäfer and de Moor, 2010), proposed to improve the correctness of refactorings by using formal techniques, to perform this refactoring will produce the program presented in Listing 2.[2] Method `m` is moved from `B` to `A`, and `super` is updated to `this`; a compilation error is avoided with this change. Nevertheless, a behavioral change was introduced: `test` yields 2 instead of 1. Since `m` is invoked on an instance of `B`, the call to `k` using `this` is dispatched on to the implementation of `k` in `B`.

Listing 1. Before Refactoring

```
public class A {
  int k() {
    return 1;
  }
}
public class B extends A {
  int k() {
    return 2;
  }
  int m() {
    return super.k();
  }
  public int test() {
    return m();
  }
}
```

[2] The same problem happens when we omit the keyword **this**.

Listing 2. After Refactoring. Applying Pull Up Method in Eclipse JDT 3.7 or JRRT leads to a behavioral change due to incorrect change of **super** to **this**.

```
public class A {
  int k() {
    return 1;
  }
  int m() {
    return this.k();
  }
}
public class B extends A {
  int k() {
    return 2;
  }
  public int test() {
    return m();
  }
}
```

Assuming the programs in Listings 1 and 2 as input, SafeRefactor first identifies the methods with matching signatures on both versions: `A.k`, `B.k`, and `B.test`. Next, it generates 78 unit tests for these methods within a time limit of two seconds. Finally, it runs the test suite on both versions and evaluates the results. A number of tests (64) passed in the source program, but did not pass in the refactored program; so SafeRefactor reports a behavioral change. Next, we show one of the generated tests that reveal behavioral changes. The test passes in the source program since the value

returned by `B.test` is 1; however, it fails in the target program since the value returned by `B.test` is 2.

```
public void test() {
  B b = new B();
  int x = b.test();
  assertTrue(x == 1);
}
```

In his seminal work on refactoring, Opdyke (1992) compares the observable behavior of two programs with respect to the `main` method (a method in common). If it is called upon both source and target programs (programs before and after refactoring, respectively), with the same set of inputs, the resulting set of output values must be the same. SafeRefactor checks the observable behavior with respect to randomly generated sequences of methods and constructor invocations; these invocations apply only to methods in common. If the source and target programs have different results for the same input, they do not have the same behavior. The approach follows an open world assumption (OWA), in which every public method can be a potential target for the test suite generated by SafeRefactor.

Considering a Rename Method refactoring from `A.m(..)` to `A.n(..)`, the set of methods identified in Step 1 includes none of them. This situation also occurs by renaming a class. We cannot compare the renamed method's behavior directly. However, SafeRefactor compares them indirectly if another method in common (`x`) calls them. Step 2 thus generates tests that call `x` in the generated tests. It is similar to Opdyke's notion. If `main` calls them, then SafeRefactor compares those methods indirectly. Moreover, a simple rename method may enable or disable overloading (Schäfer et al., 2008), which is a potential source of problems.

This approach uses a strict notion of refactoring; a transformation is considered a refactoring if SafeRefactor does not detect a behavioral change between source and target programs. Otherwise, it is considered a non-refactoring, despite the developer's intention of applying a refactoring. As a consequence, our study only considers correctly applied refactorings. Although SafeRefactor does not exhaustively guarantee that transformations are indeed refactorings, confidence on correctness is increased (Soares et al., 2010, 2012).

Before running the tests, our approach compiles the versions based on the `build.xml` files received as parameters. Still, some versions may present compilation problems. Uncompilable versions are considered non-refactorings since after a refactoring the code should compile. We also collect test coverage data to improve confidence on test results.

2.1.2. Step 2: Analyzing refactoring granularity

Step 2 analyzes the refactoring granularity. We use two approaches to classify the refactorings regarding this aspect: High/Low level and the size of the transformation. *High level* refactorings are transformations that affect classes and method signatures, including class attributes. For instance, Extract Method, Rename Method, Add Parameter are High level refactorings (Fowler, 1999). On the other hand, *Low level* refactorings, such as Rename Local Variable, and Extract Local Variable (Fowler, 1999), change blocks of code only within methods.

In order to measure granularity, we statically analyze the identified refactorings with respect to classes, fields, and methods signatures. If both versions contain a different set of method signatures, we classify the refactoring as High level, otherwise as Low level (see Algorithm 2 in Appendix A).

2.1.3. Step 3: Analyzing refactoring scope

Finally, Step 3 collects the refactoring scope. For this purpose we classify refactoring scope as local or global. *Global* refactorings affect classes in more than one package. For example, if there is a client using class `X` in a different package, renaming class `X` is a Global refactoring. *Local* refactorings, on the other hand, affect a single package, such as renaming a local variable or renaming a class that is not used outside that package. Notice that some refactorings can be classified as local or global within different situations. We perform a static analysis to identify whether the changes affect more than one package (see Algorithm 3 in Appendix A).

2.2. Manual analyses overview

The manual analysis is based on the methodology of Murphy-Hill et al. (2009, 2012), which compares the code before each commit against its counterpart after the commit. For brevity, we will simply call this approach 'Murphy-Hill'. For each commit, two evaluators sit together and use the standard Eclipse diff tool to compare files before the commit to the files after the commit. Reading through each file, the evaluators attempt to logically group fine-grained code changes together, classifying each change as either a refactoring (such as "Extract Method") or a non-refactoring (such as "Add null Check"). The evaluators also attempt to group together logical changes across files by re-comparing files as necessary. For example, if the evaluators noticed that a change to one file deleted a piece of code, they would have initially classified that change as a non-refactoring, but if later the evaluators found that the code had actually been moved to another file, the evaluators would re-classify the two changes together as a single refactoring. If the two evaluators did not agree on whether a change was a refactoring, to reach agreement they would discuss under what circumstances it might possibly change the behavior of the program.

This approach is flexible, because we are able to classify changes as refactorings, even if they had not been previously identified as refactorings in prior work (Murphy-Hill et al., 2009, 2012). By assessing the transformations performed during a commit, we are able to determine whether a commit contained only refactorings, no refactorings, or a mix of refactorings and non-refactorings.[3]

2.3. Commit message analyses overview

Ratzinger (2007) and Ratzinger et al. (2008) proposed an approach to detect whether a transformation is a refactoring by analyzing a commit message. If the message contains a number of words that are related to refactoring activities, the transformation is considered a refactoring. We implemented their approach in Algorithm 1.

Algorithm 1. Ratzinger

Input: *message* ⇐ commit message
Output: Indicates whether a transformation is a refactoring
  keywords ⇐ {refactor, restruct, clean, not used, unused, reformat, import, remove, removed, replace, split, reorg, rename, move}
  if '*needs refactoring*' ∈ *message* **then**
    FALSE
  end if
  foreach *k* ∈ *keywords* **do**
    if *k* ∈ *message* **then**
      TRUE
    end if
  end foreachFALSE

[3] One difference between the present study and the previous study (Murphy-Hill et al., 2012) was that in the previous study we included a "pure whitespace" category; in the present study, we consider "pure whitespace", "Java comments changes", and "non-Java files changes" to be a refactoring, to maintain consistency with the definition of refactoring used by SafeRefactor.

The implemented analyzer is based on Ratzinger et al.'s algorithm (Ratzinger, 2007; Ratzinger et al., 2008), which we will simply call 'Ratzinger'.

2.4. Ref-Finder overview

The Ref-Finder tool (Prete et al., 2010) performs static analysis on both source and target versions, in order to discover the application of complex refactorings. The tool identifies 63 refactorings presented by Fowler (1999). Each refactoring is represented by a set of logic predicates (a template), and the matching between program and template is accomplished by a logic programming engine. Ref-Finder was evaluated in transformations applied to three real case studies (Carol, Columba and jEdit). The goal of this tool is to decompose a transformation into a number of refactorings. It does not evaluate whether a transformation is behavior-preserving. When it finds a refactoring, it yields the line of the class and the type of the refactoring performed.

3. Evaluating techniques for identifying behavior-preserving transformations

In this section, we present our experiment (Basili et al., 1986) to evaluate approaches for identifying behavior-preserving transformations. First, we present the experiment definition (Section 3.1), and show the experiment planning (Section 3.2). Next, we describe the experiment operation, and show the results (Section 3.3). Then, we interpret and discuss them in Section 3.4. Finally, we describe some threats to validity (Section 3.5).

3.1. Definition

The goal of this experiment is to analyze three approaches (SafeRefactor, Ratzinger, and Murphy-Hill) for the purpose of evaluation with respect to identifying behavior-preserving transformations from the point of view of researchers in the context of open-source Java project repositories. In particular, our experiment addresses the following research questions:

- **Q1**. Do the approaches identify all behavior-preserving transformations?

  For each approach, we measure the true positive rate (also called *recall*). *tPos* (true positive) and *fPos* (false positive) represent the correctly and incorrectly behavior-preserving transformations, respectively. *tNeg* (true negative) and *fNeg* (false negative) represent correctly and incorrectly identified non-behavior-preserving transformations, respectively. Recall is defined as follows (Olson and Delen, 2008):

$$recall = \frac{\#tPos}{\#tPos + \#fNeg} \quad (1)$$

- **Q2**. Do the approaches correctly identify behavior-preserving transformations?

  For each approach, we measure the false positive rate (*precision*). It is defined as follows (Olson and Delen, 2008):

$$precision = \frac{\#tPos}{\#tPos + \#fPos} \quad (2)$$

- **Q3**. Are the overall results of the approaches correct?

  We measure the *accuracy* of each approach by dividing the total correctly identified behavior-preserving and non-behavior-preserving transformations by the total number of samples. It is defined as follows (Olson and Delen, 2008):

$$accuracy = \frac{\#tPos + \#tNeg}{\#tPos + \#fPos + \#tNeg + \#fNeg} \quad (3)$$

3.2. Planning

In this section, we describe the subjects used in the experiment, the experiment design, and its instrumentation.

3.2.1. Selection of subjects

We analyze two Java open-source projects. JHotDraw is a framework for development of graphical editors. Its SVN repository contains 650 versions. The second SVN repository is from the Apache Common Collections (we will simply call 'Collections'), which is an API build upon the JDK Collections Framework to provide new interfaces, implementations and utilities.

