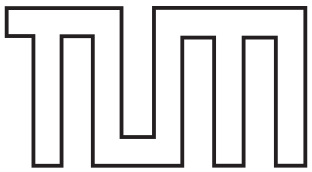


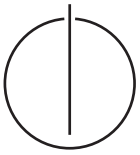
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Detection of Refactorings

Florian Dreier



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Detection of Refactorings

Erkennung von Refactorings

Author:	Florian Dreier
Supervisor:	Prof. Dr. Dr. h.c. Manfred Broy
Advisor:	Dr. Elmar Juergens (TUM) &
	Dr. Andreas Göb (CQSE)
Submission Date:	July 15, 2015

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, July 15, 2015 Florian Dreier

Acknowledgments

First of all, I would like to say thank you to Dr. Elmar Juergens who helped me to find a topic that really sparked my interest from the moment I first read about it. I also want to thank him for his support both on getting started with the topic and also for his feedback while writing this thesis. In addition, I would like to say a sincere thank you to Dr. Andreas Göb for his constant support and dedication during the implementation and writing of the actual thesis. Furthermore, a big thank you to Prof. Broy for making such an amazing company collaboration with CQSE possible.

Finally, a big thank you to Markus Christians, Cara Christians, Kordian Bruck, Maximilian Fuß, Lavinia Eifler, Xenia Eifler and Verena Dreier who carried out the most tedious work of correcting the language of this thesis and of proof-reading.

Abstract

To avoid bugs in software products a lot of effort is put into testing. Especially code which has been modified since the last release, is likely to induce new bugs and hence should be tested carefully. But the set of methods that have changed grows rapidly if structural refactorings are involved because they produce a lot of non-essential changes. That means that the changes do not have an effect on the program's behavior. To filter these out from the set of modified methods, a refactoring detection can be used to identify methods that only contain non-essential changes. If the refactoring detection classifies a method as unchanged by mistake, even if it has been modified, the method would be filtered out. That leads to the problem that this method would not be tested, which in turn can lead to bugs in the software.

This thesis describes a refactoring detection technique which is able to classify methods as changed or unchanged, whereby methods classified as unchanged are guaranteed to contain only non-essential changes. The thesis also describes approaches presented in other papers which did not meet the special requirements for the described use case, but influenced the approach of this thesis.

The presented approach is signature-based. At first two version of the project are loaded, which should be compared. Each method from the predecessor version gets matched to one method of the successor version, which is very similar and therefore likely to have emerged from the predecessor version. Renaming of methods and classes is inferred by this matching. The classification is done by applying the renaming to each predecessor method body and comparing the result with the successor method body. If they are equal, the method is classified as unchanged otherwise as changed.

An evaluation of the technique was done in three case studies. The first one uses a mutation-based benchmark to test the correctness of the implementation for a set of refactoring types. The second case study analyzes how many refactorings are done in real world projects. The third one makes a comparison to another bytecode-based refactoring detection tool and attests the presented approach to have a precision of 100% and a recall of 99%.

Contents

1 Introduction

When inspecting a new software version, developers need to take a look at the methods that have been changed since the last version. A diff of the relevant program versions is often used to accomplish this task. Especially if structural refactorings are involved, the set of changes can be very large. For example renaming the root package of a Java project causes changes in nearly all files contained in the project. Since these so called *non-essential changes* do not change the program's functionality, a developer is often not interested in those, but in code parts implementing new features.

The following example shows two versions of a method exhibiting a number of textual differences which are different kinds of non-essential changes:

Listing 1.1: Version N

```
1 import java.util.List;
2 ...
3 List list = ...;
4 void init(int pos, String item) {
5       this.list.add(pos, item);
6 }
```

Listing 1.2: Version N+1

```
1 java.util.List mList = ...;
2 void add(String name, int pos) {
3       mList.add(pos, name);
4       return;
5 }
```

In the third line of version N, the local variable `list` is renamed to `mList` and its simple type replaced with its equivalent fully-qualified type. The method itself is renamed from `init` to `add`. The method's parameter `item` is renamed to `name`. The parameters switched their positions. In the fifth line, a trivial instance of the `this` keyword is removed. The fifth line is also textually altered by the effects of the `list` attribute's rename refactoring. Finally, in the sixth line, a redundant `return` keyword is inserted. These snippets of code have the same behavior, but textually they do not have much in common.

A *refactoring detection* analysis can be used to detect source code that was changed as a result of refactoring. Hence, non-essential changes can be filtered out and help the developer focus on the essential changes. This is especially helpful when doing reviews.

Another application for refactoring detection is a code ownership analysis which tries to find out which developers have made changes to a special piece of source code. In this use case it is nevertheless a good idea to leave out developers, who only changed the code during automated refactorings and therefore cannot be considered as owner of the code.

Since in long living software projects a huge amount of code does not change very frequently [8], most of the bugs are induced by changes that have been made recently to the code. That is why test engineers put a lot of effort into testing these changes. A study [1] has shown that still a large amount of modified code ends up untested in production and causes a majority of field errors. Because testing can be very expensive, it is worthwhile to test only those code fragments that contain essential changes and therefore may induce new bugs.

Requirements This thesis presents a refactoring detection technique which especially adapts to these needs. The special requirements which the refactoring detection has to fulfill, are listed below:

R1: The approach has to operate on *source code* to provide a common basis for other languages and to make up the drawbacks of bytecode-based analysis as discussed in subsection 2.2.2.

R2: It must be possible to transfer the implementation to *other languages*.

R3: It has to be capable of recognizing the *most frequently used refactorings*, especially those that introduce a lot of non-essential changes in source code and thus produce a lot of non-essential textual differences.

R4: It may only make a *one-sided error* meaning that methods containing essential changes must not be classified as unchanged, but changes that are non-essential may be classified as modified. Otherwise, the analysis could filter out essential changes by mistake.

R5: Since it is planned to include the analysis in the software Teamscale (see subsection 2.2.3), a software tool that allows to track software quality goals in real time, *reasonable running time* is an essential requirement as well.

The remainder of this thesis is structured as follows: Chapter 2 describes some common approaches in research and presents the existing ConQAT framework used for the implementation of the approach. In chapter 3 the refactoring detection itself is presented. Chapter 4 investigates three case studies which evaluate the implementation. In chapter 5 this thesis and its evaluation results are summarized and chapter 6 proposes some ideas for future work.

2 Related Work

2.1 Research

First researches on refactoring detection started more than a decade ago [6]. Subsequently some papers are presented which emerged from this research. The papers use quite different techniques in carrying out refactoring detection. Unfortunately, the following techniques do not provide a solution that can be used in the presented use case for various reasons clarified below. Nevertheless, it was possible to reuse some of the presented ideas.

2.1.1 Identifying Refactorings from Source Code Changes

Peter Weißgerber and Stephan Diehl [9] use a signature-based analysis meaning that program elements are represented as tuples like $\langle$*classname*, *methodname*, *parameters*, *returntype*, *visiblity*$\rangle$ for a method (also called method signature). To detect refactorings, the signatures of two possibly matching methods are compared and renamings are inferred. For example a method rename is inferred if the following rule holds for the given tuples, where M is the set of methods in the first version and M' contains all methods of the second version:

$$\langle c', m', p', r', * \rangle \in M' \Rightarrow \nexists \langle c', m', p', r', * \rangle \in M \wedge \exists \langle c', m, p', r', * \rangle \in M \wedge \ m \neq m'$$

Their approach preprocesses the given versions of a project to identify classes, methods and fields. Then it generates the according class, field and method signatures. Applying the above and similar rules, the approach is able to find possible renamings of classes and methods, as well as addition and deletion of parameters, changes to method's visibility and movements of classes between packages. These possible refactorings are referred to as *refactoring candidates*. The clone detection tool CCFinder[1] is then utilized to compare the method bodies that belong together. CCFinder already takes into account that consistently renamed variables, changed method's visibility and some other refactorings, are non-essential changes. These refactoring candidates having been detected as clones by the clone detector, are rated as very likely to contain only non-essential changes.

