Analyzing Refactorings on Software Repositories

Gustavo Soares, Bruno Catão, Catuxe Varjão, Solon Aguiar, Rohit Gheyi, and Tiago Massoni
Department of Computing and Systems
Federal University of Campina Grande
Email: {gsoares, rohit, massoni}@dsc.ufcg.edu.br,
{catuxe, catao}@copin.ufcg.edu.br, solon@computacao.ufcg.edu.br

Abstract—**Currently analysis of refactoring in software repositories is either manual or only syntactic, which is time-consuming, error-prone, and non-scalable. Such analysis is useful to understand the dynamics of refactoring throughout development, especially in multi-developer environments, such as open source projects. In this work, we propose a fully automatic technique to analyze refactoring frequency, granularity and scope in software repositories. It is based on SAFEREFACTOR, a tool that analyzes transformations by generating tests to detect behavioral changes – it has found a number of bugs in refactoring implementations within some IDEs, such as Eclipse and Netbeans. We use our technique to analyze five open source Java projects (JHotDraw, ArgoUML, SweetHome 3D, HSQLDB and jEdit). From more than 40,723 software versions, 39 years of software development, 80 developers and 1.5 TLOC, we have found that: 27% of changes are refactorings. Regarding the refactorings, 63,83% are Low level, and 71% have local scope. Our results indicate that refactorings are frequently applied before likely functionality changes, in order to better prepare design for accommodating additions.**

I. INTRODUCTION

Refactoring is the process of changing a software for evolving its design while preserving its behavior [1]. In practice, developers perform refactorings either manually – which is error-prone and time consuming – or with the help of IDEs with refactoring support, such as Eclipse and Netbeans. 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. Understanding the dynamics of software evolution surely helps the conception of specific methods and tools. A number of studies have performed such investigations in the context of refactoring [2], [3], [4], [5].

Open source projects are an appropriate and manageable source of information about software development and evolution. However, manually inspecting these sources is error-prone, time-consuming and not scalable. It is almost infeasible to analyze the complete development history. As an example, Murphy-Hill et al. [2] was able to manually analyze 20 pairs of versions with up to 15 KLOC from one open source repository, in order to identify refactoring activities. Likewise, static analysis of program versions is not able to evaluate behavior preservation between pairs of repository versions, which identifies a refactoring application. As a consequence, the gathered information can be inaccurate, leading to inconclusive results. Dig et al. [5] propose such a detector, which identifies only seven kinds of refactorings, and their static analysis do not avoid false positives.

In this paper, we present a *fully automatic* technique for analyzing refactoring activities with respect to frequency, granularity, and scope, over entire repositories history. The technique takes available repository source code and configuration files as input. Then, every pair of consecutive versions is analyzed, and non-refactoring transformations are identified. In this step, we use SAFEREFACTOR (Section II), a tool for detecting behavioral changes. SAFEREFACTOR analyzes a transformation and generates tests for checking whether the behavior was preserved between two versions of the repository. It has been useful in the past for finding behavioral changes in refactorings performed in real case studies [6].

If no behavioral change is detected, the confidence on behavior preservation is higher, then we classify the transformation as a refactoring. We categorize each refactoring in terms of granularity (Low or High-level refactorings and the size of the transformation) and scope (Global and Local refactorings). We describe the technique and this classification in Section III.

We use our technique to analyze[1] the refactorings in five open-source Java projects (JHotDraw, ArgoUML, Sweet Home 3D, jEdit and HSQLDB). From more than 40,723 software versions, 39 years of software development, 80 developers and 1.5 TLOC, we have found that 73% of the transformations are not refactorings. Regarding the likely refactorings, 63,83% of them are Low level (changes inside methods body), and 71% are Local, which means changes that affect only one package.

Our findings corroborate with some previously published hypotheses about refactoring frequency. We have found that refactorings correspond to 27% of software maintenance in the analyzed subjects. This is a similar result from Murphy-Hill et al. [2]. However, they state that most of refactorings are High level, that is, change class, methods, or field signatures (e.g. add parameter), different from our work. While they evaluated the developers' intention of applying refactorings, we evaluated whether the transformations preserve behavior. We believe that, despite developers' intention of applying High-level refactorings, successfully applied refactorings are mostly Low-level.

In summary, the main contributions of this paper are the following:

[1]All experiment data is available at: http://dsc.ufcg.edu.br/~spg/papers.html

- A technique that allows analysis of refactoring activities with respect to frequency, granularity, and scope, over entire repositories history (Section III);
- We use our technique to analyze five open source Java projects repositories (JHotDraw, ArgoUML, SweetHome 3D, jEdit and HSQLDB) (Section IV);
- Based on our evaluation, we are able to draw the following conclusions on refactorings: they correspond to about 27% of the changes during the software evolution; they are most commonly used to reach an specific evolution task, such as add a feature; about 63,83% and 71% of them are low level and Local, respectively.

II. SafeRefactor

In this section, we show an overview of SafeRefactor [6], whose objective is to detect behavioral changes during refactoring activities. It receives source code and a refactoring to be applied as input, analyzes the transformation, generates tests, and then reports whether it is safe to apply the transformation. First we present a non-behavior-preserving transformation example. Next we use this example to explain how SafeRefactor detects behavioral changes.

In general, each refactoring may contain a number of *preconditions* to preserve the observable behavior. For instance, to rename an attribute, name conflicts must not be present. However, mostly refactoring tools do not implement all preconditions, because it is far from trivial to completely specify those conditions [7]. Therefore, often refactoring tools allow wrong transformations to be applied with no warnings whatsoever. For instance, Figure 1 shows a transformation [8] applied by the Eclipse 3.4.2 IDE as a refactoring, but actually changing the program's behavior. Listing 1 shows a program containing the class `A` and its subclass `B`. The method `test` yields 10. When we apply the pull up refactoring to the method `k(int)` using Eclipse, the resulting code is presented in Listing 2. The method `test` in the target program yields `20`, instead of `10`. Therefore, the transformation does not preserve behavior using the Eclipse 3.4.2 IDE.