We randomly select 40 out of 650 versions from the JHotDraw repository (four developers were responsible for these changes) and 20 out of 466 versions from the Collections repository (six developers were responsible for these changes). For each randomly selected version, we take its previous version to analyze whether they have the same behavior. For instance, we evaluate Version 134 of JHotDraw and the previous one (133).

Tables 1 and 2 indicate the version analyzed, number of lines of code of the selected version and its previous version, and characterize the scope and granularity of the transformation. We evaluate transformations with different granularities (low and high level) and scope (local and global).

3.2.2. Experiment design

In our experiment, we evaluate one factor (approaches for detecting behavior-preserving transformations) with three treatments (SafeRefactor, Murphy-Hill, Ratzinger). We choose a paired comparison design for the experiment, that is, the subjects are applied to all treatments. Therefore, we perform the approaches under evaluation in the 60 pairs of versions. The results can be "Yes" (behavior-preserving transformation) and "No" (non-behavior-preserving transformation).

3.2.3. Instrumentation

The last two authors of this article conducted the Murphy-Hill approach. We automate the experiment for checking SafeRefactor and Ratzinger results.[4] The Ratzinger approach was implemented in Algorithm 1.

We use SafeRefactor 1.1.4 with default configuration but using a time limit of 120 s, and setting Randoop to avoid generating non-deterministic test cases. Additionally, SafeRefactor may have different results each time it is executed due to the randomly generation of the test suite. So, we execute it up to three times in each version. If none of the executions finds a behavioral change, we classify the version as behavior-preserving transformation. Otherwise, we classify it as non-behavior-preserving transformation. We use Emma 2.0.5312[5] to collect the statement coverage of the test suite generated by SafeRefactor in the resulting program. Additionally, we collect additional metrics for the subjects: non-blank, non-comment lines of code, scope, and granularity. The algorithms to collect refactoring scope and granularity are presented in Appendix A.

Since we previously do not know which versions contain behavior-preserving transformations, the first author of this article compared the results of all approaches in all transformations to derive a *Baseline*. For instance, if the Murphy-Hill approach yielded "Yes" and SafeRefactor returned "No", the first author would check whether the test case showing the behavioral change reported by

[4] The automated experiment containing SafeRefactor and Ratzinger approaches, and additional information are available at: http://www.dsc.ufcg.edu.br/ spg/jss_experiments.html.

[5] http://emma.sourceforge.net/.

Table 1
Results of analyzing 40 versions of JHotDraw. LOC, non-blank, non-comment lines of code before and after the changes; Granu., granularity of the transformation; Scope, scope of the transformation; Refact., Is it a refactoring?; #Tests, number of tests used to evaluate the transformation; Cov. (%), statement coverage on the target program; MH, Murphy-Hill.

Version	LOC		Granu.	Scope	Baseline	Ratzinger	MH	SafeRefactor		
	Before	After			Refact.	Refact.	Refact.	Refact.	# Tests	Cov. (%)
134	20,422	20,422	Low	Local	No	No	No	Yes	449	24
151	28,103	28,108	Low	Local	No	No	No	No	4778	48
156	28,121	28,121	Low	Local	Yes	Yes	Yes	Yes	4778	48
173	28,101	28,052	Low	Global	No	Yes	No	Yes	454	20
174	28,052	28,053	Low	Global	Yes	No	Yes	Yes	599	23
176	28,055	28,055	Low	Local	No	No	No	No	436	33
179	28,065	28,065	Low	Local	Yes	No	Yes	Yes	3199	41
193	28,291	28,298	Low	Global	Yes	No	Yes	Yes	2108	39
251	28,398	28,398	Low	Local	Yes	No	Yes	Yes	5162	49
267	28,398	28,409	Low	Local	No	No	No	Yes	5433	48
274	32,408	32,408	Low	Local	Yes	No	Yes	Yes	418	4
275	32,408	32,408	Low	Local	Yes	No	Yes	Yes	476	3
294	39,249	39,081	High	Local	No	No	No	No	Compilation error	
300	39,161	39,161	Low	Local	Yes	No	Yes	Yes	314	14
302	38,993	39,161	High	Local	No	No	No	No	Compilation error	
304	39,161	39,161	Low	Local	Yes	No	Yes	Yes	286	15
318	39,160	39,173	Low	Local	No	No	No	Yes	2356	8
322	39,377	39,480	High	Local	No	No	No	Yes	802	26
324	39,472	39,551	High	Global	No	No	No	No	490	10
344	51,339	51,596	High	Global	No	No	No	Yes	1022	13
357	52,991	52,636	Low	Global	No	No	Yes	No	Compilation error	
384	52,594	52,601	Low	Local	No	No	No	Yes	2167	24
405	53,708	53,708	Low	Local	Yes	No	Yes	Yes	1816	10
409	53,712	53,721	High	Global	No	No	No	Yes	1687	10
458	64,939	64,940	Low	Local	No	No	No	No	1549	12
501	69,300	69,404	High	Global	Yes	No	Yes	Yes	2600	29
503	69,570	69,566	High	Global	Yes	No	Yes	No	2084	21
518	71,578	71,979	High	Global	No	No	No	No	1114	9
526	72,027	72,053	High	Global	No	No	No	No	1942	7
549	72,245	72,286	Low	Global	No	No	No	No	1940	12
590	74,235	71,943	High	Local	Yes	No	Yes	Yes	255	7
596	72,402	72,553	High	Global	No	No	No	No	823	25
609	72,752	72,754	High	Global	No	No	No	Yes	2417	31
649	75,664	75,664	Low	Local	No	No	No	Yes	1752	27
650	75,664	76,220	High	Global	No	No	No	Yes	1755	27
660	76,469	79,135	High	Global	No	No	No	No	966	27
697	79,708	79,708	Low	Local	Yes	No	Yes	Yes	1418	21
700	79,731	79,741	Low	Global	No	No	No	No	1282	23
704	79,746	79,746	Low	Local	No	No	No	Yes	2334	28
743	80,208	80,213	Low	Local	No	No	No	Yes	1175	23
					Precision	0.5	0.93	0.5		
					Recall	0.07	1	0.93		
					Accuracy	0.65	0.98	0.65		

SafeRefactor was correct. If so, the correct result was "No". So, we establish a *Baseline* to check the results of each approach, and calculate their recall, precision, and accuracy.

3.3. Operation

Before performing the experiment, we implemented a script to download 60 pairs of versions and log commit information: version_id, date, author, and commit message. We named each pair of versions with suffix _BEFORE and _AFTER to indicate the program before and after the change. The versions that were non-Eclipse projects were made Eclipse projects so that the Murphy-Hill approach could use the Eclipse diff tool. The third and fourth authors of this article scheduled two meetings to analyze the subjects following the Murphy-Hill approach. The automated analyses of SafeRefactor and Ratzinger were performed on a MacBook Pro Core i5 2.4 GHz and 4GB RAM, running Mac OS 10.7.4.

Additionally, for SafeRefactor we also downloaded all dependencies of JHotDraw. SafeRefactor compiles each version and than generates tests to detect behavioral changes. We also manually create buildFiles to compile the JHotDraw subjects. As software evolves, it may modify the original build file due to changes in the project structure, compiler version or used libraries. For JHotDraw's subjects, we needed 4 buildFiles, and used JDK 1.5 and 1.6. We do not have information which JDK they used. For each subject, we used SafeRefactor with a specific buildFile. The Apache Common Collections subjects were compiled with JDK 1.6. Moreover, we performed the test generation of Randoop, and the test execution using JDK 1.6 on both samples.

Tables 1 and 2 present the results of our evaluation for JHotDraw and Collections, respectively. Column *Version* indicates the version analyzed, and Column *Baseline* shows whether the pair is indeed a refactoring. This column was derived based on all results, as explained in Section 3.2.3. The following columns represent the results of each approach. In the bottom of the table, it is shown the precision, recall, and accuracy of each approach with respect to Column *Baseline*.

We have identified 14 and 11 refactorings (Baseline) in JHotDraw and Collections, respectively. In 17 out of 60 pairs, all approaches have the same result. While some versions fixed bugs, such as Versions 134, 176, and 518, or introduced new features, for instance Version 572952, others are refactorings (see Baseline of Tables 1 and 2). Some versions did not change any Java file (Versions 251, 274, 275, 300, 304, 405, 697, 609497, 923339,

Table 2
Results of analyzing 20 versions of Apache Common Collections. LOC, non-blank, non-comment lines of code before and after the changes; Granu., granularity of the transformation; Scope, scope of the transformation; Refact., Is it a refactoring?; #Tests, number of tests used to evaluate the transformation; Cov. (%), statement coverage on the target program; MH, Murphy-Hill.

Version	LOC		Granu.	Scope	Baseline	Ratzinger	MH	SafeRefactor		
	Before	After			Refact.	Refact.	Refact.	Refact.	# Tests	Cov. (%)
572952	26,350	26,428	High	Global	No	No	No	Yes	879	29
609497	26,428	26,428	Low	Local	Yes	No	Yes	Yes	2259	42
637489	26,428	26,454	High	Local	No	No	No	No	3158	44
656960	26,501	26,514	Low	Local	No	No	No	Yes	3487	47
711140	26,536	26,539	Low	Local	No	No	No	Yes	1247	36
814123	26,558	26,558	Low	Global	Yes	No	Yes	Yes	2972	44
814128	26,558	26,558	Low	Global	Yes	No	Yes	Yes	2741	44
814997	26,558	26,761	High	Global	No	No	Yes	No	Compilation error	
815022	20,221	20,222	Low	Local	No	Yes	Yes	No	Compilation error	
815042	20,258	20,255	Low	Local	No	Yes	Yes	No	Compilation error	
923339	20,901	20,901	Low	Local	Yes	No	Yes	Yes	2712	49
956279	20,901	20,848	High	Local	No	No	No	No	2709	49
966327	20,926	21,513	Low	Global	Yes	No	Yes	Yes	2667	49
1023771	21,551	21,551	Low	Global	Yes	No	Yes	Yes	2201	44
1023897	21,551	21,551	Low	Global	Yes	No	Yes	Yes	2033	44
1095934	21,608	21,608	Low	Local	Yes	No	Yes	Yes	3180	51
1148801	21,618	21,628	High	Global	Yes	Yes	Yes	Yes	3237	50
1299210	21,627	21,627	Low	Global	Yes	Yes	Yes	Yes	1886	49
1300075	21,632	21,632	Low	Local	Yes	Yes	Yes	Yes	1813	48
1311904	21,636	21,893	High	Global	No	No	No	Yes	2072	48
					Precision	0.6	0.79	0.73		
					Recall	0.27	1	1		
					Accuracy	0.5	0.85	0.8		

1095934) or changed just Java comments (Versions 156, 814123, 814128, 966327, 1023771, 1023897, 1299210, 1300075). In this study, we regard them as refactorings (behavior-preserving transformations).