[1] `http://www.ccfinder.net/`

What makes the approach valuable for the thesis' problem statement is that it is able to operate on method header level and hence is able to classify the changes made to a specific method. Therefore, the signature-based idea was reused. But on the other side rating refactoring candidates does not guarantee the found results to really being refactorings. This violates one of our requirements, namely R4.

2.1.2 Non-essential Changes in Version Histories

David Kawrykow and Martin P. Robillard [4] developed a technique based on abstract syntax trees (AST). Since their implementation is based on the tool SemDiff[2], a change analysis tool for studying framework evolution, the infrastructure already provides ASTs for two corresponding files in the version history. The first step is to run a *partial program analysis* (PPA) on the given ASTs to resolve type bindings. Then ChangeDistiller[3], a tool that identifies changes on statement-level, is used to detect renamings based on the AST pairs. After that, the renamings are applied to the ASTs and ChangeDistiller is executed again utilizing the modified AST pairs. Based on this output the program decides, whether the identified structural changes are non-essential.

The approach presented in this thesis also tries to collect all renamings and applies them to the predecessor methods' bodies, as suggested in the paper of Kawrykow and Robillard. But for this thesis' problem statement, their approach has a few downsides. First of all, the technique works on file level and not on method level required to detect methods that need to be tested. This also implicates that this approach is not able to detect method movements between classes or inner class to outer class refactorings. Finally, the tool ChangeDistiller, on which the technique heavily depends, is only available for Java meaning that it is not trivial to port this solution to other programming languages. That violates requirement R2.

[2] `http://cs.mcgill.ca/~swevo/semdiff/`
[3] `http://www.ifi.uzh.ch/seal/research/tools/changeDistiller.html`

2.1.3 Discovering and Representing Systematic Code Changes

Miryung Kim and David Notkin [5] have built a tool named LSDiff[4]. This tool works with a fact database on statement level and focuses on representing the found refactorings in a human readable form. They generate a fact base for each version of the source code by using the tool JQuery[5]. For further investigations, the difference between two consecutive versions' fact bases is computed. This allows to infer some rules, which describe the changes that have been made at a higher level. For example, this tool is able to find out coherences such as that a call to `Log.trace()` is added to all methods that call `Log.on()`.

The paper of Prete et al. [7] uses exactly the same concept and can be considered an extension to the above paper, since it incorporates more fact types allowing to reconstruct more complex refactorings.

This approach achieves good results and also does find complex refactoring patterns, but it does not fit very well to this thesis' problem statement, because it provides no direct information on what methods changed their behavior. It also produces a lot of information not needed for the testing use case and therefore causes a huge computational overhead (R5). In addition, the tool JQuery is again limited to Java.

[4] `http://www.cs.ucla.edu/~miryung/lsdiff-web/index.html`
[5] `http://jquery.cs.ubc.ca/index.htm`

2.2 Technical Fundamentals

In the following the framework, which was used to implement this thesis' approach, is presented.

2.2.1 ConQAT

The Continuous Quality Assessment Toolkit[6] (ConQAT) is a highly configurable software quality analysis engine. ConQAT is based on a pipes and filters architecture, which allows the user to set up complex software analysis configurations. The system is made up of modules which implement processors. Every processor is specifically written to do one special task. Processors can be combined with other processors to build a block. A block can accomplish more complex tasks and may contain other blocks as well. The analysis can be composed with a graphical user interface.

The thesis' implementation builds upon ConQAT. This implicates that some of the blocks and processors could be reused. The actual refactoring detection is a new ConQAT processor, which allows to combine it with other ConQAT elements.

Clone detection ConQAT contains a module that is able to do clone detection analysis on a given project. Code clones are pieces of code that are equal if renamings are not considered. To make the code pieces comparable and robust against renamings, the variable names appearing in the code are normalized as illustrated in the following example [3]:

Listing 2.1: Original

```
int a = 0;
int b = a * a;
```

Listing 2.2: Normalized identifiers

```
int var0 = 0;
int var1 = var0 * var0;
```

The approach described in this paper uses a similar technique to detect local variable renames.

Shallow parser ConQAT is open source, but the company CQSE has developed extensions which are closed source. The most commonly used extension in the implementation is called *shallow parser*, which is able to parse a source code file into an abstract model. The shallow parser interface is already implemented for a set of programming languages, namely ABAP, ADA, C++, C#, Delphi, Fortran, Java, JavaScript, Magik, MATLAB, PL/SQL, Python and Ruby. The implementation of refactoring detection currently only works for Java, but building on the shallow parser makes it easier to implement the refactoring detection for other supported languages as well.

[6] `https://www.conqat.org/`

2.2.2 Test Gap Analysis

The company CQSE also developed a technique called *Test Gap analysis*. The analysis is able to detect code fragments that have indeed been modified since the last version, but not been tested yet. Those modified methods are also referred to as *Test Gaps*.

The Test Gap analysis is based on the ConQAT framework. It includes a refactoring detection which works based on Java's bytecode. Bytecode is the translation of source code to lower level statements generated by the compiler, and later executed by the Java virtual machine. Bytecode has the advantage that all cosmetic changes to the source code have already been removed by the compiler. This includes changes in source code formatting and inserted or modified comments. Also classes, methods and attributes have already been resolved to their fully-qualified names. *Fully-qualified name* means that a class "sample" contained in the package "com.container" is referred to as "com.container.sample". The type resolution is an essential requirement to be able to rollback and hence detect renamings.

Downsides Taking bytecode as basis for refactoring detection has a few disadvantages as well. When using more advanced syntactic java constructs, like anonymous inner classes, the compiler generates bytecode that has no direct counterpart in source code. This makes it harder to translate analysis results back to source code level, which is required to tell the developer where the Test Gaps have been found. In other cases, the compiler generates code that is not even visible in source code. For example, the field `serialVersionUID` is added to each class that implements `Serializable`. In addition, if not manually specified, this field is initialized with a random number that differs from version to version without any changes made to the source code which could have been tested. Another problem is that the detection operates on bytecode, a format that does not exist for scripting languages like JavaScript and Python or languages like C, which directly compiles to executable machine code. Since machine code is very low level, it is nearly impossible to do Test Gap analysis on it.

2.2.3 Teamscale

Teamscale is a software quality analysis tool that helps to monitor software quality goals and provides real-time feedback. This system is built upon ConQAT, but makes it easier to use. Another main difference is the real-time feedback whereas ConQAT only runs in batch mode. In the long run, it is planned to integrate Test Gap analysis into Teamscale. That is why reasonable running time (R5) is a requirement for the refactoring detection to be able to use it in a real-time environment.

3 Approach

3.1 Overview

A short summary of the problem statement: The refactoring detection technique should work on source code basis (R1). It should be possible to extend the approach to other programming languages (R2). Furthermore structural refactorings causing a lot of non-essential changes should be detected (R3). It should not occur that methods are classified as unchanged by mistake even though they contain essential changes (R4). The approach should run in reasonable time (R5), but speed optimization is not addressed in the design of the approach.
The approach can logically be separated into four steps:

1. Model Construction: In the first step, all versions of the source code, which should be analyzed, are read into the memory. With that information a hierarchical structure is created providing information about all packages, classes, methods and fields contained in the respective version.

2. Method Matching: In the second step for each pair of consecutive versions, a matching algorithm is executed to determine which methods belong together. Belonging together means that the successor version of the method emerged from its predecessor version. This is achieved by comparing several method characteristics, like the method's class, its parameters or its body. All rename refactorings that have been found by this matching are stored.