Suppose that we use SafeRefactor in the transformation described in Figure 1. The process is composed of five sequential steps for each refactoring application under test (Figure 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 in both source and target programs (Step 1). Step 2 aims at generating unit tests for methods identified in Step 1. It uses Randoop [9] 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.

In his seminal work on refactoring, Opdyke [10] 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, 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. Consider a renaming method refactoring from `A.m(..)` to `A.n(..)`. The set of methods identified in Step 1 includes none of them. A similar thing occurs when 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 [11]. This feature is a potential source of problems.

SafeRefactor has been used to evaluate seven real case study refactorings (from 3 to 100 KLOC) [6]. For instance, it analyzed JHotDraw (23 KLOC) and its refactored version, and automatically detected a behavioral change. This problem was not identified by developers using refactoring tools and JHotDraw's test suite. Moreover, SafeRefactor has detected more than 50 bugs [12] in refactorings implemented by IDEs, such as Eclipse and Netbeans, showing it applicability in detecting behavior changes.

III. Technique

In this section, we present our technique that analyzes refactorings in open source software repositories relative to three properties:

- *Frequency*. Measures how often refactorings were applied over the project's lifetime;
- *Granularity*. Evaluates refactorings on its impact over the program structure – whether it affects only the internals of a method, for instance, or spans over several methods and classes;
- *Scope*. Defines whether a refactoring spans over a single package or affects multiple packages.

Our technique uses, as input, repository source code and their configuration files – for instance, `build.xml`. As result, it reports the total number of refactorings, the granularity and the scope of these refactorings. The technique process consists of three major steps. The first step analyzes each pair of consecutive versions and classifies it as non-refactoring or refactoring. Then, we analyze the identified refactorings with respect to granularity (Step 2) and scope (Step 3).

The first step evaluates whether a transformation is a refactoring. In this work, we use a strict notion of refactoring; a transformation is considered a refactoring if SafeRefactor

Listing 1. Source Program

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

Listing 2. Target Program

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

Fig. 1. Pull up method refactoring enables overloading

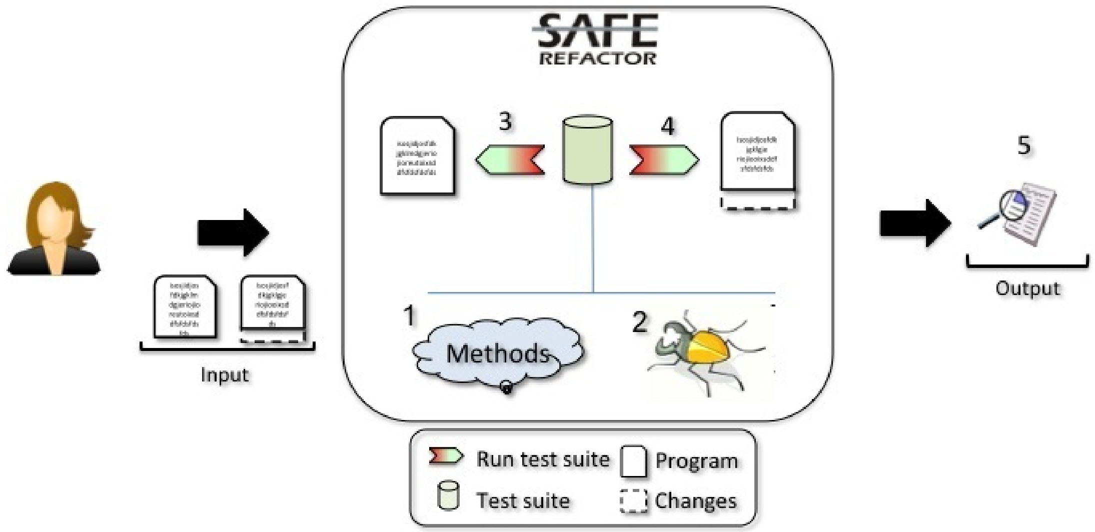


Fig. 2. The SafeRefactor major steps. 1. It identifies common methods in the source and target programs; 2. The tool generates unit tests using our modified Randoop; 3. It runs the test suite on the source program; 4. The tool runs the test suite on the target program; 5. The tool shows whether the transformation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

does not detect a behavioral difference between source and target programs. Otherwise, it is discarded as refactoring, despite the developer's intention of applying a refactoring. As a consequence, our study considers exclusively effectively applied refactorings. Although SafeRefactor does not exhaustively guarantee that transformations are indeed refactoring, confidence on correctness is considerably higher [6].

We consider only transformations between two consecutive pairs of versions. For each pair, if we do not detect a behavioral change, the transformation is a refactoring. Even though refactorings must also improve the internal structure of the code, in this work we focus on behavior preservation.