The Murphy-Hill approach detected all refactorings of JHotDraw and Collections, which means a recall of 1 on both samples. However, it classifies four uncompilable versions as refactoring: one in JHotDraw (Version 357) and three in Collections (Versions 814997, 815022, 815042). This is the main reason why the manual inspection performed by the Murphy-Hill approach is not considered as the Baseline alone. So, 14 out of the 15 detected refactorings were correct in JHotDraw (precision of 0.93) and 11 out of the 14 detected refactorings in Collections were correct (precision of 0.79). The Murphy-Hill analysis correctly classified 39 out of 40 versions in JHotDraw and 17 out of 20 versions in Collections, leading to an accuracy of 0.98 and 0.85, respectively.

SafeRefactor identified all refactorings but one (Version 503), leading to a recall of 0.93 in JHotDraw sample. However, it also classified 13 non-refactoring as refactoring, which gives it a precision of 0.5. SafeRefactor correctly classified 26 out of the 40 pairs of JHotDraw (Accuracy of 0.65). On the other hand, it had an accuracy of 0.8 in Collections, which means that it was correct in 16 out of the 20 versions. SafeRefactor identified 11 out of the 11 refactorings (recall of 1). However, it incorrectly classified 4 versions as refactoring (precision of 0.73).

Finally, the Ratzinger approach correctly classified 26 out of the 40 versions of JHotDraw (accuracy of 0.65) and 10 out of 20 versions of Collections (accuracy of 0.5). The approach detected 1 (Version 156) out of 14 refactorings in the JHotDraw sample, and 3 out of 11 refactorings in Collections, having recall values of 0.07 and 0.27, respectively. The approach also incorrectly classified three versions as refactoring: Version 173 of JHotDraw (precision of 0.5) and Versions 815022 and 815042 of Collections (precision of 0.6). Table 3 summarizes the approaches' results with respect to false positives, false negatives, true positives, and true negatives. It also shows the overall recall, precision, and accuracy of each approach.

Performing the evaluated approaches involves different time costs. The Murphy-Hill approach took around 15 min to evaluate each subject. However, in some subjects containing larger changes, the approach took up to 30 min and was not able to check all changed files. Ratzinger automatically evaluate the commit message in less than a second. SafeRefactor took around 4 min to analyze each subject.

3.4. Discussion

In this section, we interpret and discuss the results. First, we present the main advantages and disadvantages of each approach. Then, we summarize the answers of the research questions (Section 3.4.4).

3.4.1. Murphy-Hill

The manual analysis presented the best results in terms of accuracy, recall, and precision, in our evaluation. An evaluator can carefully review the code to understand the syntax and the semantic changes to check whether they preserve behavior. Although a manual process can be error-prone, the Murphy-Hill et al. approach (Murphy-Hill et al., 2009, 2012) double checked the results by using two experienced evaluators. Moreover, they systematically decompose the transformation in minor changes making it easier to understand them. They also used a diff tool to help them analyze the transformation.

On the other hand, it is time consuming to analyze all changes in large transformations. For instance, Collections Versions 1148801, 814997, 815042, and 966327 were so large that the reviewers could not inspect all the changes. Furthermore, it is not trivial to identify

Table 3
Summary of false positives, false negatives, true positives, and true negatives.

	Ratzinger	Murphy-Hill	SafeRefactor
False positive	3	4	17
False negative	21	0	1
True positive	4	25	24
True negative	32	31	18
Total	60	60	60
Recall	0.16	1	0.96
Precision	0.57	0.86	0.59
Accuracy	0.6	0.93	0.7

Table 4
False positives of SafeRefactor. Problem, description of the reason of the false positive; Versions, ids of the versions related to the false positives.

#	Problem	Versions
1	Testing of GUI code	318, 322, 344, 384, 409, 609, 650, 704, 743
2	Tests do not cover impacted methods	173, 267, 322, 344, 649, 650
3	Tests do not cover impacted branches	134, 322, 711140
4	Weak JUnit assertions	650
5	Cannot apply regression testing	572952, 656960, 1311904

whether the code compiles by manually inspecting the transformation. The approach classified four versions that do not compile as refactoring.

In Version 357 of JHotDraw, among with other changes, the `AbstractDocumentOrientedApplication` class was moved from folder `org/jhotdraw/app` to folder `org/jhotdraw/application`. Although this seems to be a move package refactoring, it fix a compilation error because the class begins with the statement `package org.jhotdraw.application;` in both versions. Also, the commit message describes the transformation as fixing broken repository, which suggest that the transformation is not a refactoring. SafeRefactor detected compilation errors in this version.

Finally, the manual analysis classified 15 versions as having a mix of refactorings and non-refactorings. The SafeRefactor and Ratzinger approaches are not able to identify which refactorings are applied. Section 4 will evaluate two approaches (manual analysis and Ref-Finder) for identifying which refactorings happened between two program versions.

3.4.2. SafeRefactor

Although the manual analysis had the best results, it is a time-consuming activity to manually analyze all versions. It also depends on experienced evaluators. SafeRefactor has the advantage of automating this process, making an entire repository analysis feasible. The main problem of SafeRefactor was the high number of false positives in the JHotDraw sample, that is, non-refactorings that were classified as refactoring, which leaded to the precision of only 0.5. In the Collections sample, its precision was close to manual analysis (0.73–0.79), though. Next, we discuss about the false positives, false negatives, and also the true negatives of SafeRefactor.

3.4.2.1. False positives. SafeRefactor had 13 and 4 false positives in the JHotDraw and Collections samples, respectively. We manually analyzed each one and classified them as shown in Table 4. Most of the false positives were related to testing of GUI code. Application code may interact with the user (such as creating a dialog box) in a variety of different situations. In JHotDraw, some generated tests needed manual intervention to cover the functionality under test. SafeRefactor ignored them during evaluation. Moreover, Randoop did not generate tests for methods that require events from the Java AWT framework, for instance `MouseEvent`, since Randoop could not generate this type of dependence.

Recently, a new feature was added to Randoop to allow specifying a mapping from current method calls to a replacement call (Robinson et al., 2011). For instance, the `javax.swing.JOptionPane.showMessageDialog` method, which usually presents a dialog box, can be replaced with a call that simply prints out the message and returns. In this way, it can be used to remove dialog boxes that require a response. We plan to incorporate this feature into SafeRefactor's approach in the near future.

SafeRefactor also generated false positives because the tests generated by Randoop within a time limit did not cover methods changed by the transformation. For instance, while in Versions 173, 267, 649, one changed method was not covered by the tests, in Versions 322 and 650, two and three changed methods were not covered, respectively. SafeRefactor passes to Randoop the list of all methods in common for both versions of a pair. The time limit passed to Randoop to generate the tests may have been insufficient to produce a test for these methods. The average statement coverage of the tests was 22.68% and 45.12% in JHotDraw and Collections, respectively. As future work, we intend to improve SafeRefactor by identifying the methods impacted by a transformation. In this way, we can focus on generating tests for those methods.

Moreover, Randoop uses primitive, String and return values as input to the called methods. Still, some methods may present additional dependencies. For instance, parameters from class libraries may not be tested by Randoop if the library is not also under test.

Additionally, in Versions 134, 322, and 711140, Randoop produced tests that call the changed methods, but the tests did not cover the branches affected by the change. In those cases, the arguments produced by Randoop to the methods under test were not sufficient to exercise every behavior possible. The Randoop team recently incorporated the option of using any constant that appears in the source code as input to the methods under test (Robinson et al., 2011). Moreover, it allows users to specify primitives or String values as input to specific methods. We plan to investigate whether applying them may reduce SafeRefactor's false positives.

On the other hand, in Version 650 there were two changes that were covered by the tests, but the assertion established in the tests were not sufficient to detect the change. For instance, the `ComplexColorWheelImageProducer.getColorAt` method returns an array of floating-point values. Version 650 fixes the value returned by this method, but the test generated by Randoop only checks whether the value returned was not null. If Randoop could generate asserts to check the values of the array, the behavioral change would be detected. The other change affects one private attribute. Recently, Robinson et al. (2011) introduced an enhancement to Randoop that allows the user to define a set of observer methods to the attributes, and check their results – an observer method is a method with no side effects. Therefore, instead of having a single assertion at the end of a generated test, there may be many assertions at the end, one for each applicable observer method. As future work, we will investigate how to automatically compute the observer methods and pass to Randoop to check whether this option improves its effectiveness.

Finally, 3 out of the 4 false positives of Collections were due to addition or removal of methods not used in other parts of the program. If the transformation removes a method, it invalidates every unit test that directly calls the absent method. Likewise, if a method and its unit test is added, this unit test would not compile in the original version. Because of that, SafeRefactor identifies the common methods of the program, and tests them in the two versions of the pair, comparing their results. The tests indirectly exercise the change cause by an added/removed method, as long as this method affects the common methods. Opdyke compares the observable behavior of two programs with respect to the `main` method (a method in common). If it is called twice (source and target programs) with the same set of inputs, the resulting set of output values must be the same (Opdyke, 1992). SafeRefactor checks the observable behavior with respect to randomly generated sequences of methods and constructor invocations. They only contain calls to methods in common. Therefore, SafeRefactor can produce false positives in the API context when features are removed or added, since their code may not be used in other parts of the program but only by clients of the API.