3. Post-Processing: Third post-processing is done to detect renamed attributes and method signature changes. This is done in a separate step, because class renames that have happened are by now already known and thus can be taken into account. These renamings are applied to the attribute and parameter types. That allows to detect renamed attributes and parameters which have possibly changed their position in the method signature.

4. Change Detection: The last step is to apply all found rename refactorings from step 2 as well as signature changes of step 3 to the method bodies. Some other normalization is also done in this step to eliminate local variable renamings. The normalized versions of the method bodies are compared to the methods that they have been matched to. If the normalized bodies are equal, the method is classified as unchanged, otherwise it is classified as modified. If a method in the successor version has not been matched to any method in the predecessor version, the method is classified as added.

Example The following source code snippets show an example that is used to illustrate the steps that are described in the following sections. Listings on the left hand side belong to the predecessor version, listings on the right hand side belong to the successor version. The applied refactorings are the following: `ClassB` has been renamed to `Printer`. Its `print` method has been renamed to `draw` and the parameters of the method have changed their order. In `ClassA` the attribute `b` was renamed to `printer` and the parameter was renamed from `x` to `in`. The comments were also altered or removed.

Listing 3.1: ClassB.java

```
package org.sample;
public class ClassB {
	private String text;
	public ClassB(String text) {
		this.text = text;
	}

/** Some comment */
	public void print(int color,
			float x, float y) {
// [...]
	}
}
```

Listing 3.2: Printer.java

```
package org.utils;
public class Printer {
	private String text;
	public Printer(String text) {
		this.text = text;
	}

/** Some documentation for draw */
	public void draw(float x, float y,
			int color) {
// [...]
	}
}
```

Listing 3.3: ClassA.java (Predecessor)

```
import org.sample.ClassB;

public class ClassA {
	private ClassB b;
	public ClassA(String x) {
// No plan what's going on here
// TODO use speaking names
		b = new ClassB(x);
		b.print(0xFF000000, 50, 100);
	}
}
```

Listing 3.4: ClassA.java (Successor)

```
import org.utils.Printer;

public class ClassA {
	private Printer printer;
	public ClassA(String in) {
		printer = new Printer(in);
		printer.draw(50, 100, 0xFF000000);
	}
}
```

3.2 Model Construction

Since the implementation uses the existing Test Gap environment (cf. subsection 2.2.2), there was already a convenient implementation which takes the projects root folder and a file name pattern e.g. $*.java$ to query all relevant files. The different versions are expected to be in separate folders inside the root folder. Per version a list of all files is created that matches the specified file pattern.

At first package name, imported classes and declared classes are extracted from each file. This information is aggregated in an `availableClasses` list containing all fully-qualified class names available within the program, including both framework classes and classes contained in the given source code.

Utilizing the shallow parser a so called *genealogy model* is created as shown in Figure 3.1.

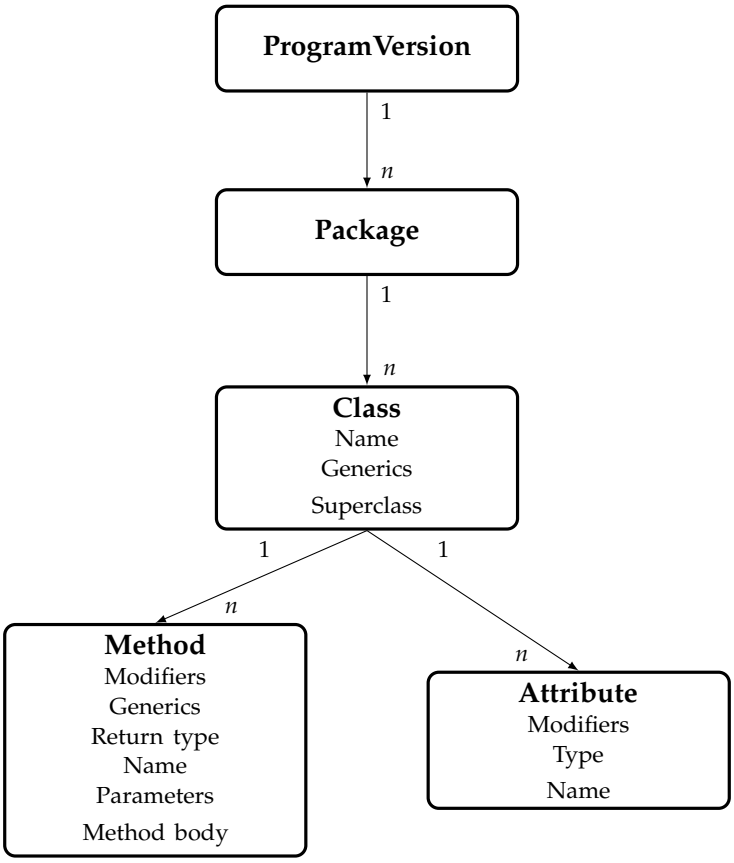


Figure 3.1: Genealogy model used to store program versions

For each program version an object is created storing references to all package names that are part of the source code. Each package object in turn contains references to each class being part of that package. A class stores, besides the references to its contained methods and attributes, information about generics used in the class' declaration and possibly its superclass name. The method stores information about used generics, its return type, parameter names and types, modifiers like `static`, `final` or `public` and a copy of the methods body. The attributes contain similar information. All classes have a common base class which is able to store a reference to their respective predecessor and successor object. The base class also takes care of storing the child elements, but is not shown in Figure 3.1.

Type resolution Since all classes, which are available within a specific file are known, types appearing in method and attribute declarations can be directly resolved to their fully-qualified class names. Besides resolving method and attribute types, all local variables are collected and get their types resolved. The declaration scope is also known to the respective local variable.

The genealogy model for the example introduced above is shown in Figure 3.2.

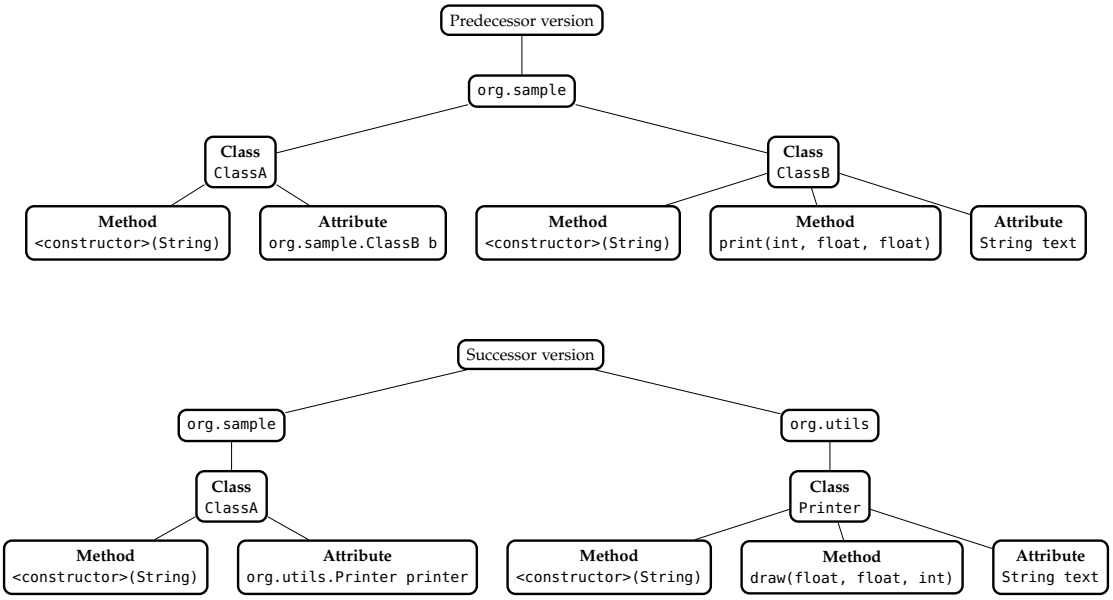


Figure 3.2: Genealogy model of the example

3.3 Method Matching

The next step is about finding a mapping which connects methods from the predecessor version to methods from the successor version. An example matching is shown in Figure 3.3.