Next we formalize the process of detecting behavioral changes with SafeRefactor. The function $Evaluation$ receives a version v_i, where i corresponds to the i^{th} repository version, and yields whether v_i and v_{i+1} have the same behavior. If SafeRefactor does not find a behavioral change, the function evaluates to $true$ (considered to be a refactoring).

$$\begin{array}{l} Version : TYPE \\ Evaluation(v_i : Version) : boolean = \\ \quad SafeRefactor(v_i, v_{i+1}) \end{array}$$

SafeRefactor generates a test suite for each pair of versions and reports whether behavioral changes are detected, as formalized next. This test suite covers all methods in common between the two versions. If all tests yield the same results for both versions, SafeRefactor considers that the transformation preserves behavior. $randoop$ produces the generated test suite, while $exec$ represents either success or failure of these tests, based on the oracle generated by

SafeRefactor, which detects behavioral differences.

$$
\begin{array}{l}
Test : TYPE \\
Method : TYPE \\
SafeRefactor(v, v' : Version) : boolean = \\
\quad \forall t : Test \bullet \\
\quad\quad t \in randoop(v, commonMethods(v, v')) \Rightarrow \\
\quad\quad\quad exec(v, t) = exec(v', t)
\end{array}
$$

$$
\begin{array}{l}
commonMethods(v, v' : Version) : \mathcal{P}[Method] \\
exec(v : Version, t : Test) : boolean \\
randoop(v : Version, m : \mathcal{P}[Method]) : \mathcal{P}[Test]
\end{array}
$$

Before running the tests, our technique compiles the versions based on the `build.xml` files received as parameters. Uncompilable versions are discarded. Data on test coverage is collected to improve confidence on test results. By detecting refactorings, we calculate the frequency of refactorings in the repository.

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, refactorings [1] Extract Method, Rename Method, Add Parameter are High level. On the other hand, *Low level* refactorings change blocks of code within methods, such as: Rename Local Variable, and Extract Local Variable [1].

In order to measure granularity, we statically analyze the identified refactorings with respect to classes, fields, and methods signatures. If both versions contain different set of signatures, we classify the refactoring as High level, otherwise as Low level. We use the Unix `diff` tool with no parameters to collect the number of lines of code that changed between each pair of versions.

Finally, Step 3 collects the refactoring scope. For this purpose we classify refactoring scope as Local or Global. *Global* refactorings affect classes on more than one package. For example, if there is a client using a 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. The whole technique is illustrated in Figure 3.

Empirical analysis of refactoring tasks in open source 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. There is a number of studies that observed a high incidence of refactoring in open source projects [2], [4], based either on manual analysis or parsing of commit messages. Our automatic approach, in contrast, shows different results for the analyzed projects. Concerning software evolution, specific methods can be conceived, specially in open-source contexts.

IV. Evaluation

In this section we use our technique to evaluate five software repositories with respect to refactoring activity. First, we characterize the analyzed repositories (Section IV-A). Next we describe the environment configuration (Section IV-B). In Section IV-C we present our results and then we discuss issues related to the experiment (Section IV-D).

A. Subject Characterization

We analyzed the following open source Java projects: ArgoUML (an open source UML modeling tool), HyperSQL Database (HSQLDB – a SQL relational database engine), jEdit (a text editor with support to more than 130 languages), JHotDraw (a framework for development of graphical editors), and SweetHome 3D (a software for buildings interior design with support to 2D editing with 3D visualization).

Table I characterizes the subjects. The column Subject shows the name of the each subject. The repository (either CVS or SVN) is specified in the column Repository Type. The columns Total KLOC and Versions specify the KLOC (non-blank, non-comment lines of code) sum of all versions and the number of versions, respectively. The column Buildfiles presents the number of different build configurations for each subject. As software evolves, it may modify the original build file due to changes in the project structure, new used libraries used or new versions of the language. The columns Started and Years state when the project started and how many years we considered in our analysis, respectively. The number of programmers that have been submitted at least one commit is indicated in the column Programmers. The last row of Table I shows total values for each column. For example, we analyzed 40,723 versions.

B. Environment Setup

For Subversion (SVN) repositories, a version is defined as a single commit transaction. For Concurrent Versions System (CVS) repositories (which is the case for the subject SweetHome 3D only), we defined a transaction as all files committed by the same author with the same message within the time interval of one minute, following the same approach adopted by Murphy-Hill et al. [2].

We set the time limit of SafeRefactor to 90 seconds. Since most subjects do not have a build file available, we created them. In some of them, we create more than one build file due to the evolution of the software. We performed analysis using ten Pentium Core 2 Duo computers with 1GB RAM each, running Ubuntu 9.04 and Java 6.

C. Results

We evaluate transformations with respect to the refactoring frequency, its granularity, and scope.

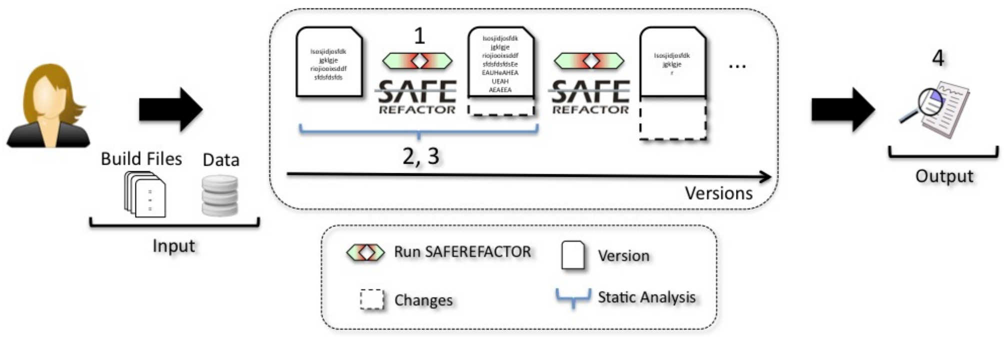
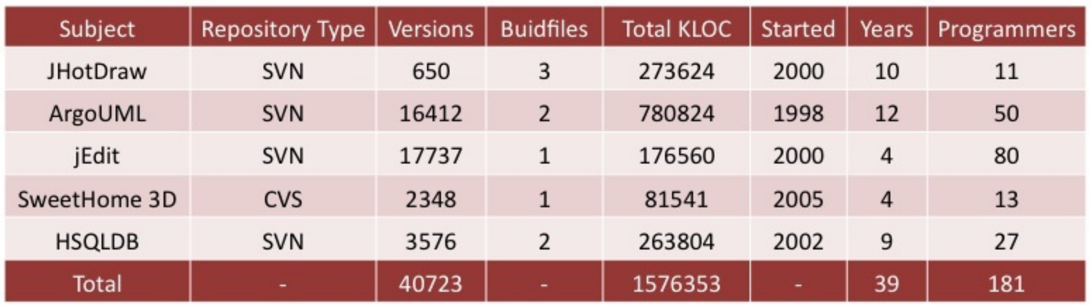