3.4.2.2. False negatives. In Version 503 of JHotDraw, SafeRefactor showed a false negative. By manually inspecting the results we identified that the behavioral change was due to a non-deterministic behavior of JHotDraw. Randoop generated an assertion that indirectly checks the value returned by the `toString` method of an object of class `DrawingPageable`. This class does not implement `toString`. Therefore, it was returned the default value of `toString`, which prints a unique identifier based on the hashcode. The hashcode may change each time the program is executed, which was the cause of the non-deterministic result.

Nondeterministic results tend to fall into simple patterns, such as the default return value of `toString`. To avoid that, Randoop has the option of executing the tests twice and removing the tests that return different results (Robinson et al., 2011). We also implemented this option in SafeRefactor, which was used in the experiment. However, it was not sufficient to eliminate all cases of non-deterministic results, such as the one in Version 503.

3.4.2.3. True negatives. In this section, we discuss some of the non-behavioral transformations detected by SafeRefactor. In Version 637489 of the Collections API, an overridden method was changed, while Version 956279 changes a `toString` method. Any overridden method may have a very different behavior from the original, which favors its detection by SafeRefactor.

In JHotDraw, Version 151 changes the field value inside a constructor, which is detected by an assertion generated by Randoop. In some transformations, the target program raised an exception. In Versions 176, 518 and 526, SafeRefactor identified a `NullPointerException` in the target program inside a method body and constructors. In Version 324, the transformation removed an interface from a class. The resulting code yields a `ClassCastException` identified by SafeRefactor. Version 596 removed a `System.exit` from a method body.

On the other hand, the behavioral changes found by SafeRefactor in Versions 458, 549, 660, 700 were due to non-deterministic results of JHotDraw. JHotDraw contains global variables that lead to different results of the tests depending of the other that they are executed. SafeRefactor currently executes the tests generated by Randoop in batch through an Ant script. As future work, we plan to implement in SafeRefactor an option to execute the tests in the same order in the source and target versions to avoid non-deterministic results because of the order of the tests.

In our experiments, SafeRefactor had better results evaluating a repository of a data structure library (Collections) than one of a GUI application (JHotDraw). The first one was easier to evaluate since it does not have GUI, does not produce non-deterministic results, and require simpler arguments to exercise its behavior. On the order hand, APIs are less likely to have behavioral changes during its evolution (Robinson et al., 2011).

3.4.3. Ratzinger

The Ratzinger approach has the advantage of being the simplest and fastest approach for identifying behavior-preserving transformations. However, in our experiment, many of the commit messages do not contain keywords related to refactoring, which led this approach to a recall of only 0.27 in the Collections sample and 0.07 in the JHotDraw sample. Only 4 out of 25 refactoring revisions in both repositories contain some of the refactoring keywords established by the approach.

Additionally, 3 out of 7 refactorings identified by the approach were false positives. In Version 173 of JHotDraw, the commit message indicates that developers removed unused imports and local variables, which suggests the commit was a refactoring. However, by manually inspecting the changes, we checked that one of the removed local variable assignments contains a method call that changes UI components. SafeRefactor also classified this transformation as refactoring since the tests generated by Randoop did not detect this behavioral change in the GUI. This approach also classified Versions 815022 and 815042 as refactoring, but SafeRefactor detected that these versions do not compile, so they cannot be classified as refactorings.

It is not simple to predict refactorings by just inspecting the commit message. The results confirm Murphy-Hill et al. findings (Murphy-Hill et al., 2009, 2012), which suggest that simply looking at commit messages is not a reliable way of identifying refactorings. Nevertheless, in some situations, if the company recommend strict patterns when writing a commit message, this approach may be useful.

3.4.4. Answers to the research questions

From the evaluation results, we make the following observations:

- **Q1**. Do the approaches identify all behavior-preserving transformations?

  No. We found evidence that Murphy-Hill approach is capable of detecting all behavior-preserving transformations since it achieved a recall of 1.0. With respect to the automated approaches, SafeRefactor had an excellent recall of 0.96, but it may miss non-behavior-preserving transformations due to limitations of Randoop. On the other hand, our results show evidence that Ratzinger approach may miss a number of behavior-preserving transformations since it had an overall recall of only 0.16. Many of the evaluated behavior-preserving transformations were not documented in the commit messages in the way it is expected by this approach (see Section 3.4.3).
- **Q2**. Do the approaches correctly identify behavior-preserving transformations?

  No. Our results show evidence that the Murphy-Hill approach is the most precise among the evaluated approaches (precision of 0.86). However it may incorrectly classify transformations that contain compilation errors as behavior-preserving transformations. It is difficult to manually reason whether a program compiles. With respect to the automated approaches, the results indicate that SafeRefactor (0.59) is slightly more precise than Ratzinger (0.57). Some of the non-behavior-preserving transformations evaluated contain commit messages related to refactorings that were applied among other changes, leading the Ratzinger approach to incorrectly classify them as behavior-preserving transformations.
- **Q3**. Are the overall results of the approaches correct?

  The results indicate the Murphy-Hill approach is very accurate. In our experiment, it only failed in 4 out of the 60 subjects (accuracy of 0.93). Also, the results show evidence that SafeRefactor is more accurate (0.70) than Ratzinger's approach (0.60). Although close in terms of accuracy, SafeRefactor and Ratzinger have different limitations. While the former had a total of 17 false positives, the latter had just 3. On the other hand, the former had just one false negative, while the latter had 21.

3.5. Threats to validity

There are several limitations to this study. Next we describe some threats to the validity of our evaluation.

3.5.1. Construct validity

To evaluate the correctness of the results of each approach, we created the baseline (see Column *Baseline* of Tables 1 and 2) by comparing the approaches' results since we did not previously know which versions contain behavior-preserving transformations. Therefore, if all approaches present incorrect results, our baseline may also be incorrect. Another threat was our

assumption that changes to non-Java files are refactorings. This may not be true in some cases, such as when a library that the code depends upon is upgraded.

With respect to SafeRefactor, it does not evaluate developer intention to refactor, but whether a transformation changes behavior. Moreover, in the closed world assumption, we have to use the test suite provided by the program that is being refactored. SafeRefactor follows an open world assumption, in which every public method can be a potential target for the test suite generated by Randoop. Randoop may generate a test case that exposes a behavioral change. However, the test case may show an invalid scenario according to the software domain.

3.5.2. Internal validity

The time limit used in SafeRefactor for generating tests may have influence on the detection of non-refactorings. To determine this parameter in our experiment, we compared the test coverage achieved by different values of time limit. Achieving 100% test coverage in real applications is often an unreachable goal; SafeRefactor only analyzes the methods in common of both programs. For each subject, we evaluated one of the selected pairs, and analyzed the statement coverage of the test suite generated by SafeRefactor on the source and the target programs. After increasing the time limit to more than 120 s, the coverage did not present significant variation. So, the value of time limit chosen was 120 s. We follow the same approach used in previous evaluations on Randoop (Robinson et al., 2011).

In 17 changes classified as refactoring by SafeRefactor, our manual analysis showed different change classifications. Some of these changes were not covered by SafeRefactor's test suite. In transformations that only modify a few methods, SafeRefactor considers most methods in common. When this set is large the time limit given to Randoop (120 s) may not be sufficient to generate a test case exposing the behavioral change. As a future work, we intend to improve SafeRefactor by generating tests only for the methods impacted by the transformation (Ren et al., 2004). In this way, we can use SafeRefactor using a smaller time limit.

We used the default value for mostly Randoop parameters. By changing them, we may improve SafeRefactor results. Moreover, since SafeRefactor randomly generates a test suite, there might be different results each time we run the tool. To improve the confidence, we ran SafeRefactor three times to analyze each transformation. If SafeRefactor does not find a behavioral change in all runs, we consider that the transformation to be behavior-preserving. Otherwise, it is classified as a non-behavior-preserving transformation. The tests generated by Randoop had coverage lower than 10% in some versions of JHotDraw. By manually inspecting the tests, we check that they contain calls to JHotDraw's methods that call `System.exit()`, which ends the test execution. As future work, we plan to improve the test execution by avoiding some method calls.

We manually created the buildFiles for JHotDraw, and downloaded the dependencies. We made sure the compilation errors found by SafeRefactor were not related to any missed dependency. We do not have information on the SVN indicating the JDK version used to build the program. By changing the JDK, results may change. Moreover, we run tests using JDK 1.6.

The Murphy-Hill approach was performed by two experienced evaluators. They also have an extensive background in refactoring. The accuracy of this approach may change according to the level of Java expertise of the inspectors.

3.5.3. External validity

We evaluated only two open-source Java projects (JHotDraw and Apache Collections) due to the costs of manual analyses. Our results, therefore, are not representative of all Java projects. To maximize the external validity we evaluated two kinds of software: a GUI application (JHotDraw) and an API (Apache Common Collections).

Randoop does not deal with concurrency. In those situations, SafeRefactor may yield non-deterministic results. Also, SafeRefactor does not take into account characteristics of some specific domains. For instance, currently, it does not detect the difference in the standard output (*System.out.println*) message. Neither could the tool generate tests that exercise some changes related to the graphical interface (GUI) of JHotDraw. These changes may be non-trivial to be tested by using JUnit tests.

Moreover, some changes (Versions 743 and 549) improve the robustness of JHotDraw. Randoop could not generate test cases that produce invalid conditions of JHotDraw to identify these behavioral changes. Also, it seems that some of the bug fixes need complex scenarios to expose behavioral changes. For instance, Version 267 introduces a work-around in one method to avoid a bug in the JDK. Since we may have tested it using a new JDK, probably, the transformation does not change program's behavior. In Version 700, developers change some instructions to assign a copy of the array instead of the array itself. Although this change fixed the array exposure, Randoop could not detect any behavioral change.

Similarly, the manual analysis presents a number of limitations as well. Manually inspecting code leaves room for human error. We only selected changes from two projects (JHotDraw and Collections), which may not be representative of other software projects. In other software domains, it may be harder to understand the logic of the software and define whether the change preserves behavior. Moreover, Java semantics is complex. Even formal refactoring tools may fail to identify whether a transformation preserves behavior (Soares et al., 2012). We tried to mitigate this by having two experienced evaluators simultaneously analyzing the source code. Finally, during our manual analysis, we encountered six very large changes that we were unable to manually inspect completely; in these cases we spent about 30 min manually cataloging refactorings, but did not find any semantics changes in doing so. Had we spent significantly more time inspecting, we may have encountered some non-refactorings. This illustrates that manual inspection, while theoretically quite accurate, is practically difficult to perform thoroughly.