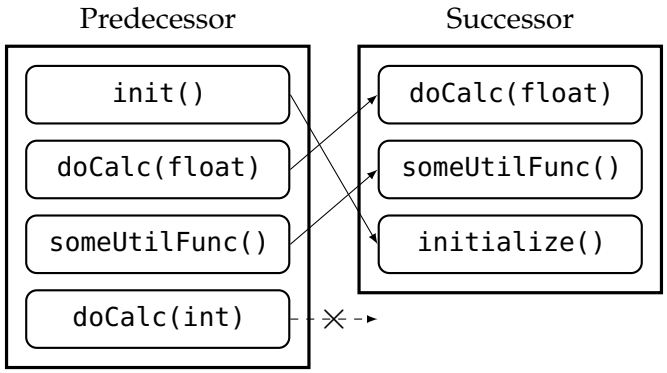


Figure 3.3: Method matching example

The matching is done with a fix-point iteration algorithm. In the first place, methods are matched by very strict criteria. All methods that have not found a matching partner are then matched by a less restrictive matching. Each matching that has been found is saved. If a renaming can be derived from the matching, the renaming is saved to a so called `RenameOracle`. Since found renamings influence the result of next matchings this is repeated until no new matchings are found. Since the number of unmatched methods decreases with every loop, infinite loops cannot occur.

The criteria which are used for the matching are the following:

1. Methods are matched together where none of *package*, *class*, *method name* and *parameter types* have been changed. This matching normally finds most of the matched methods because the majority of methods do neither change their name, signature nor parent within a version step.

2. Methods, whose *bodies* have been completely untouched, are matched. This step is executed because it is likely that methods, which have exactly equal bodies, belong together, even if their name or parent class has changed. If more than one matching partner is found with the same method body, a rating system as described in subsection 3.6.2 is used to decide which methods are matched. If multiple matching candidates are found in the following steps, the rating system is used as well.

3. This matching is very similar to step 1 except that known renamings and methods *return types* are considered as well as *package name*, *class name*, *method name* and *parameter types*. So if there is a class `MyClass`, which has been renamed to `NewClass`, and a method `doSomething(MyClass obj)`, step 1 would not be able to match the method with `doSomething(NewClass obj)` of the successor version. But after the program found out about the renamed class, it is capable of renaming all parameter types and hence can match the methods.

4. The next matching does not consider the parent class, but only *return type*, *method name* and *parameter types*.

5. Here the same matching is performed as in the step before, but types are compared only by their *simple class name*, not fully-qualified names.

6. The matching normalizes the method body in a way that all class names which appear in it, are replaced by a placeholder "Class". All other variable identifiers like attributes or local variables are replaced by a placeholder "Identifier". Finally, all method names are replaced by the placeholder "Method". All other irrelevant tokens like comments and whitespaces are removed. The normalized method body of the example `ClassA` constructor (Listing 3.5) is shown in Listing 3.6.

   Listing 3.5: `ClassA.<constructor>(String)` body

   ```
   // No plan what's going on here
   // TODO use speaking names
   b = new ClassB(x);
   b.print(0xFF000000, 50, 100);
   ```

   For reasons of better readability the all normalized source codes have been formatted. The actual normalized code neither contains line-breaks nor spaces between statements.

   Listing 3.6: Normalized method body

   ```
   Identifier = new Class(Identifier);
   Identifier.Method(0xFF000000, 50, 100);
   ```

   The normalized version only mirrors the method's structure. Methods are then matched based on this structure.

7. Lastly, a class renaming rule is added to the `RenameOracle` if all methods of one class in the predecessor version have been matched to one single class in the successor version. This means that the class has been renamed. Package renamings do not have to be handled separately because the class names are fully-qualified, which means their package name is a prefix of the class' fully-qualified name. Renaming a package therefore results in renaming all classes of the package.

   Every time a class rename is recorded, all fields of that class are mapped to the corresponding successor class as well. The algorithm works analogously to the parameter mapping described in section 3.4.

Running the algorithm on our example leads to the matching shown in Figure 3.4.

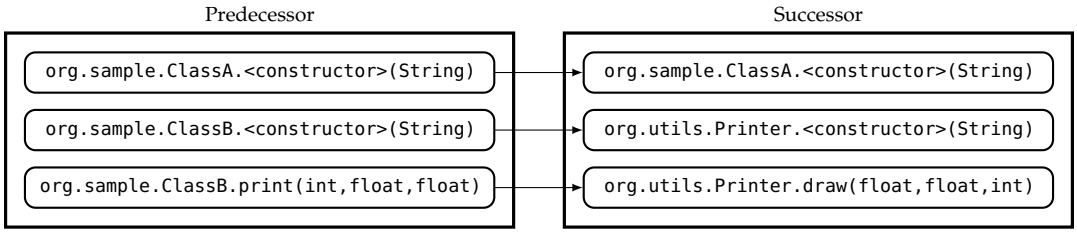


Figure 3.4: Method matching for the example

Figure 3.5 shows the corresponding type and method renames which can be inferred from the matching, and the attribute renames found by the attribute mappings in step 7.

$$\texttt{RenameOracle} = \left(\begin{array}{lcl} \text{typeRenames} & = & \{\texttt{org.sample.ClassB} \rightarrow \\ & & \texttt{org.utils.Printer}\} \\ \text{methodRenames} & = & \{\texttt{org.sample.ClassB.} \\ & & \texttt{print(int, float, float)} \rightarrow \\ & & \texttt{org.utils.Printer.} \\ & & \texttt{draw(float, float, int)} \\ \text{fieldRenames} & = & \{\texttt{org.sample.ClassA.b} \rightarrow \\ & & \texttt{org.sample.ClassA.printer}\} \end{array} \right)$$

Figure 3.5: `RenameOracle` for the example

3.4 Post-Processing

For all matched methods a parameter mapping is generated for the predecessor method parameters versus the successor method parameters, as shown in Figure 3.6. With this mapping, signature changes can be detected by the refactoring detection like reordering or renaming the parameters.

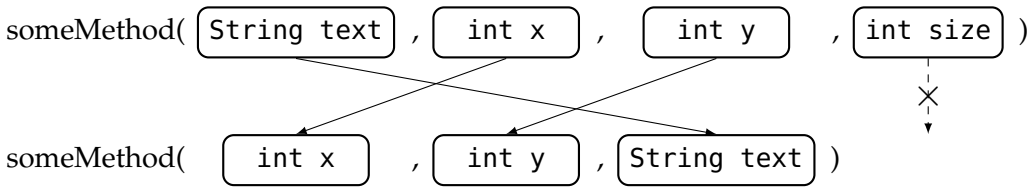


Figure 3.6: Parameter mapping

This is done in several stages:

1. Types of the predecessor version are renamed if possible to their successor type names with `RenameOracle`.
2. Parameters get matched which have equal position, type and name.
3. Parameters get matched which have not been mapped, but have an unmapped counterpart with the same type and name.
4. Unmatched parameters of the same type are matched. If there are multiple possibilities, the one with the smallest textual disparity is chosen. This step can possibly produce incorrect results, e.g. if `int size` is removed and `int index` is added, but if the parameters are semantically different, the method body has changed anyway. So the method gets classified as changed. If the mapping is right, the algorithm produces better results.