Fig. 3. A technique for analyzing refactoring on software repositories. 1. The tool runs SAFEREFACTOR over all consecutive pairs of versions and identifies the refactorings; 2. It analyzes refactorings with respect to granularity; 3. It analyzes refactorings with respect to scope; 4. The tool reports the analysis results.

TABLE I
SUBJECT CHARACTERIZATION; REPOSITORY TYPE = SVN OUR CVS; VERSIONS = NUMBER OF REPOSITORY VERSIONS; KLOC = SUM OF THE KLOC (NON-BLANK, NON-COMMENT LINES OF CODE) OF ALL VERSIONS; BUIDFILES = NUMBER OF DIFFERENT BUILD CONFIGURATIONS; STARTED = THE YEAR THAT THE REPOSITORY STARTED; YEARS = NUMBER OF YEARS CONSIDERED IN OUR ANALYSIS; NUMBER OF PROGRAMMERS THAT SUBMITTED FILES.



Subject	Repository Type	Versions	Buidfiles	Total KLOC	Started	Years	Programmers
JHotDraw	SVN	650	3	273624	2000	10	11
ArgoUML	SVN	16412	2	780824	1998	12	50
jEdit	SVN	17737	1	176560	2000	4	80
SweetHome 3D	CVS	2348	1	81541	2005	4	13
HSQLDB	SVN	3576	2	263804	2002	9	27
Total	-	40723	-	1576353	-	39	181

Refactoring Frequency

We analyzed almost 41,000 distinct versions. Table II summarizes the gathered data: subject's name, refactoring incidence (related to the total number of versions), proportion of Low level and Local refactorings, respectively, and test coverage rates in SAFEREFACTOR. The last row of Table II shows the average of refactorings for the five projects. This value, as well as the total of Low and Local refactorings, was obtained considering all versions evaluated.

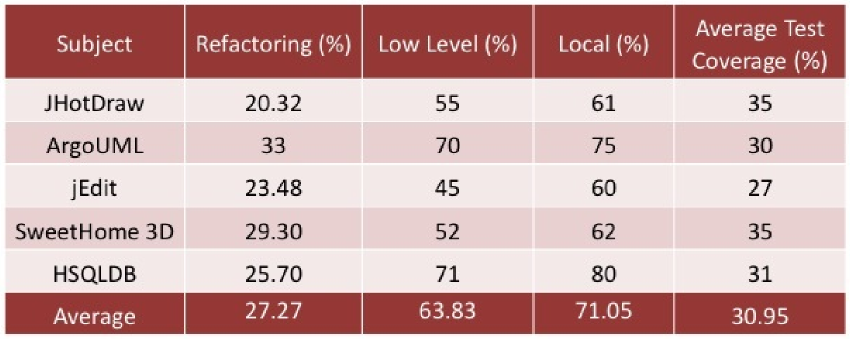
We identified that 72.73% of the transformations change behavior. For each of them we have at least one unit test detecting the behavior change. The rest presented no behavioral change, thus being classified as refactorings. The column Refactoring shows the refactoring rate for each subject evaluated.

While JHotDraw project shows the smaller incidence of refactorings (20.32%), ArgoUML has the highest (33%). These teams present distinct characteristics: while JHotDraw is a project maintained mostly by only one developer who accomplished 66% of the repository commits, ArgoUML is a project kept up by a professional team of 50 developers. We believe that a developer working in a more balanced team submits versions more frequently to synchronize with rest of the team. In constrast, the JHotDraw developer may submit more significant changes at a time, which may turn difficult the refactoring analysis in pairs of versions, since the transformation may include, besides refactorings, addition of functionalities or bug fixing.

In fact, open source development provides an interesting context for studying software evolution. As the environment is often free of schedule-related demands, teams with a balanced workload are likely to have members doing commits that only refactor the code, not adding new functionality. Usually, developers in those projects are self-managed in their tasks, which favors a clearer separation of types of commits. However, from the results we can speculate that the low rate of transformations that are refactorings probably illustrate how difficult it is to apply refactorings correctly. Many of the non-refactoring commits may have been intended refactorings, but did allow behavioral change.

Furthermore, we analyzed how often refactorings are applied within the analyzed projects. The longest sequence of consecutive refactorings found by SAFEREFACTOR was six in the ArgoUML and HSQLDB projects. In JHotDraw and SweetHome 3D the longest sequence was four. We found out that, in average, for the studied subjects, one refactoring occurs approximately every four days, after on average 3.65 non-

TABLE II
THE AVERAGE OF REFACTORING FREQUENCY, GRANULARITY, AND SCOPE, AND TEST COVERAGE FOR EACH SUBJECT, AND CONSIDERING ALL ANALYZED VERSIONS.



Subject	Refactoring (%)	Low Level (%)	Local (%)	Average Test Coverage (%)
JHotDraw	20.32	55	61	35
ArgoUML	33	70	75	30
jEdit	23.48	45	60	27
SweetHome 3D	29.30	52	62	35
HSQLDB	25.70	71	80	31
Average	27.27	63.83	71.05	30.95

refactoring transformations. There is, however, a considerable variance: the minimal interval found between two refactorings was 1 second (probably performed by different developers), while the maximum interval was 611 days (time when the JHotDraw project did not have any activity). The refactoring interval (minimum, maximum and mean values) for each subject is shown in column Refactoring Interval (Table III).