4. Evaluating techniques for identifying applied refactorings

In this section, we evaluate approaches for identifying which refactorings happened between two program versions. First, we show the experiment definition (Section 4.1) and planning (Section 4.2). Then, we present the experiment operation and its results (Section 4.3). We interpret the results and discuss them in Section 4.4. Finally, we present some threats to validity of the experiment (Section 4.5).

4.1. Definition

The goal of this experiment is to analyze two approaches (Murphy-Hill and Ref-Finder) for the purpose of evaluation with respect to identifying which refactorings happened between two program versions from the point of view of researchers in the context of open-source Java project repositories. In this experiment, we address two research questions:

- **Q1**. Do the approaches identify all refactorings that happened between two program versions?

  We use recall to evaluate this question. *tPos* (true positive) represents the refactorings correctly identified by each approach.

fNeg (false negative) represents the refactorings not identified by each approach. Recall is defined as follows (Olson and Delen, 2008):

$$recall = \frac{\#tPos}{\#tPos + \#fNeg} \quad (4)$$

- **Q2**. Do the approaches correctly identify refactorings?

  We use precision to evaluate this question. *fPos* (false positive) represents the refactorings incorrectly identified by each approach. It is defined as follows (Olson and Delen, 2008):

$$precision = \frac{\#tPos}{\#tPos + \#fPos} \quad (5)$$

Although the Murphy-Hill approach identify all changes that happen between two program versions (refactoring and non-refactorings), Ref-Finder only indicates the refactorings. Therefore, we do not calculate the number of true negatives.

4.2. Planning

We used the sample of the previous experiment (see Section 3.2.1), which include 60 JHotDraw and Collections versions.

In this experiment, we evaluate one factor (approaches to identify which refactorings happen between two program versions) with two treatments (Murphy-Hill and Ref-Finder). We choose a paired comparison design for this experiment. The approaches should identify the refactorings that happen between the two program versions of each pair.

We use Ref-Finder 1.0.4 with default configuration. We used the Murphy-Hill approach following the same guidelines presented in Section 3.

As we mentioned in Section 4.1, we choose two metrics, recall and precision to assess and discuss the approaches with respect to our research questions. Since we previously do not know which refactorings happened in the sample, the first author of this article performed an additional manual inspection to compare the approaches' results, and establish a Baseline containing the set of refactorings that were applied in the sample. Then, we compare the results of each approach with respect to the expected results obtained from our baseline.

4.3. Operation

All subjects were downloaded and configured as Eclipse projects, which is needed to perform the analysis of Ref-Finder. We run it on a MacBook Pro Core i5 2.4 GHz and 4GB RAM, running Mac OS 10.7.4.

Table 5 indicates the number of refactorings that happened in each versions (see Column *Baseline*), and the results of Murphy-Hill and Ref-Finder approaches in terms of true positives, false positives, and false negatives. Table 5 presents only the versions where it was found at least one refactoring by Murphy-Hill or Ref-Finder. Additionally, we did not have time to evaluate Versions 324, 357, 501, 518, 596, 814997, 815022, 815042 and 1148801 where Ref-Finder found more than 50 refactorings. We thus did not include these versions in Table 5. Finally, we also excluded Version 637489 since Ref-Finder could not evaluate it.

Furthermore, Table 5 presents the precision and recall of each approach. Ref-Finder identified 19 out of 81 refactorings that happened in the JHotDraw and Collections samples, leading to a recall of 0.24. It also incorrectly identified 35 refactorings (precision of 0.35). On the other hand, the Murphy-Hill approach results were the same of the Baseline, leading to a recall and precision of 1. Only in Version 711140 the detected refactorings matched in both approaches.

Table 5
Summary of results of the experiment.

Version	Baseline	Murphy-Hill	Ref-Finder		
	Refactorings	tPos	tPos	fPos	fNeg
134	1	1	0	0	1
151	3	3	2	0	1
173	6	6	0	0	6
174	3	3	0	0	3
179	7	7	0	0	7
193	9	9	2	6	7
267	2	2	0	0	2
294	0	0	0	4	0
322	0	0	0	1	0
344	11	11	0	0	11
409	1	1	0	0	1
503	7	7	4	3	3
526	0	0	0	2	0
549	4	4	0	1	4
609	2	2	0	0	2
650	1	1	0	2	0
660	23	23	10	15	13
711140	1	1	1	0	0
956279	0	0	0	1	0
Total	81	81	19	35	61
Precision		1.00		0.35	
Recall		1.00		0.24	

We also gathered the kinds of refactorings identified by each approach as shown in Table 6. Column *Refactoring* shows the name of the identified refactoring, and its detected occurrences with the Murphy-Hill and Ref-Finder approaches. While Column *Murphy-Hill* shows the versions where each refactoring was manually identified, Column Ref-Finder shows which refactorings were correctly identified by Ref-Finder (true positive), and also which ones were incorrectly identified (false positive).

For instance, the *Extract method* refactoring was identified twice and three times considering Murphy-Hill and Ref-Finder results, respectively. Two of them were correctly detected by Ref-Finder in Version 151, but one of them in Version 526 was a false positive. By manually inspecting the result, we checked the transformation was not a refactoring. Meanwhile, Murphy-Hill detected six occurrences of the *Remove unused variable* refactoring in Version 173. However, it was not identified by Ref-Finder (a false negative). The *Introduce explaining variable* and *Add parameter* were the refactorings mostly applied in the versions of JHotDraw and Collections analyzed.

Ref-Finder took on average 1 min to check each subject except for Version 637489. After 10 min, we stop its execution. The Murphy-Hill approach took around 15 min to evaluate each transformation.

4.4. Discussion

In the this section, we discuss the false positives and negatives yielded by Ref-Finder (Section 4.4.1). Then, we summarize the answers of the research questions (Section 4.4.2).

4.4.1. Ref-Finder

Ref-Finder incorrectly identified 35 refactorings. The refactoring that generated more false positives was *Move method* with 9 occurrences. Similarly, *Remove Parameter* generated 7 false positives. Also, the *Consolidate duplicate code fragment* refactoring was detected in 4 false positives.

4.4.1.1. False positives. By manually analyzing them, we classified the false positives in three kinds of categories (Table 7): incorrect refactoring identified, incorrect template, and no change identified.

Table 6
Refactorings identified by manual analysis and Ref-Finder.

Refactoring	Murphy-Hill	Ref-Finder	
Name	Version (quantity)	True positive – Version (quantity)	False positive – Version (quantity)
Replace code with method call	134 (1)	–	–
Move operation to listener	151 (1)	–	–
Extract method	151 (2)	151 (2)	526 (1)
Remove unused variable	173 (6)	–	–
Change instance access to static	174 (3)	–	–
Remove immutable object copy	179 (2)	–	–
Replace direct access with getter	179 (5)	–	–
Replace instance of with isInstance	193 (2)	–	–
Add parameter	193 (1), 503 (4), 660 (3)	193 (1), 503 (4), 660 (3)	193 (3), 660 (1)
Remove parameter	193 (1), 660 (2)	193 (1), 660 (2)	193 (3), 503 (3), 660 (1)
Replace field with method	193 (2)	–	–
Decrease method visibility	193 (2)	–	–
Replace direct access with setter	193 (1)	–	–
Inline temp	267 (2), 660 (2)	–	–
Move method	–	–	294 (4),660 (5)
Consolidate duplicate code fragment	–	–	322 (1), 549 (1), 660 (3)
Rename constant	344 (1)	–	–
Rename local variable	344 (4)	–	–
Replace generic cast with classCast	503 (2)	–	–
Replace generic cast with isInstance	503 (1)	–	–
Replace method with method object	–	–	526 (1), 650 (1)
Change statement order	549 (4)	–	–
Swap access method	609 (2)	–	–
Remove duplicate assignment	650 (1)	–	–
Consolidate conditional expression	–	–	650 (1), 660 (1)
Introduce explaining variable	711140 (1), 660 (5), 344 (6)	711140 (1), 660 (4)	–
Remove assignment to parameters	–	–	956279 (1)
Rename class	660 (2)	–	–
Increase method visibility	660 (3)	–	–
Rename method	660 (1)	660 (1)	–
Rename field	660 (1)	–	–
Replace if with switch	660 (1)	–	–
Replace equivalent method call	660 (3)	–	–
Introduce Null object	–	–	660 (2)
Replace magic number with constant	–	–	660 (2)

In Versions 193 and 660, Ref-Finder identified one kind of refactoring, but the actual refactoring was another one. In Version 660, the method `mousePressed` belongs to class `QuaquaTrackListener` in the source program. In the target program, it belongs to the `TrackListener` class. Ref-Finder classified this transformation as the *Move method* refactoring. In fact, this transformation was a *Rename class* refactoring. Although the tool defines an order of the refactorings, the *Rename class* refactoring is not supported by it. That is the reason for this mistake. On the other hand, in Version 193, developers changed the qualified name of the `java.awt.event.MouseEvent` parameter to its simple name `MouseEvent`. Ref-Finder incorrectly classified this transformation as *Remove Parameter* and *Add Parameter* refactorings.

Prete et al. (2010) mention limitations of Ref-Finder with respect to false positives of *Add and Remove Parameters* when renaming parameters. However, notice that in Version 660, the fully qualified name of the parameter did not change, developers just changed the way it was declared. Therefore, it seems to be a bug in Ref-Finder implementation.

In Versions 322, 526, 549, 650, 660, and 956279, Ref-Finder identified refactorings that did not match their refactoring templates. For example, in Version 526, the method `clearDisclosedComponents` was added to the target program, and a call to this method was included inside the method `setEditor`. Ref-Finder incorrectly indicated that this transformation was the *Extract method* refactoring. In fact, no code fragment was extracted from the method `setEditor`. This is an example that Ref-Finder may identify a refactoring that may not preserve behavior. In Version 650, the tool incorrectly detected the *Consolidate Condition Expression* refactoring, which combines a sequence of conditional tests with same result into a single conditional expression. However, the transformation applied to the `ComplexColorWheelImageProducer` class only changes the conditional expression from `(flipY)` to `(!flipY)`. Additionally, the tool incorrectly detected the *Consolidate Duplicate Code Fragment* refactoring in Versions 322, 549, 660, and the *Remove Assignment to Parameters* refactoring in Version 956279. In all of these refactorings, the tool performs a clone detecting technique to check for similarity between method bodies. We performed the experiment with default configurations of Ref-Finder. By increasing the threshold for the algorithm of similarity, we may improve the tool's precision, but we may also decrease its recall, as mentioned by its authors (Prete et al., 2010).