3.5 Change Detection

For all matched method body pairs a normalization is performed in order to make the methods comparable. To make the following steps of normalization more concrete, every step is applied to our example `ClassA` constructor of the predecessor version shown in Listing 3.7 as a reminder.

Listing 3.7: Original `ClassA` constructor

```
public ClassA(String x) {
// No plan what's going on here
// TODO use speaking names
       b = new ClassB(x);
       b.print(0xFF000000, 50, 100);
}
```

At first, unnecessary whitespace and comments are removed, because these have no influence on the method's behavior.

```
b=new ClassB(x);
b.print(0xFF000000,50,100);
```

Going further all type names are resolved to their fully-qualified names, so that we can be sure the class is really the same for the predecessor and the successor version.

```
b=new org.sample.ClassB(x);
b.print(0xFF000000,50,100);
```

For the types as well as methods and fields, all known renamings which have been saved to `RenameOracle` are applied to the predecessor method. To achieve this, the normalization has to keep track of variable types and method return types.

```
printer=new org.utils.Printer(x);
printer.draw(0xFF000000,50,100);
```

Detected signature changes have to be applied as well by reordering the arguments and possibly removing them. If parameters have been inserted the argument is left empty. That leads to a textual difference in the normalized predecessor and successor method body correctly resulting in a *changed* classification.

```
printer=new org.utils.Printer(x);
printer.draw(50,100,0xFF000000);
```

For local variable names and method parameters a technique is utilized that is also used in code clone detection tools (cf. subsection 2.2.1). All variable names are replaced by distinct identifiers numbered by their declaration order. This allows to normalize

both methods to the same format if only renamings happened to the variables. For the method parameters, signature changes have to be considered as well. This means that if the order of parameters has changed, the declaration order is used from the successor version for both.

```
printer=new org.utils.Printer(param0);
printer.draw(50,100,0xFF000000);
```

The same procedure is applied to the successor method body. After performing the normalization, both method bodies are equal in the presented example because only refactorings have been performed within the two compared versions.

This normalized format allows the algorithm to classify a method as *unchanged*, if the normalized method bodies are equal, because it is safe to assume that only the detected non-essential changes have been applied. All other essential changes would cause the normalized bodies to be textually different and hence lead to a *changed* classification. Methods of the successor which have not found a matching partner are classified as *added*.

3.6 Implementation Details

3.6.1 Wildcard Resolution

As described in section 3.5 the approach tries to resolve every type name to its fully-qualified name. This is necessary to create a unique identifier for each class. If the approach fails to resolve the type in one of both versions, this leads to classifying the method as changed. This can especially happen if one of the compared versions uses wildcard imports and the other one makes use of fully-qualified imports. The resolution works as expected if the wildcard imported package is part of the analyzed source code, because all classes that live within the package are known. For packages that are not available as source code within the analysis like external libraries, the approach is not able to resolve types that are imported via wildcards. In this special case only the types of the fully-qualified version can be resolved. Since switching from wildcards to fully-qualified imports is mostly done tool supported, this applies to all methods of the analyzed system and thus nearly all methods would be marked as changes.

A possibility to overcome this problem would be to include the library's source code in the analysis. But this is only viable if the source code is available, which is not the case for some third-party dependencies. So the problem is solved by creating a pool of `availableClasses`, which helps guessing what classes are imported by wildcard imports. This pool is filled with all fully-qualified class names that are imported in any file and any version of the system. The following example demonstrates this approach.

Listing 3.8: Wildcard imports

```
import javax.swing.*;
import java.awt.*;
import eu.mine.util.*;
```

Listing 3.9: Fully-qualified imports

```
import javax.swing.Icon;
import java.awt.Color;
import java.awt.Paint;
import eu.mine.util.FileUtils;
import eu.mine.util.StringUtils;
```

$$\texttt{availableClasses} = \begin{pmatrix} \texttt{javax.swing.Icon} \\ \texttt{java.awt.Color} \\ \texttt{java.awt.Paint} \\ \texttt{eu.mine.util.FileUtils} \\ \texttt{eu.mine.util.StringUtils} \end{pmatrix}$$

Now the program is able to guess what classes were contained in which wildcard imported package. For example:

```
import java.awt.* ⇒ import java.awt.Color
                    import java.awt.Paint
```

This works because if both versions use wildcards, none of the classes can be resolved and both types are compared by their unresolved name. If both use fully-qualified references there is no problem at all and if only one of them uses wildcards the above procedure is able to recover almost all type bindings. Only if the wildcard version uses classes which are no longer used in the fully-qualified version, it is not possible to get the fully-qualified name. But in this case the affected methods have clearly changed their behavior, which means the whole normalization, and as a consequence the fully-qualified name, is irrelevant.

3.6.2 Resolution of Non-unique Matchings

As stated in section 3.3 if a matching is non-unique and a method in the predecessor version has between two and four possible matching partners in the successor version, a rating system is used to decide which of the possible matching candidates is chosen.

For more than four possible matching partners the likelihood for wrong matchings would increase. But since a fix-point algorithm is used, it is likely that some of the candidates get matched in some of the other steps which are more restrictive and hence have less matching partners. In the next round the number of matching partners could have decreased so that the rating can be applied.

The rating is achieved by calculating a score value respectively for each possible method pair. The scoring points are based on the textual similarity of the method's package name, class name and method name and other method characteristics listed in Table 3.1. Every respective method characteristic yields a score in a given range and the sum of all rated characteristics results in the final score.

Table 3.1: Maximum scoring points per characteristic

Characteristic	Maximum score
Package name	20
Class name	30
Method name	30
Return type	30
Parameter list	40
Method body	60

Name rating The package name and class name rating scores are calculated with a Levenshtein edit-distance (=$distance = lev(s1, s2)$), where $s1$ and $s2$ are pairs of package or class names. The corresponding *score* function is defined as follows:

$$score(s1, s2, maxScore) := \begin{cases} maxScore & \text{if } distance = 0 \\ \left(1 - \frac{distance}{max(s1.length,\ s2.length)}\right) \cdot \frac{maxScore}{2} & \text{otherwise} \end{cases}$$

This ensures that only equal texts get the maximum score and each modification results in deduction of points. Edited texts get only half of the maximum score, since it is unlikely that package name and class name both only changed in some characters. The example in Figure 3.7 illustrates the effects of that decision. Method $m1$ is part of the predecessor version and $m1'$ and $m2'$ are possible matching candidates. We assume that the methods have equal normalized method bodies and are hence considered as matching candidates. A method is defined as $method = \langle package, class, name, parameters \rangle$.

$$m1 = \langle \texttt{com.my.prog.utils}, \texttt{BmpUtils}, \texttt{getBmpFile}, \langle \texttt{File} \rangle \rangle$$

$$\Downarrow$$

$$\left\{ \begin{array}{l} m1' = \langle \texttt{com.my.prog.utils}, \texttt{BmpUtils}, \texttt{getBitmapResourceFile}, \langle \texttt{File} \rangle \rangle \\ m2' = \langle \texttt{com.my.prog.utils}, \texttt{BinUtils}, \texttt{getBinFile}, \langle \texttt{File} \rangle \rangle \end{array} \right\}$$

Figure 3.7: Multiple matching candidates

Method $m1'$ could be the successor if the method's name has been changed from `getBmpFile` to `getBitmapResourceFile`. Method $m2'$ would require a class renaming to `BinUtils` and a method renaming to `getBinFile`, which is even in sum a lower edit distance. But since it is more common that only the method name has changed, $m1$ should be associated with $m1'$.