Figure 4 shows the distribution of refactoring activities in average over the time, which is represented in the format *mm.yy*, illsutrated by the SweetHome 3D and HSQLBD projects. Refactoring activity occurs frequently over the time, presumably in parallel with other changes. This results improve confidence on previous assumption [2] that refactorings are most commonly applied for a specific reason, such as adding a feature or fixing a bug, than in exclusively refactoring sessions to evolve the software design.

Refactorings become more popular due to agile methodologies that stand on refactoring practices, such as XP and Scrum. This is apparent in some subjects from our study. For instance, SweetHome3D is developed using the XP process [13] as confirmed by one of the developers. In our analysis, it has the second highest rate of refactoring (29.3%) (see Table II).

According to the standard for software engineering and maintenance [14], software maintenance can be divided in four categories: corrective, adaptive, perfective and preventive. The first one is related to identifying and fixing bugs. The next one is a change to add new features on the software. Perfective maintenance is a modification to improve maintainability or performance. We can view refactoring as a kind of perfective maintenance. Finally, preventive maintenance looks for latent faults before they become effective faults. Lientz and Swanson [15] state that 65% of software maintenance are adaptive. Our results endorse this statement, since 72,2% of the transformations changed behavior. We can consider them as adaptive or corrective maintenance.

Refactorings Granularity

The average proportion of Low level refactorings was 68.83% in the five repositories. The column Low level in Table II shows the average for each subject. While jEdit was the project with less Low level refactorings (45%), HSQLDB was the one with more refactorings of this type (71%).

Although High level refactorings form the majority of well-known refactoring catalogs [1], Low level refactorings tend to be more frequent, as show in our results. We believe that High level refactorings, by presenting more preconditions to preserve behavior, are harder to apply correctly, as their related changes may propagate over the classes. For instance, when a method is renamed, the whole class hierarchy must be checked for unintended overriding and overloading after the refactoring, to avoid behavioral changes, as show in [6], [8].

Currently, there is no formal theory that identifies and proves all refactoring preconditions considering the complete Java language. Therefore, the quality of IDEs to perform these changes is still low, and many of these transformations introduce compilation errors and other bugs. This forces developers to manually perform these refactorings [16]. Moreover, many of the High level refactorings performed by IDEs may change program behavior [6], [8], [11], [7].

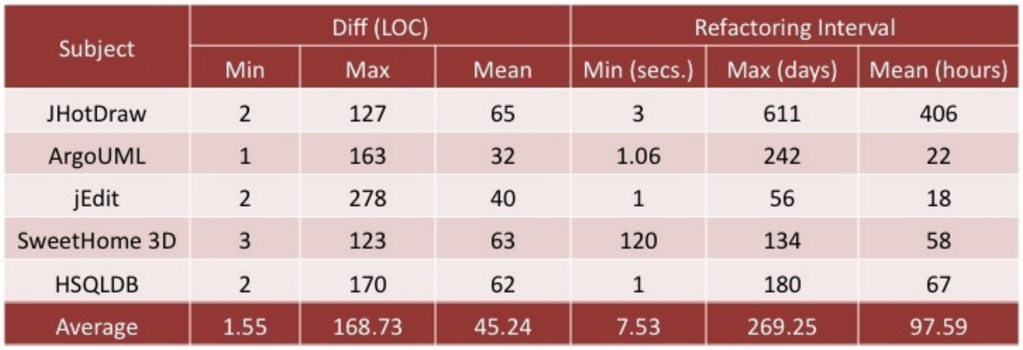
We also analyzed the variance on the size of the refactorings. The smallest refactoring size we found was 1 line of code (maybe just simplifying an expression), while the maximum refactoring size changed 278 lines. The average was 45 lines, leading to the recognition that most refactorings involve a considerable number of changes. Table III shows this information for each subject.

Refactoring Scope

Most refactorings were categorized as Local (71.05%) with respect to scope. This result was expected, since low level were the most common, and they affect only one package. Additionally, High level refactorings may also be Local. For instance, a rename of a method with package visibility will only affect invocations in this package. This may be an explanation for the rate of Local refactorings being higher than the rate of Low level refactorings.

Local refactorings tend to be easier to perform than Global ones, since they may have simpler preconditions to check. For instance, previous works identified corner cases involving packages that reveal bugs in IDEs [6], [8].

TABLE III
REFACTORING SIZE AND PERIODICITY. THE COLUMN DIFF(LOC) SHOWS MINIMUM, MAXIMUM AND MEAN VALUES ABOUT THE PORTION OF SOURCE CODE AFFECTED BY ACTIVITY IN THE ANALYZED REPOSITORIES. SIMILARLY, THE COLUMN REFACTORING INTERVAL SHOWS MINIMUM, MAXIMUM AND MEAN PERIODICITY OF REFACTORING. THE LAST ROW SHOWS THE AVERAGE OF ALL THESE VALUES.