In the last category of false positives, Ref-Finder found a refactoring in a part of the code that did not change in Versions 294, 503, 660. For instance, in Version 660, Ref-Finder detected that the same parameter was removed and added to its method leading to two refactorings (*Add and Remove parameter*). In fact, the code did not change. The other refactorings identified by Ref-Finder that did not change the code were *Move method* and *Consolidate duplicate conditional fragments*. Additionally, Version 294 does not compile. Ref-Finder may incorrectly identify refactoring activities in uncompilable programs

Table 7
False positives of Ref-Finder.

Problem category	Version (quantity)
Incorrect refactoring detected	193 (6), 660 (3)
Incorrect template	322, 526 (2), 549, 650 (2), 660 (7), 956279
No change identified	294 (4), 503 (3), 660 (5)

4.4.1.2. False negatives. With respect to false negatives, Ref-Finder did not find 61 refactorings that were found in the Murphy-Hill approach. However, 50 out of 61 refactorings are not in the original Fowler's catalog, and thus are not supported by Ref-Finder (Table 6). In spite of not being in Fowler's catalog, some of them are common in practice, such as *Rename Class*, *Rename Field* and *Increase and Decrease Method Visibility*.

On the other hand, Ref-Finder did not detected 11 refactorings presented in Fowler's catalog. The *Inline temp* refactoring was not detected by the tool in Versions 267 and 660. Moreover, Ref-Finder did not detect 7 out of 12 *Introduce explaining variable* refactoring presented in Versions 344 and 660.

4.4.2. Answers to the research questions

From the evaluation results, we make the following observations:

- **Q1**. Do the approaches identify all refactorings that happened between two program versions?

  We found evidence that by applying the Murphy-Hill approach, we can identify all of the refactorings applied between two program versions. However, it is a time consuming task, and does not scale. In some of the subjects, it was not possible to analyze all changes. On the other hand, Ref-Finder has the advantage of automating this process, but our results indicate that this tool may miss a number of the refactorings applied in practice since it only focuses on the refactorings catalogued by Fowler (1999). It had a recall of only 0.24.
- **Q2**. Do the approaches correctly identify refactorings?

  In this experiment, the Murphy-Hill approach did not yield any false positive. On the other hand, Ref-Finder had more false positives than true positives (precision of 0.35). Therefore, our results show evidence that this tool has a low precision.

4.5. Threats to validity

There are a number of limitations to this study. Next we describe some threats to the validity of our evaluation.

With respect to construct validity, we yield Column *Baseline* by analyzing the results of Ref-Finder and Murphy-Hill approaches in order to evaluate the results of each approach. Therefore, if the results of both approaches are incorrect, our baseline may also be incorrect. Additionally, our baseline contains refactorings that are not present in Fowler's catalog (Fowler, 1999). Since the current version of Ref-Finder only detects refactorings in this catalog, better precision and recall might have been found if we had considered only those refactorings in our evaluation.

Concerning internal validity, we used Ref-Finder with default parameters. By increasing the threshold for its algorithm of similarity, it may have less false positives, increasing its precision, but it may also have more false negatives, decreasing its recall. Additionally, the Murphy-Hill approach was performed by two experienced evaluators. They also have an extensive background on refactoring. The precision and recall of this approach may change according to the level of Java expertise of the inspectors.

Concerning external validity, similar to the previous experiment, the sample of software repository versions may not be representative for other kinds of software projects. In the same way, the refactorings presented in our sample may not be representative for other behavior-preserving transformations.

5. Related work

In this section, we relate our work to a number of approaches proposed for refactoring detection and practice, and refactoring implementation.

5.1. Refactoring detection and practice

Demeyer et al. (2000) first proposed the idea of inferring refactorings by comparing two programs based on a set of ten characteristic metrics, such as LOC and the number of method calls within a method. Godfrey and Zou (2005) identify merge, split, and rename refactorings based on origin analysis, which serves as a basis of refactoring reconstruction by matching code elements using multiple criteria. Malpohl et al. (2003) present a tool that automatically detects renamed identifiers between pairs of program modules. Van Rysselberghe and Demeyer (2003) use a clone detector to detect moved methods. Antoniol et al. (2004) detects class-level refactorings using a vector space information retrieval approach. It can identify class evolution discontinuities.

Görg and Weißgerber (2005) proposed a technique to identify and rank refactoring candidates using names, signatures, and clone detection results. Later, Weißgerber and Diehl (2006) evolved and evaluated this tool. Weißgerber and Diehl (2006) analyzed the version histories of JEdit, JUnit, and ArgoUML and reconstructed the refactorings performed using the tool proposed before (Görg and Weißgerber, 2005). They also obtained bug reports from various sources. They related the percentage of refactorings per day to the ratio of bugs opened within the next five days. They found that the high ratio of refactoring is sometimes followed by a an increasing ratio of bug reports.

Xing and Stroulia (2006) propose an approach to detect refactoring. They use a tool (UMLDiff (Xing and Stroulia, 2005)) to match program entities based on their name and structural similarity. They compare program versions at the design level. UMLDiff infers refactorings after matching code elements. They can infer 32 refactorings. However, both tools (Weißgerber and Diehl, 2006; Xing and Stroulia, 2006) do not analyze method bodies. So, it does not detect intra-method refactoring changes, such as an Inline Temp refactoring.

Prete et al. (2010) propose Ref-Finder, which we give an overview in Section 2.4, and evaluate in Section 4. Ref-Finder can detect 63 refactoring types from Fowler's catalog (Prete et al., 2010). It can detect all refactorings of the previous works, and it can detect intra-method refactoring changes. A more comprehensive comparison can be found in Prete et al. (2010).We use Ref-Finder in our evaluation in Section 4.

Prete et al. (2010) evaluated Ref-Finder in toy examples and real open source projects, and found an overall precision of 0.79 and Recall of 0.95. They calculated the recall by comparing the results of two executions of Ref-Finder. For each subject, they executed Ref-Finder with a threshold for its similarity algorithm of 0.65, manually analyzed the results, and identified 10 correct refactorings. Then, they executed Ref-Finder with the default threshold of 0.85 and checked whether the tool detected the correct refactorings previously identified. By doing so, they only check the recall against supported refactorings previously detected by Ref-Finder. In our experiment, we check Ref-Finder recall against a Baseline. We found a recall of only 0.25 for Ref-Finder. The main reason for that is that most of the refactorings applied in our subjects are not supported by Ref-Finder. We found evidence from our experiment, differently from the previous one, that Ref-Finder may miss to detect a number of refactorings, and incorrectly identify others.

Ratzinger et al. (2008) analyzed the relationship between refactoring and software defects. They proposed an approach to automatically identify refactorings based on commit messages, which we describe in Section 2.3. Using evolution algorithms, they confirmed the hypothesis that the number of software defects in the period decreases if the number of refactorings increases as overall change type. To evaluate the effectiveness of the commit message analysis, they randomly sampled 500 versions from 5 projects, and analyzed whether their analysis correctly classify each version. In

their experiment, the commit message analysis had only 4 false positives 10 false negatives in 500 software versions, leading to a high precision and recall. In this article, we compared Ratzinger approach with other two approaches (Murphy-Hill and SafeRefactor) in a random sample of 60 versions. Differently from the original work, our results show a low recall and precision of Ratzinger approach, which we discuss in Section 3.4.3.

Murphy-Hill et al. (2009, 2012) evaluated nine hypotheses about refactoring activities. They used data automatically retrieved from users through Mylyn Monitor and Eclipse Usage Collector. That data allowed Murphy-Hill et al. to identify the frequency of each automated refactoring. The most frequently applied refactorings are: Rename, Extract local variable, Move, Extract method, and Change method signature. They confirmed assumptions such as the fact that refactorings are frequent. Data gathered from Mylyn showed that 41% of the programming sessions contained refactorings.

Additionally, they evaluated the Ratzinger analysis. By using Ratzinger algorithm, they classified the refactoring versions from Eclipse CVS repository. Then, they randomly selected 20 versions from each set of refactoring versions and non-refactoring versions identified by Ratzinger, and applied to these 40 versions the manual inspection proposed by them, which we describe in Section 2.2. From the 20 versions labeled as refactoring by Ratzinger, only 7 could be classified as refactoring versions. The others include non-refactoring changes. On the other hand, the 20 versions classified as non-refactoring by Ratzinger were correct. In this article, we compared the results of these two techniques (Ratzinger and Murphy-Hill) with SafeRefactor's results (Section 3). The Murphy-Hill approach was the most accurate among the refactoring technique we evaluated. However, it incorrectly classified versions containing compilation errors as refactoring versions.

Soares et al. (2011) propose an approach using SafeRefactor to identify refactoring versions. In that previous paper, five repositories have been analyzed with respect to refactoring frequency, granularity and scope. In this article, we use part of that infrastructure for comparing the automatic approach with other approaches (Murphy-Hill and Ratzinger). The focus is the comparison between these approaches to detect behavior-preserving transformations in software repositories.

Kim et al. (2011) investigate the relationship of API-level refactorings and bug fixes in three open source projects. They use a tool (Kim et al., 2007) to infer systematic declaration changes as rules and determine method-level matches (a previous version of Ref-Finder (Prete et al., 2010) that identifies 11 refactorings). They found that the number of bug fixes increases after API-level refactorings while the time taken to fix them decreases after refactorings. Moreover, the study indicated that refactorings are performed more often before major releases than after the releases. Rachatasumrit and Kim (2012) analyze the relationship between the types and locations of refactorings identified by Ref-Finder and the affecting changes and affected tests identified by a change impact analyzer (FaultTracer). They evaluate their approach in three open source projects (Meter, XMLSecurity, and ANT) and found that refactoring changes are not very well tested. By selecting the test cases that only exercise the changes, we may decrease the regression test cost. In this article, we compare Ref-Finder and the Murphy-Hill approach to detect refactorings in some versions of JHotDraw and the Apache Common Collections. Ref-Finder did not find some refactorings and incorrectly identified others.