Table 3.2: Name rating with factor 0.5 and 1

Method pair	Score with 0.5	Score with 1
$\langle m1, m1' \rangle$	$20 + 30 + 7 = 57$	$20 + 30 + 14 = 64$
$\langle m1, m2' \rangle$	$20 + 11 + 12 = 43$	$20 + 22 + 24 = 66$

It is evident in Table 3.2 that the approach decides that $m1'$ is the successor of $m1$, because $57 > 43$. If the function would not deduce points for multiple renamings $m2'$ would be selected as successor which would be the wrong choice. This approach is not guaranteed to work in all cases, but it seemed to be a good choice.

Return types are rated the same way, with the only difference that all known renames are applied before comparing the two texts.

Parameter rating For parameters a parameter mapping is generated as described in section 3.4.

The score is calculated based on how many parameters have got a successor with the *exact* same name and type and how many have been *renamed*. *maxScore* is always set to 40 as shown on Table 3.1.

$$params = max(m1.params.size, m2.params.size)$$

$$scoreParam(params, maxScore, exact, renamed) := \frac{maxScore \cdot exact + \frac{maxScore}{3} \cdot renamed}{params}$$

Method body rating The method bodies are normalized as described in section 3.5 using the parameter mapping generated before. The normalized bodies are then rated the same way as names, with the only difference that the highest possible score is 60.

Total score The total score is calculated as the sum of each characteristic rating. Then the method pair with the highest score is matched, but the score has to be at least 150 from a total possible score of 210 points. If the highest score is lower than 150, no method gets matched. The value has been selected by trial and error to sort out methods that are somehow equal, but are not meant to be mapped.

4 Evaluation

The following chapters present three case studies. The first one uses a mutation-based benchmark to analyze which basic types of refactorings the presented approach is able to detect. The second case study investigates the relevance of refactoring detection in real world projects. The third case study tests how well the presented approach performs compared to the bytecode-based approach (cf. subsection 2.2.2)

4.1 Mutation-based Benchmark

Research question *Which refactorings can be correctly detected by the presented approach?*

Study objects To test which refactorings can be detected, two projects have been used. A tiny toy project has been used to see if the approach is generally able to detect this type of refactoring. To verify if the approach is also capable of detecting the refactoring in a more complicated environment, the project ConQAT (~417k LOC) has been used as second instance.

All tested mutations are listed in Table 4.1.

Table 4.1: Tested refactorings

Null-refactoring	Local variable extraction
Add comment to method body	Rename method
Rename package	Rename abstract method
Rename class	Rename method parameter
Move class	Reorder method parameters
Reorder attributes	Extract method
Rename attribute	Pull up method
Rename public attribute in base class	Move static method to other class
Rename local variable	Automated code cleanup[1]

[1] Format source code, Remove unused imports, Remove 'this' qualifier for non static field accesses, Change non static accesses to static members using declaring type, Change indirect accesses to static members to direct accesses (accesses through subtypes)

Study design To actually see which refactorings can be detected, a mutation-based benchmark has been created. A *mutation-based benchmark* evaluates the correctness of a system by applying the system to a set of test cases which have been manually created. Thus, the expected outcome can be specified and compared with the actual result of the system.

Each refactoring from the table above has been applied to the toy project and to ConQAT. Respectively for each project, an unmodified version has been taken as predecessor version. The project with the applied refactoring has been taken as the successor version. The packages, classes etc., which have been changed, have been chosen randomly, trying to pick instances that are widely used in the whole project.

Since all the above changes are pure refactorings the expected outcome is always the same - all changes are non-essential.

Procedure For each test case a zip file has been prepared containing the predecessor and the successor version. The refactoring detection creates a csv file as output. The file lists all methods contained in the project with their corresponding change classification. For all methods which already existed in the previous version, the method bodies have been compared to detect methods that have been changed textually (Affected methods). Methods that did not exist before were classified as added methods, and method bodies that have changed have been classified accordingly. This was implemented by using the framework of the implementation, but with disabled refactoring detection logic.

Results As shown in Table 4.2 most of the refactorings that passed the toy project have also been detected in the big project.

Discussion The study shows that rename refactorings or automated code cleanups are likely to induce a huge amount of non-essential changes and that the presented approach is able to detect most of them. The reason why some refactorings remain undetected in ConQAT is that the approach failed to resolve some of the e.g. method calls, which have been renamed.

All refactoring types, which the approach is generically not able to find, are only local refactorings. Thus they have only a minor influence on the intended use case, namely finding methods that have changed and thus need to be tested, which the approach has been designed for.

Table 4.2: Tested refactorings

	Toy	ConQAT	
Description	Detected	Affected methods	Undetected methods
Null-refactoring	✓	0	-
Add comment to method body	✓	1	-
Rename package	✓	64	-
Rename class	✓	38	-
Move class	✓	187	-
Reorder attributes	✓	1	-
Rename attribute	✓	3	-
Rename public attribute in base class	✓	9	9
Rename local variable	✓	1	-
Local variable extraction	✗	1	1
Rename method	✓	3	-
Rename abstract method	✓	270	4
Rename method parameter	✓	1	-
Reorder method parameters	✓	270	9
Extract method	✗	2	2
Pull up method	✓	1	-
Move static method to other class	✓	4	-
Automated code cleanu[1] on page 22	✓	1275	-

Threats to validity This study only takes into account a few common refactorings, but leaves out a lot of other refactorings. That means there could be refactorings which additionally induce a lot of non-essential changes that the approach may not detect. The refactorings are only applied to two projects and the instances that are refactored have been chosen randomly, which leaves the possibility that the approach not always yields similar results. The benchmark also only tests for refactorings which do not contain essential changes, and thus false-positives cannot occur.

4.2 Refactorings in Real World Projects

Research question It is commonly known that refactorings are applied to make code better. It has also been discussed in chapter 1 that refactorings induce non-essential changes in the source code, which blows up the textual differences between two versions. This fact is a problem in use cases like reviews, code ownership analysis or testing. But the question that remains unanswered is: *How many changes are induced by refactorings in real world projects? And thus is it worth to use a refactoring detection tool?*

Study objects To answer this question some open source projects have been picked to analyze how many refactorings happened. The projects had to be written in Java because the analysis only works for Java projects. The compared versions have been chosen randomly, except that the newer version was always the latest stable release. Table 4.3 shows a brief overview over the projects that have been analyzed.

Table 4.3: Study objects

Study object	Language	Size	Old version	New version	Commits
Apache Ant	Java	267k LOC	1.9.3	1.9.5	254
Apache Tomcat	Java	363k LOC	7.0.42	8.0.0	5064
ConQAT	Java	417k LOC	Rev. 51447	Rev. 52948	1501
jabRef	Java	148k LOC	2.8	2.10	746
jHotDraw	Java	135k LOC	7.4.1	7.6	110

Study design Firstly, a list of textually changed methods has been created for each respective version pair. These results have been compared to the set of methods that the presented refactoring detection classified as non-essential changes.

Procedure To detect methods that contained only non-essential changes the normal implementation has been used. To detect raw changes the refactoring detection logic has been disabled as described in the previous case study. Based on these two method classifications the following results have been obtained.

Results The results are shown in Table 4.4. The table lists the total number of methods per project. The column *Changed* shows the number of methods that have been textually changed or renamed. Finally, the column Δ*Changed* contains the amount of methods that has been filtered out by the refactoring detection, because the changes were non-essential.

Table 4.4: Study objects

Study object	Total	Changed	ΔChanged
Apache Ant	11,487	974	88
Apache Tomcat	19,156	6,974	1,630
ConQAT	19,559	3,628	602
jabRef	6,286	1,208	103
jHotDraw	7,670	1,741	358

Discussion The results clearly show that refactorings have been done in every inspected project. The number of affected methods varies due to the types of refactorings that have been applied. Nevertheless the fact that refactorings have been made in every project reinforces that it is worth using a refactoring detection tool whenever changed methods are considered like in the use cases described in chapter 1.