Subject	Diff (LOC)			Refactoring Interval		
	Min	Max	Mean	Min (secs.)	Max (days)	Mean (hours)
JHotDraw	2	127	65	3	611	406
ArgoUML	1	163	32	1.06	242	22
jEdit	2	278	40	1	56	18
SweetHome 3D	3	123	63	120	134	58
HSQLDB	2	170	62	1	180	67
Average	1.55	168.73	45.24	7.53	269.25	97.59

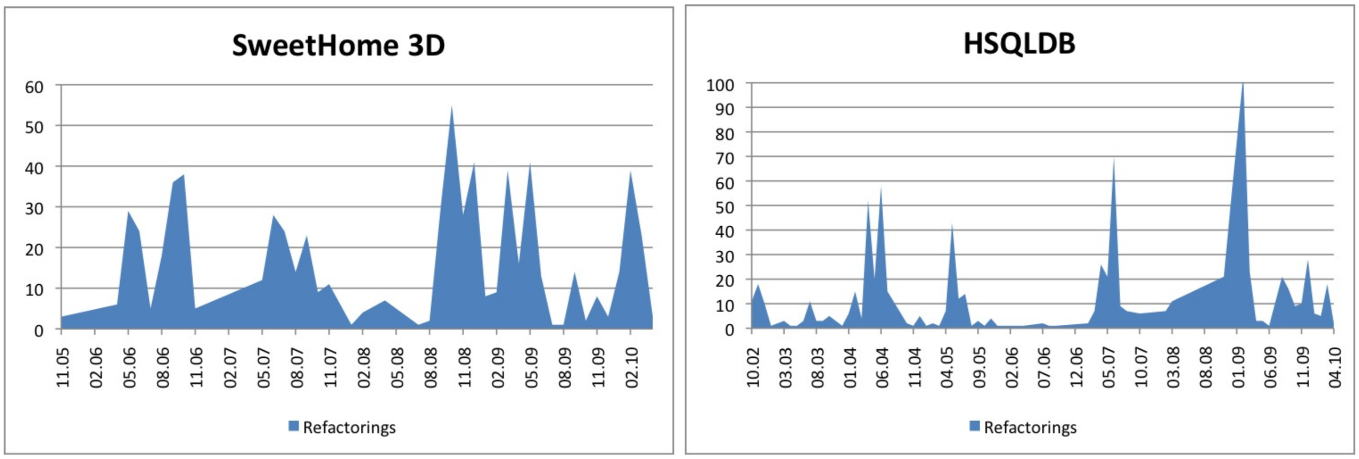


Fig. 4. Refactoring activity in repositories of SweetHome 3D and HSQLDB.

D. Threats do Validity

1) Construct validity: We do not evaluate programmers intention in refactoring, but whether a transformation preserves behavior. Therefore, the identified set of non-refactorings may include intended refactorings that were incorrect applied due to manual errors or bugs in refactoring tools [6]. Moreover, this set may also include transformations containing refactorings committed together with other behavioral changes, such as bug fixing. We believe that our technique can show more accurate results in projects with a high rate of version submissions, since the probability of changes being committed in isolation is higher. On the other hand, the fact that SAFEREFACTOR does not find behavioral changes in a transformation do not prove that it preserves behavior. However, we can have more confidence by analyzing test coverage.

2) Internal validity: For each subject, it took us two weeks to analyze the full repository. We built a system that downloads all data from repository and analyzes each consecutive pair of versions with respect to refactoring. We split the processes in parallel using 10 computers, which reduced the analysis time. We used computers with the same configuration, in order to avoid measuring bias.

Moreover, the time limit used in SAFEREFACTOR for generation tests may have influences in the detection of non refactorings. To determined its time limit in our experiment, we compared the test coverage achieve by it using different time limits. We analyzed SAFEREFACTOR in 10 consecutive versions of JHotDraw. We use the *Cobertura*[2] tool to evaluate the test suite coverage. Figure 5 shows in average the evolution of the number of generated tests and test coverage for different time limits values. The number of unit tests grows as this time is increased. However, from a given point in time, the test coverage does not grow proportionally to the number of tests; it becomes stable. The average results of these executions are that with a time limit of 90 seconds, the test coverage was 34% and the number of generated tests was 6631, while with a time limit of 240 seconds the number of generated tests were 15792 and the test coverage was 37%.

Achieving 100% test coverage in real applications is often an unreachable goal. There are evidences that this value is more likely to be 28 to 34 percent [17]. We thus chose

[2]Cobertura is available at http://cobertura.sourceforge.net/

the time limit of 90 seconds. To improve the confidence in the SAFEREFACTOR's tests, we plan to check the test coverage regarding the entities impacted by the transformation as proposed by Wloka et al. [17].

Moreover, since SAFEREFACTOR randomly generates the tests, we can have different results each time we run it. To improve the confidence, we run SAFEREFACTOR three times to analyze each transformation. If SAFEREFACTOR does not find a behavioral change in all runs, we consider that the transformation is a refactoring. Otherwise, it is classified as a non-behavior transformation.

With respect to the analysis of granularity, we used the Unix `diff` tool with no parameters, therefore, the analysis considers comments and blank lines. Therefore, the number of non comment and non blank changed lines may be lower than what we measured.

3) External validity: SAFEREFACTOR does not take into account characteristics of some specific domains. For instance, currently, it does not detect difference in the standard output (*System.out.println*) message, neither can be used with concurrent programs. Four of our subjects have graphical interfaces. By manual inspection, we observed that our technique detected some behavior changes related to graphical user interfaces (GUI). It has detected, for instance, behavioral changes related to the GUI evolution, where methods that returned the color used in some windows had changed its behavior.

V. RELATED WORK

A. Refactoring practice

Murphy-Hill et al. [2] evaluated nine hypotheses about refactoring activities. They used data automatically retrieved from users through Mylyn Monitor and Eclipse Usage Collector. This data allowed Murphy-Hill et al. to identify the frequency of each automated refactoring. The most frequently applied are: Rename, Extract Local Variable, Move, Extract Method, and Change Method Signature. They confirmed assumptions such as that refactorings are frequent. Data gathered from Mylyn showed that 41% of the programming sessions contain refactorings. This data only contains refactorings performed using these tools, therefore they do not consider refactorings manually performed.