Kim et al. (2012) interview a subset of engineers who led the Windows refactoring effort and analyzed Windows 7 version history data. They found that in practice developers may allow non-behavior-preserving program transformations during refactoring activities. Moreover, developers indicate that refactoring involves substantial cost and risks. By analyzing Windows 7 version history, the study indicated that refactored modules experienced higher reduction in the number of inter-module dependencies and post-release defects than other changed modules. In our work, we do not evaluate developer's intention and whether the refactoring improves program's quality.

Dig et al. (2006) created an automatic refactoring detection tool, called RefactoringCrawler, targeting software components and addressing the maintenance of software that uses components when these components evolve. His technique generates logs of detected changes. Any software that uses the component should use these logs to replicate the changes that occurred within the component in order to remain compatible with it. This technique uses an automatic refactoring detector, based on a two-phase algorithm. First, it performs a syntactic identification of possible refactorings. Then it performs a semantic evaluation of the possible refactorings identified in the first phase. It is able to detect 14 types of refactorings. On its evaluation, RefactoringCrawler was executed over two different versions of four open source programs. Compared to a previous manual analysis (Dig and Johnson, 2005), RefactoringCrawler succeeded in finding 90% of the refactorings supported by the tool. In this article, we evaluated Ref-Finder, a similar tool but can detect many more types of refactorings, if compared to RefactoringCrawler.

Murphy-Hill et al. (2008) proposed several hypotheses related to refactoring activity, and outline experiments for testing those hypotheses. They categorized four approaches to analyze refactoring activity: analysis of the source code repository, analysis of repository commit logs, observation of programmers, and analysis of refactoring tool usage logs. They suggest which analysis should be used to test the hypotheses. Their method has the advantage of identifying each specific refactoring performed.

Vakilian et al. (2012) conducted a field study on programmers in their natural settings working on their code. Based on their quantitative data and interviews, they found a number of factors that affect the appropriate and inappropriate uses of automated refactorings. In our work, we do not evaluate automated tools. We focus on identifying behavior-preserving transformations, and which refactorings are applied between two program versions.

5.2. Refactoring implementation

Defining and implementing refactoring preconditions is non-trivial. Preconditions are a key concept of research studies on the correctness of refactorings. Opdyke (1992) proposes a number of refactoring preconditions to guarantee behavior preservation. However, there was no formal proof of the correctness and completeness of these preconditions. In fact, later, Tokuda and Batory (2001) showed that Opdyke's preconditions were not sufficient to ensure preservation of behavior.

Proving refactorings with respect to a formal semantics is a challenge (Schäfer et al., 2008). Some approaches have been contributing in this direction. Borba et al. (2004) propose a set of refactorings for a subset of Java with copy semantics (ROOL). They prove the refactoring correctness based on a formal semantics. Silva et al. (2008) propose a set of behavior-preserving transformation laws for a sequential object-oriented language with reference semantics (rCOS). They prove the correctness of each one of the laws with respect to rCOS semantics. Some of these laws can be used in the Java context. Yet, they have not considered all Java constructs, such as overloading and field hiding.

Furthermore, Schäfer et al. (2009) and Schäfer and de Moor (2010) present a number of Java refactoring implementations. They translate a Java program to an enriched language that is easier to specify and check conditions, and apply the transformation. As correctness criteria, besides using name binding preservation, they used other invariants such as control flow and data flow preservation. Another specialized approach for testing

refactorings – generalization-related refactorings such as Extract Interface and Pull Up Method – is proposed by Tip et al. (2003). Their work proposes an approach that uses type constraints to verify preconditions of those refactorings, determining which part of the code they may modify. Steimann and Thies (2009) show that by changing access modifiers (`public`, `protected`, `package`, `private`) in Java one can introduce compilation errors and behavioral changes. They propose a constraint-based approach to specify Java accessibility, which favors checking refactoring preconditions and computing the changes of access modifiers needed to preserve the program behavior. Such specialized approach is extremely useful for detecting bugs regarding accessibility-related properties.

Soares et al. (2010) present a tool called SafeRefactor for improving safety during refactoring activities. It was evaluated in 7 transformations applied to real case studies. For example, Soares et al. (2011) propose a technique to identify overly strong conditions based on differential testing (Mckeeman, 1998). If a tool correctly applies a refactoring according to SafeRefactor and another tool rejects the same transformation, the latter has an overly strong condition. In a sample of 42,774 programs generated by JDolly (the program presented in Listing 1 is generated by JDolly), they evaluated 27 refactorings in Eclipse JDT, NetBeans and JRRT, and found 17 and 7 types of overly strong conditions in Eclipse JDT and JRRT, respectively. This approach is useful for detecting whether the set of refactoring preconditions is minimal. Moreover, Soares et al. (2012) present a technique to test Java refactoring engines. It automates test input generation by using a Java program generator that exhaustively generates programs for a given scope of Java declarations. The technique uses SafeRefactor, a tool for detecting behavioral changes, as oracle to evaluate the correctness of these transformations. Finally, the technique classifies the failing transformations by the kind of behavioral change or compilation error introduced by them. They have evaluated this technique by testing 29 refactorings in Eclipse JDT, NetBeans and the JastAdd Refactoring Tools. They analyzed 153,444 transformations, and identified 57 bugs related to compilation errors, and 63 bugs related to behavioral changes. In this work, we evaluate three different approaches to identify behavior-preserving transformations in software repositories. We found some limitations of using SafeRefactor in real case studies (see Section 3.4.2).

6. Conclusions

In this article, we conducted two experiments to evaluate approaches for identifying refactoring activities on software repositories. The first experiment aims at evaluating three different approaches (SafeRefactor, Murphy-Hill, and Ratzinger) to identify behavior-preserving transformations. Moreover, the second experiment evaluates two approaches (Murphy-Hill and Ref-Finder) to identify which refactorings happen between two program versions. These approaches were evaluated in a sample containing 40 pairs of versions from JHotDraw and 20 from Apache Common Collections.

In our experiments, we found evidence that the Murphy-Hill approach is the most reliable approach in detecting behavior-preserving transformations and refactorings applied. However it is time-consuming; in large changes, it may not be able to evaluate the whole transformation. Additionally, it may incorrectly evaluate uncompilable programs, and its accuracy depends on developers' experience.

With respect to the automated approaches, our results show evidence that SafeRefactor can detect almost all behavior-preserving transformations; it had a recall of 0.96. However, the results also indicate that the tool is not very precise (precision of 0.59). It may not detect a number of non-behavior-preserving transformations due to the limitations of its test suite generator. In our experiment, it could not generate a test case exposing behavioral change for some programs containing graphical user interface, manipulating files.

On the other hand, we found evidence that the Ratzinger approach may fail to detect a number of behavior-preserving transformations. In our experiment, most of them were categorized as non-behavior-preserving transformations by this approach. It had low recall (0.16) and only average precision (0.57). It depends on guidelines that must be followed by developers during software development.

Finally, Ref-Finder identified only 24% of the refactorings presented in our experiment. Moreover, 65% of the refactorings detected by Ref-Finder were incorrect. It is not simple to identify which refactorings were applied statically based on template matching. Some refactoring templates are similar, and the tool incorrectly identified some of them.

Currently, to the best of our knowledge, there is no tool that assures the correctness of a transformation (behavior preservation). Moreover, due to the complexity of Java semantics, it is also non-trivial to manually evaluate various aspects of a language, such as: accessibility, types, name binding, data flow, concurrency and control flow. We have to take them into consideration in the results of our experiment.

As future work, we aim to evaluate the approaches for a more even distribution of selected versions of the chosen subjects. We also intend to evaluate all approaches in more software repositories. Additionally, we plan to investigate whether combining the techniques would be useful. To improve the confidence of SafeRefactor, we intend to incorporate impact analysis to generate tests only for the entities affected by the transformation. We also plan to check the test coverage regarding the entities impacted by the transformation as proposed by Wloka et al. (2010). Finally, we intend to evaluate SafeRefactor using other test suite generators than Randoop.

Acknowledgment

We gratefully thank Paulo Borba, Sérgio Soares, Tiago Massoni, Bruno Catão, Catuxe Varjão, Solon Aguiar, Alessandro Garcia, and the anonymous referees from the Brazilian Symposium on Software Engineering and *Journal of Systems and Software* for their useful suggestions. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq grants 573964/2008-4, 304470/2010-4, and 480160/2011-2.

Appendix A. Algorithms

Next we formalize some algorithms used to collect data from repository. Algorithm 2 indicates when a transformation is low or high-level. If a transformation only changes inside a method, it is considered low-level. Otherwise it is considered high-level. *methods* yields the set of methods of a program. *signature* yields the method signature of all methods received as parameter.

Algorithm 2. Refactoring Granularity

Input: source ⇐ program before transformation
Input: target ⇐ program after transformation
Output: Indicates whether a transformation is low or high-level
  mSource ⇐ methods(*source*)
  mTarget ⇐ methods(*target*)
  if signature(*mSource*) = signature(*mTarget*) **then**
    LOW
  else
    HIGH
  end if

Algorithm 3 establishes when a transformation is local or global. If a transformation only changes at most one package, it is considered local. Otherwise it is considered global. *packages* yields the set of packages of a program. *name* yields the name of a package. *diff* is the shell command used to compare to directories.

Algorithm 3. Refactoring Scope

```
Input: source ⇐ program before transformation
Input: target ⇐ program after transformation
Output: Indicates whether a transformation is local or global
  count ⇐ 0
  foreach p ∈ packages(source) do
    pTarget ⇐ package(name(p),target)
    if diff(p,pTarget) ≠ ∅ then
      count++
    end if
  end foreach
  foreach p∈ packages(target) do
    pSource ⇐ package(name(p),source)
    if diff(p,pSource) ≠ ∅ then
      count++
    end if
  end foreach
  if count ≤ 1 then
    LOCAL
  else
    GLOBAL
  end if
```

References

Antoniol, G., Penta, M., Merlo, E., 2004. An automatic approach to identify class evolution discontinuities. In: Proceedings of the 7th International Workshop on Principles of Software Evolution, IWPSE '04, IEEE Computer Society, Washington, DC, USA, pp. 31–40.