Threats to Validity At first, the case study only considered open source software as study objects, which may not necessarily have the same characteristics as proprietary software. The study used only two snapshots of the selected projects allowing no generalization of the results. Furthermore, the study just inspected source code written in Java, which may behave differently compared to other languages.

4.3 Comparison to Bytecode-based Analysis

Research question *How does the presented approach perform compared to the bytecode-based approach currently used in Test Gap analysis?*

Study objects To get a meaningful number of methods that only contain non-essential changes two successive versions of ConQAT, namely revision 52650 and revision 52663, where it was known that a big refactoring had been done. Within these revisions the main scanner functionality of ConQAT had been moved and merged into another existing package. This resulted in a lot of rename refactorings producing non-essential changes all over the project. Since detecting those was one of the requirements R3 (cf. chapter 1), using this pair of versions seemed to be a good choice.

Study design To compare the results of both refactoring detection techniques the version pair has been analyzed with the presented source code-based detection and with the bytecode-based technique implemented in Test Gap analysis (cf. subsection 2.2.2).

To decide which approach worked better, a list of refactorings that are truly refactorings has been made. It was assumed that methods which have been classified as non-essential changes by both bytecode and source code-based approach are really refactorings. For those 341 methods that had different classifications it was decided by manual inspection whether the change was a real refactoring.

Procedure To get a first picture of how many methods have been changed by the refactoring, a simple mapping of the methods has been done primarily based on their exact package, class and method-signatures. If those methods which have been matched have not changed on a textual level, the methods are classified as unchanged. Otherwise as changed or added if it did not exist in the previous version. This is referred to as *without refactoring detection* in the following. The results of this analysis are shown in Figure 4.1.

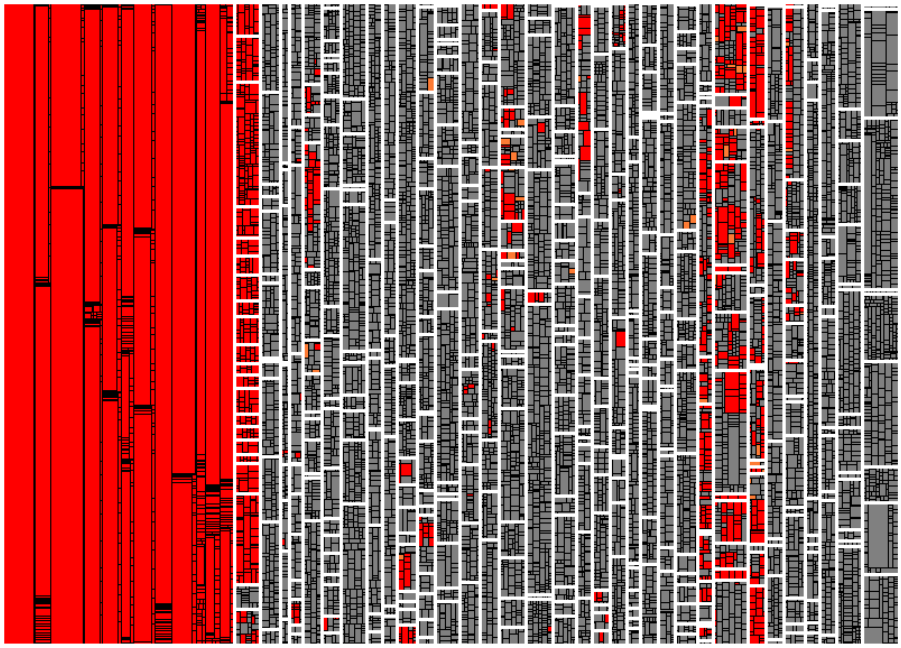
To make the outputs of both analyses comparable, the outputs of the bytecode-based analysis first had to be manually converted to match the format of the source code-based output. Also, some methods had to be excluded from the comparison like interface methods because they do not appear in bytecode. Nevertheless, they are matched and used within the source code-based detection. Switch tables, which are helper methods generated by the compiler to make switch statements faster, have been excluded as well because they have no corresponding methods in source code.

Results The changes between two versions are visualized in a *tree map*. In a tree map each rectangle corresponds to a method in the successor version. Methods that are

in the same class are next to each other in the tree map. The color of the rectangle visualizes the methods classification. Gray means that the method does not contain an essential change (unchanged), orange means that the method has been changed since the predecessor version and red means that the method has been recently added to the source code and was not included in the previous version.

The complete project contained 14,882 methods. Figure 4.1 displays 1,844 methods that have been added and 34 methods that have been modified. This is the result of moving the scanner classes to another package, due to the fact that the methods which are displayed red are in the new package. In the previous version, these methods did not exist, at least not in this package. The modified methods have just been updated to use the new package name. Methods that only use the fully-qualified name of the moved classes in the import statements are not even marked as modified because the refactoring did not change their method bodies.

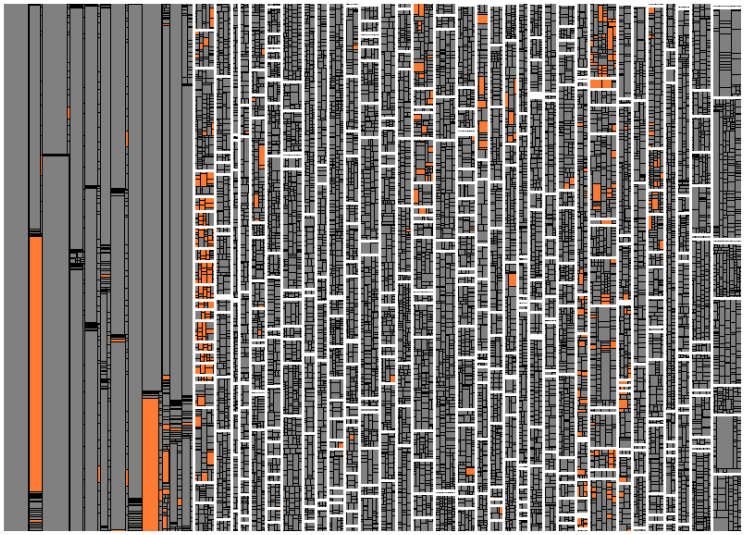
Figure 4.1: Without refactoring detection



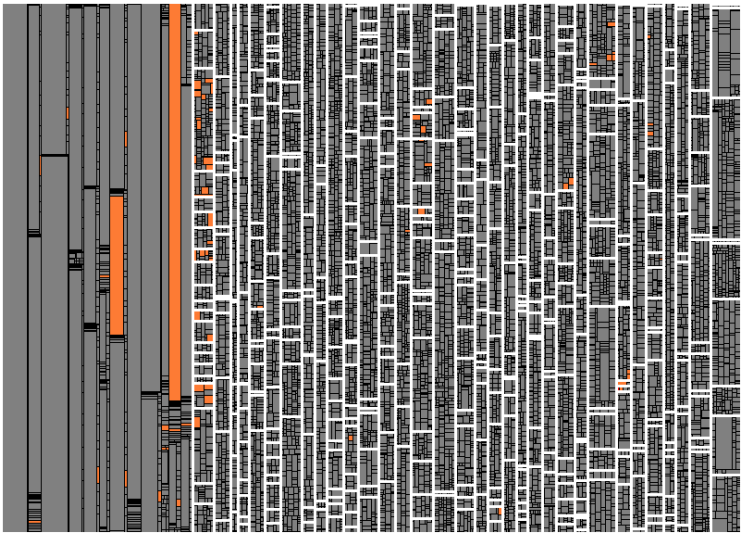
Using refactoring detection, more methods which have previously been classified as added, are classified as changed or unchanged afterwards. Figure 4.2a demonstrates the results of the bytecode-based approach and Figure 4.2b shows the results of the approach presented in this thesis.