In contrast, while they evaluated the intention of applying refactorings, we evaluated the occurrence of refactorings by checking whether two consecutive repository versions were behavior-preserving transformation. We can evaluate a transformation no matter it was manually or automatically applied. However, as mentioned in Section IV-D, we may miss refactorings that are committed with other behavioral changes and refactorings incorrectly performed by tools or developers. In our experiments, using tests for preservation of behavior, we have found that 27.27% of the pairs of versions are refactorings and that they occur frequently during the software evolution.

They also analyzed 20 pairs of code versions from Eclipse CVS repository to identify refactoring activities. They compared each pair of versions and identified the performed refactorings. By using this data through manual analysis, they studied the granularity of the changes. Besides Low and High level refactorings, they defined *Medium level* for refactorings that changes the element signature and its block (we consider this as High level). They confirmed the hypothesis that many of refactorings are Medium and Low level. However, while they identified 18-33% of refactorings as Low level, this number is much higher in our experiment (63.83%). Moreover, they stated that only 3% of consecutive pairs of versions are purely refactorings. In our work, we identified 27.27% of the transformations as behavior-preserving.

B. Detecting refactorings

Ratzinger and Gall [4] analyzed the relation of refactoring and software defects. They proposed an approach to automatically identify refactorings based on commit messages. It searches for words like "refactor", "rename", and exclude strings as "needs refactoring". As a result, they found a low number of false positive refactorings. Using evolution algorithms they confirmed hypothesis such as that the number of software defects in the period decreases if the number of refactorings increases as overall change type. However, commit messages may be ambiguous, dependent on the language of the messages (in this case English), software conventions employed and commitment of the developers to stick with these conventions. Thus, analysis about them may generate a number of false positives and false negatives. An evaluation of its correctness using static analysis is not enough since it does not check the program with respect to full semantics. We inspected a number of commit messages of some subjects and noticed that they do not predict refactorings. This result is also confirmed by Murphy-Hill et al. [2].

Dig et al. [5] created an automatic refactoring detection tool, called RefactoringCrawler, targeted on software components, addressing the maintenance of software that uses components when these components evolve. His technique generates logs of the detected changes. Any software that uses the component should use these logs to replicate the changes occurred with 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 over the possible refactorings identified on the first phase. It finds seven kinds of refactorings. On its evaluation, RefactoringCrawler was executed over two different versions of four open source programs. Compared to a previous manual analysis [18], RefactoringCrawler succeeded in finding 90% of the refactorings. Our technique uses SAFEREFACTOR to identify refactorings. It analyzes any kind of transformation and generates tests for identifying behavioral changes. SAFEREFACTOR has been useful for identifying behavioral changes in real case studies [6]. In our approach, we perform a stronger semantic evaluation than their work.

Murphy-Hill et al. [19] proposed several hypotheses related to refactoring activity, and outline experiments for testing those hypotheses. They categorized four approaches to analyze

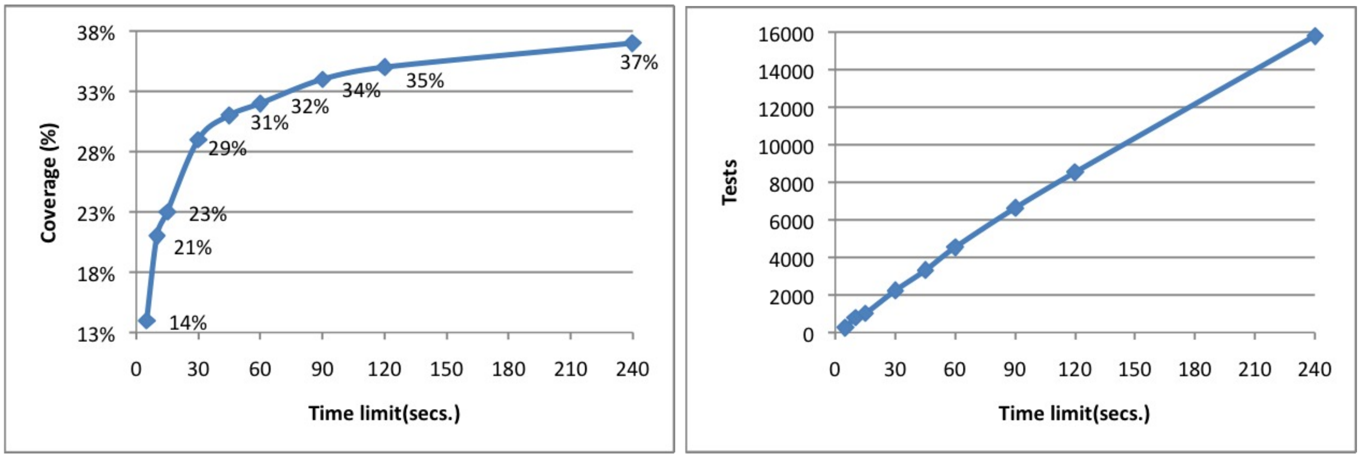


Fig. 5. Analysis of generated test suite with respect to test coverage using different time limits.

refactoring activity: analysis of 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. The their method has the advantage of identifying each specific refactoring performed. However, while the manual analysis is error-prone and does not scale, conclusions based on information from refactoring tools may not hold for refactorings manually applied. On the other hand, our technique scales well allowing full repository analysis but we do not identify which refactorings were applied (only its scope and granularity).

C. Refactoring implementation

Our findings in some way confirm an important conclusion from previous work on the subject: defining and implementing refactoring preconditions is non-trivial. While Opdyke [10] coined the term "refactoring" and specify conditions that supposedly guarantee behavior preservation, other research results formalized and showed conditions for refactorings, but only for a small subject of Java [20], [21]. Other approaches try to improve previous definitions with additional constraints, but still gaps are admitted [8], [22].