Basili, V.R., Selby, R.W., Hutchens, D.H., 1986. Experimentation in software engineering. IEEE Transactions on Software Engineering 12 (7), 733–743.

Borba, P., Sampaio, A., Cavalcanti, A., Cornélio, M., 2004. Algebraic reasoning for object-oriented programming. Science of Computer Programming 52, 53–100.

Demeyer, S., Ducasse, S., Nierstrasz, O., 2000. Finding refactorings via change metrics. In: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '00, ACM, New York, NY, USA, pp. 166–177.

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

Dig, D., Comertoglu, C., Marinov, D., Johnson, R.,2006. Automated detection of refactorings in evolving components. In: Proceedings of the 20th European Conference on Object-Oriented Programming, ECOOP '06. Springer-Verlag, Berlin, Germany, pp. 404–428.

Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA.

Görg, C., Weißgerber, P., 2005. Detecting and visualizing refactorings from software archives. In: Proceedings of the 13th International Workshop on Program Comprehension, IWPC '05, IEEE Computer Society, Washington, USA, pp. 205–214.

Godfrey, M., Zou, L., 2005. Using origin analysis to detect merging and splitting of source code entities. IEEE Transactions on Software Engineering 31 (2), 166–181.

Henkel, J., Diwan, A., 2005. CatchUp!: capturing and replaying refactorings to support API evolution. In: Proceedings of the 27th International Conference on Software Engineering, ICSE '05, ACM, New York, NY, USA, pp. 274–283.

Kim, M., Notkin, D., Grossman, D., 2007. Automatic inference of structural changes for matching across program versions. In: Proceedings of the 29th International Conference on Software Engineering, ICSE '07, IEEE Computer Society, Washington, DC, USA, pp. 333–343.

Kim, M., Cai, D., Kim, S., 2011. An empirical investigation into the role of API-level refactorings during software evolution. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, ACM, New York, NY, USA, pp. 151–160.

Kim, M., Zimmermann, T., Nagappan, N., 2012. A field study of refactoring challenges and benefits. In: Proceedings of the 20th Foundations of Software Engineering, FSE '12, ACM, New York, NY, USA.

Malpohl, G., Hunt, J., Tichy, W., 2003. Renaming detection. Automated Software Engineering 10, 183–202.

Mckeeman, W., 1998. Differential testing for software. Digital Technical Journal 10 (1), 100–107.

Murphy-Hill, E., Black, A., 2008. Refactoring tools: fitness for purpose. IEEE Software 25 (5), 38–44.

Murphy-Hill, E., Black, A., Dig, D., Parnin, C., 2008. Gathering refactoring data: a comparison of four methods. In: Proceedings of the 2nd Workshop on Refactoring Tools, WRT '08, pp. 1–5.

Murphy-Hill, E., Parnin, C., Black, A.P., 2009. How we refactor, and how we know it. In: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, Washington, DC, USA, pp. 287–296.

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

Olson, D.L., Delen, D., 2008. Advanced Data Mining Techniques. Springer, Berlin, Heidelberg.

Opdyke, W., 1992. Refactoring object-oriented frameworks. Ph.D. Thesis. University of Illinois at Urbana-Champaign.

Pacheco, C., Lahiri, S., Ernst, M., Ball, T., 2007. Feedback-directed random test generation. In: Proceedings of the 29th International Conference on Software Engineering, ICSE '07, IEEE Computer Society, Washington, DC, USA, pp. 75–84.

Prete, K., Rachatasumrit, N., Sudan, N., Kim, M., 2010. Template-based reconstruction of complex refactorings. In: Proceedings of the 26th IEEE International Conference on Software Maintenance, ICSM '10, IEEE Computer Society, Washington, DC, USA, pp. 1–10.

Rachatasumrit, N., Kim, M., 2012. An empirical investigation into the impact of refactoring on regression testing. In: Proceedings of the 28th IEEE International Conference on Software Maintenance, ICSM '12, IEEE Computer Society, Washington, USA.

Ratzinger, J., Sigmund, T., Gall, H., 2008. On the relation of refactorings and software defect prediction. In: Proceedings of the 5th Mining Software Repositories, MSR '08, pp. 35–38.

Ratzinger, J., 2007. sPACE: software project assessment in the course of evolution. Ph.D. Thesis. Vienna University of Technology.

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

Robinson, B., Ernst, M., Perkins, J., Augustine, V., Li, N., 2011. Scaling up automated test generation: automatically generating maintainable regression unit tests for programs. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, IEEE Computer Society, Washington, DC, USA, pp. 23–32.

Schäfer, M., de Moor, O., 2010. Specifying and implementing refactorings. In: Proceedings of the 25th ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '10, ACM, New York, NY, USA, pp. 286–301.

Schäfer, M., Ekman, T., de Moor, O., 2008. Challenge proposal: verification of refactorings. In: Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification, PLPV '09, ACM, New York, NY, USA, pp. 67–72.

Schäfer, M., Ekman, T., de Moor, O., 2008. Sound and extensible renaming for Java. In: Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '08, ACM, New York, NY, USA, pp. 277–294.

Schäfer, M., Verbaere, M., Ekman, T., Moor, O.,2009. Stepping stones over the refactoring rubicon. In: Proceedings of the 23rd European Conference on Object-Oriented Programming, ECOOP '09. Springer-Verlag, Berlin, Heidelberg, pp. 369–393.

Silva, L., Sampaio, A., Liu, Z., 2008. Laws of object-orientation with reference semantics. In: Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods, SEFM '08, IEEE Computer Society, Washington, DC, USA, pp. 217–226.

Soares, G., Gheyi, R., Serey, D., Massoni, T., 2010. Making program refactoring safer. IEEE Software 27, 52–57.

Soares, G., Mongiovi, M., Gheyi, R., 2011. Identifying overly strong conditions in refactoring implementations. In: Proceedings of the 27th IEEE International Conference on Software Maintenance, ICSM '11, IEEE Computer Society, Washington, DC, USA, pp. 173–182.

Soares, G., Catão, B., Varjão, C., Aguiar, S., Gheyi, R., Massoni, T., 2011. Analyzing refactorings on software repositories. In: Proceedings of the 25th Brazilian Symposium on Software Engineering, SBES '11, IEEE Computer Society, Washington, DC, USA, pp. 164–173.

Soares, G., Gheyi, R., Massoni, T., 2012. Automated behavioral testing of refactoring engines. IEEE Transactions on Software Engineering 99, http://dx.doi.org/10.1109/TSE.2012.19.

Steimann, F., Thies, A.,2009. From public to private to absent: refactoring Java programs under constrained accessibility. In: Proceedings of the 23rd European Conference on Object-Oriented Programming, ECOOP '09. Springer-Verlag, Berlin, Germany, pp. 419–443.

Tip, F., Kieżun, A., Bäumer, D., 2003. Refactoring for generalization using type constraints. In: Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '03, ACM, New York, NY, USA, pp. 13–26.

Tokuda, L., Batory, D., 2001. Evolving object-oriented designs with refactorings. Automated Software Engineering 8, 89–120.

Vakilian, M., Chen, N., Negara, S., Rajkumar, B.A., Bailey, B.P., Johnson, R.E.,2012. Use, disuse, and misuse of automated refactorings. In: Proceedings of the 34th International Conference on Software Engineering, ICSE '12. IEEE Press, Piscataway, NJ, USA, pp. 233–243.

Van Rysselberghe, F., Demeyer, S., 2003. Reconstruction of successful software evolution using clone detection. In: Proceedings of the 6th International Workshop on Principles of Software Evolution, IWPSE '03, IEEE Computer Society, Washington, DC, USA, pp. 126–130.

Weißgerber, P., Diehl, S., 2006. Are refactorings less error-prone than other changes? In: Proceedings of the 3rd Mining Software Repositories, MSR '06, ACM, New York, NY, USA, pp. 112–118.
Weißgerber, P., Diehl, S., 2006. Identifying refactorings from source-code changes. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, ASE '06, IEEE Computer Society, Washington, DC, USA, pp. 231–240.
Wloka, J., Host, E., Ryder, B., 2010. Tool support for change-centric test development. IEEE Software, 66–71.
Xing, Z., Stroulia, E., 2005. UMLDiff: an algorithm for object-oriented design differencing. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05, ACM, New York, NY, USA, pp. 54–65.
Xing, Z., Stroulia, E., 2006. Refactoring detection based on UMLDiff change-facts queries. In: Proceedings of the 13th Working Conference on Reverse Engineering, WCRE '06, IEEE Computer Society, Washington, DC, USA, pp. 263–274.

Gustavo Soares is a PhD student in the Department of Computer Science at Federal University of Campina Grande. His research interests in software engineering include software development, evolution and verification. He holds a MSc in Computer Science from the Federal University of Campina Grande, and is a member of the IEEE and ACM. More on Soares can be found at http://www.dsc.ufcg.edu.br/gsoares.

Rohit Gheyi is a professor in the Department of Computer Science at Federal University of Campina Grande. His research interests include refactorings, formal methods and software product lines. He holds a Doctoral degree in Computer Science from the Federal University of Pernambuco, and is a member of the ACM. More on Gheyi can be found at http://www.dsc.ufcg.edu.br/ rohit.

Emerson Murphy-Hill is an Assistant Professor of Computer Science at NCSU. He received a PhD from Portland State University under Professor Andrew P. Black in 2008 and completed a postdoc with Professor Gail C. Murphy at the University of British Columbia in 2010. His research interests lie in the intersection of software engineering and human-computer interaction. He has received three ACM SIGSOFT Distinguished Paper Awards for his work on software engineering tools. More on Murphy-Hill can be found at http://people.engr.ncsu.edu/ermurph3/.

Brittany Johnson is a PhD student at North Carolina State University under Emerson Murphy-Hill. She received her Bachelor's of Arts in Computer Science from College of Charleston in 2007. Her research interests are in software engineering and human-computer interaction. More on Brittany can be found at http://www4.ncsu.edu/ bijohnso.