Figure 4.2: Comparison between bytecode and source code-based refactoring detection

(a) Bytecode-based



(b) Source code-based



It is obvious that large amounts of red rectangles have been detected as moved methods by both approaches resulting in areas drawn as gray or orange instead of red. Nevertheless Figure 4.2b detected more refactorings.

Comparing the results in more detail, both approaches made some one-sided mistakes. The results are shown in Table 4.5.

Table 4.5: Detected Refactorings

	Correct classifications	False-negative classification	False-positive classification
Bytecode-based	1623 methods	336 methods	0 methods
Source code-based	1932 methods	27 methods	0 methods

Time and memory consumption The running time for both approaches has been measured by running the analysis on the project 5 times in a row on a Laptop (Intel i7 2620M 2.7GHz, 4GB RAM, SSD). The average over all running times were used for the following comparison. For bytecode analysis the time it took to build the project was not included because for the Test Gap use case it can be assumed that the project has already been built by the build server when the analysis starts. For the sake of completeness the build process took 134s in total. The detailed results are presented in Table 4.6.

Table 4.6: Time and memory requirements

	Avg. running time	Avg. max. memory usage
Bytecode-based	386s	1048MB
Source code-based	140s	1676MB

With a speedup of 2.8 times compared to the bytecode analysis, the source code-based implementation improves the overall performance of the analysis, even if its maximal allocated memory was 628MB higher.

Discussion The first thing to note is that both approaches fulfilled requirement R4 because they did not classify any method as a refactoring, even though it contained essential changes. So both had a precision of 100%. Using the values from above, bytecode-based refactoring detection yields a recall (% of refactorings found) of $\frac{1623}{1623+336} = 83\%$ and the source code-based approach scores a recall of $\frac{1932}{1932+27} = 99\%$.

Undetected methods The 27 methods the approach could not detect have been overseen due to incomplete statement resolution. For example the implementation did not recognize that for calls to the method `token(ETokenType type)`, where `ETokenType` is an enum, the statements `token(EOF)` and `token(ETokenType.EOF)` are semantically equal.

Threats to validity The results presented above cannot be generalized to other refactorings because only one project has been tested. The evaluation also just used one implementation for Java. So in general it is not guaranteed that the approach yields similar results for other programming languages as well. For refactorings that are done on the interface of some third-party libraries, where the source code is either not available or not included in the analysis, the results may differ as well, because the approaches may fail to resolve those dependencies. Another problem could be that the probability of false-positives is very low in the tested project because most of the methods only contained non-essential changes, which makes false-positives unlikely. So projects containing more essential method changes would be more likely to produce false-positives.

Since the only part of the refactoring detection that guesses is the matching, it is the likeliest point where false-positives could be produced. But false matchings of methods lead the approach to rename all calls to this method to the name of the matched method. This always leads to classifying all methods which contain calls to this method, as changes. This classification is false, but produces a false-negative, which is not good, but allowed (R4).

5 Conclusion

The task was to design, implement and evaluate a refactoring detection technique which could replace the already existing bytecode-based refactoring detection used in Test Gap analysis.

The presented approach was to build a tree model of the input system, which had all methods of the system as its leafs. Then the methods of the predecessor version have been matched to the methods of the successor version of the system. Renamings that could be derived from the matchings were saved. After doing some additional post-processing, each method pair which was matched, was normalized using the detected renamings. By comparing the normalized methods a classification was made whether the method contains essential changes or only non-essential changes.

The requirements which had to be met by the approach were (see chapter 1):

R1 Source code-based

R2 Transferability to other languages

R3 Recognition of most frequently used refactorings, especially those that introduce a lot of non-essential changes in source code

R4 Precision of 100%

R5 Reasonable running time

Looking at the requirements, the implementation meets requirement R1, because it works on source code basis, which in turn led to a speedup of the refactoring detection (R5), without even optimizing it. R2 is fulfilled as well, since the presented technique is partly language independent as long as it does not work on statement level. For languages similar to Java the normalization function can be easily adapted and for other programming languages it can be implemented as well with reasonable effort. R3 can also be considered as met, as the case study presented in section 4.3 mainly contained rename refactorings, which the presented approach detected with a rate of 99%. R4, as already mentioned in the previous chapter, is fulfilled as well, because no methods that were classified as refactorings contained essential changes. In conclusion, the approach met all requirements and led to a good result.

6 Future Work

For future work the following aspects must be considered. The case study presented in section 4.3 has to be performed with more projects, which include all kinds of refactorings, project sizes and more essential changes, to see if the precision is always 100% and thus can be generalized.

Furthermore, if has to be evaluated whether the approach really works with other languages as well. Therefore, porting to other languages, consecutive testing and evaluation has to be performed too.

Finally, the current implementation only works with rename and some move refactorings. But as stated in Fowler's book [2], there are 72 possible refactorings which cannot all be detected with this implementation by design. Possibly the approach of Prete et al. [7] could be merged with the solution presented in this thesis to detect more different kinds of refactorings, with the guarantee not to produce false-positive classifications. But since most of those refactorings are only locally applied, and thus do not induce changes all over the project, it is not that important to detect them for the intended use case.

List of Figures

List of Listings

List of Tables

Bibliography

[1] S. Eder, B. Hauptmann, M. Junker, E. Juergens, R. Vaas, and K. H. Prommer. "Did we test our changes? Assessing alignment between tests and development in practice." In: *2013 8th International Workshop on Automation of Software Test, AST 2013 - Proceedings* (2013), pp. 107–110. DOI: 10.1109/IWAST.2013.6595800.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. "Refactoring: Improving the Design of Existing Code." In: *Xtemp01* (1999), pp. 1–337. ISSN: 14359456. DOI: 10.1007/s10071-009-0219-y.

[3] B. Hummel, L. Heinemann, E. Juergens, L. Heinemann, and M. Conradt. "Index-based code clone detection: incremental, distributed, scalable." English. In: *2010 IEEE International Conference on Software Maintenance*. IEEE, Sept. 2010, pp. 1–9. ISBN: 978-1-4244-8630-4. DOI: 10.1109/ICSM.2010.5609665.

[4] D. Kawrykow and M. P. Robillard. "Non-essential changes in version histories." In: *2011 33rd International Conference on Software Engineering (ICSE)* (2011), pp. 351–360. ISSN: 0270-5257. DOI: 10.1145/1985793.1985842.

[5] M. Kim and D. Notkin. "Discovering and representing systematic code changes." In: *Proceedings - International Conference on Software Engineering* (2009), pp. 309–319. ISSN: 02705257. DOI: 10.1109/ICSE.2009.5070531.

[6] S. Kim, K. Pan, and E. J. Whitehead. "When functions change their names: Automatic detection of origin relationships." In: *Proceedings - Working Conference on Reverse Engineering, WCRE* 2005 (2005), pp. 143–154. ISSN: 10951350. DOI: 10.1109/WCRE.2005.33.

[7] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. "Template-based reconstruction of complex refactorings." English. In: *2010 IEEE International Conference on Software Maintenance*. IEEE, Sept. 2010, pp. 1–10. ISBN: 978-1-4244-8630-4. DOI: 10.1109/ICSM.2010.5609577.

[8] D. Steidl and F. Deissenboeck. "How do Java Methods Grow?" In: *submitted for SCAM 2015*. 2015.

[9] P. Weißgerber and S. Diehl. "Identifying refactorings from source-code changes." In: *Proceedings - 21st IEEE/ACM International Conference on Automated Software Engineering, ASE 2006* (2006), pp. 231–240. ISSN: 1527-1366. DOI: 10.1109/ASE.2006.41.