The low number of confirmed refactorings (less than 28% in average) indirectly shows that many transformations may have had the intention of refactoring the code, but turned out to change behavior unintentionally. We believe that SAFEREFACTOR can also be used to evaluate either manually-applied or tool-supported application of refactoring [6].

VI. CONCLUSIONS

In this paper, we presented a technique to automatically perform refactoring analysis with respect to frequency, granularity, and scope on Java source code repositories. Each consecutive pair of versions is classified as refactorings or non-refactorings. To identify refactorings, it uses SAFEREFACTOR [6]. We also perform a *static analysis* to classify the refactoring scope and its granularity (Section III). We used our technique to evaluate five Java open source repositories. We observed that refactorings occur frequently over all software life cycle. In 40,723 analyzed software versions, 27.27% of transformations preserved the behavior. Besides, we observed that the majority of refactorings are Low level (63.83%), they occur inside method's body (not changing the method's signature). Most of the refactorings (71.05%) are Local. And the average size of a refactoring is 45 LOC. Obtained results confirm the hypothesis that wide and global transformations are harder to perform while preserving the behavior, even with the help of tools.

The high refactoring frequency can be related to the increase of the number of automated refactorings in IDEs over the past 10 years. For instance, in the first Eclipse IDE release (2001), three refactorings were included: rename, move, and extract method [23]. Eight years later, Eclipse 3.5 automated 28 refactorings. It is important to develop better refactoring tools with respect to reliability and usability. Moreover, it is important to propose more powerful refactoring tools, as the one proposed by Knisel and Koch [24], which allows users to compose refactorings.

In our experiments, some previous assumptions were rejected. According to experiments [2], Low level refactorings correspond only to 18% of all refactorings. Moreover, in the same work, they state that only 3% of consecutive pairs of versions are refactorings, different from our work. They draw these conclusions from some versions of one subject. We evaluated all versions of five repositories.

As future work we intend to identify the refactoring applied by developers. Using this data, we can recommend new refactoring to be implemented in Eclipse. Moreover, we will evaluate whether each detected behavior-preserving change improves program's quality. We aim at using a number of metrics for classifying each transformation. By correlating our results with software quality metrics, we can draw additional conclusions regarding software evolution. We also intend to evaluate more open source repositories.

Acknowledgment

We gratefully thank the anonymous referees for 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, 477336/2009-4, and 304470/2010-4.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[2] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," in *International Conference on Software Engineering*, 2009, pp. 287–297.

[3] E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," *IEEE Software*, vol. 25, no. 5, pp. 38–44, 2008.

[4] J. Ratzinger, T. Sigmund, and H. C. Gall, "On the relation of refactorings and software defect prediction," in *Mining Software Repositories*, 2008, pp. 35–38.

[5] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *European Conference on Object-Oriented Programming*, 2006, pp. 404–428.

[6] G. Soares, R. Gheyi, D. Serey, and T. Massoni, "Making program refactoring safer," *IEEE Software*, vol. 27, pp. 52–57, 2010.

[7] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio, "Algebraic Reasoning for Object-Oriented Programming," *Science of Computer Programming*, vol. 52, pp. 53–100, 2004.

[8] F. Steimann and A. Thies, "From public to private to absent: Refactoring java programs under constrained accessibility," in *European Conference on Object-Oriented Programming*, 2009, pp. 419–443.

[9] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *International Conference on Software Engineering*, 2007, pp. 75–84.

[10] W. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[11] M. Schäfer, T. Ekman, and O. de Moor, "Sound and extensible renaming for java," in *Object-Oriented Programming, Systems, Languages & Applications*, 2008, pp. 277–294.

[12] G. Soares, M. Mongiovi, and R. Gheyi, "Identifying too strong conditions in refactoring implementations," in *International Conference on Software Maintenance*, 2011, to appear.

[13] E. Puybaret, *Les Cahiers du Programmeur Swing*. Editions Eyrolles, 2006.

[14] ISO/IEC 14764:1999, "Software engineering - software maintenance," 1999, iSO and IEC.

[15] B. P. Lientz and E. B. Swanson, *Software Maintenance Management*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1980.

[16] M. P. Technische and M. Pizka, "Straightening spaghetti-code with refactoring?" in *Software Engineering Research and Practice*, 2004, pp. 846–852.

[17] J. Wloka, E. W. Host, and B. G. Ryder, "Tool support for change-centric test development," *IEEE Software*, pp. 66–71, 2010.

[18] D. Dig and R. Johnson, "The role of refactorings in api evolution," in *ICSM*, 2005, pp. 389–398.

[19] E. Murphy-Hill, A. P. Black, D. Dig, and C. Parnin, "Gathering refactoring data: a comparison of four methods," in *Workshop on Refactoring Tools*, 2008, pp. 1–5.

[20] P. Borba, A. Sampaio, and M. Cornélio, "A refinement algebra for object-oriented programming," in *European Conference on Object-Oriented Programming*, 2003, pp. 457–482.

[21] L. Silva, A. Sampaio, and Z. Liu, "Laws of object-orientation with reference semantics," in *Software Engineering and Formal Methods*, 2008, pp. 217–226.

[22] M. Schäfer and O. de Moor, "Specifying and implementing refactorings," in *OOPSLA '10: Proceedings of the 25th ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, 2010.

[23] R. Fuhrer, A. Kiezun, and M. Keller, "Refactoring in the eclipse jdt: Past, present, and future," in *Workshop on Refactoring Tools at ECOOP*, 2007.

[24] G. Kniesel and H. Koch, "Static composition of refactorings," *Science of Computer Programming*, vol. 52, no. 1-3, pp. 9–51, 2004.