Error Detection by Refactoring Reconstruction

Carsten Görg
Saarland University
Computer Science
D-66041 Saarbrücken
Germany
goerg@cs.uni-sb.de

Peter Weißgerber
Catholic University Eichstätt
Computer Science
D-85072 Eichstätt
Germany
peter.weissgerber@ku-eichstaett.de

ABSTRACT

In many cases it is not sufficient to perform a refactoring only at one location of a software project. For example, refactorings may have to be performed consistently to several classes in the inheritance hierarchy, e.g. subclasses or implementing classes, to preserve equal behavior.

In this paper we show how to detect incomplete refactorings – which can cause long standing bugs because some of them do not cause compiler errors – by analyzing software archives. To this end we reconstruct the class inheritance hierarchies, as well as refactorings on the level of methods. Then, we relate these refactorings to the corresponding hierarchy in order to find missing refactorings and thus, errors and inconsistencies that have been introduced in a software project at some point of the history.

Finally, we demonstrate our approach by case studies on two open source projects.

1. INTRODUCTION

Refactoring is the process of changing a software system such that it does not alter the external behavior of the code, but improves its internal structure [2]. In many cases the same refactoring has to be applied to more than one entity to achieve that the behavior really does not alter. For example, changing a method signature often requires to change the signature of methods in sub, super, and sibling classes as well. To help programmers with this task modern integrated development environments, like ECLIPSE [7], provide (semi-) automated application of refactoring to a software project. But unfortunately, not all programmers make consistent use of this feature and possibly introduce bugs, which are hard to find later on.

In this paper, we show an approach to investigate the application of refactorings over the lifetime of a software system. In particular we check if refactorings have been performed consistently, i.e. in such a way that the behavior of the software system has not altered. We concentrate on refactorings of types *Add/Remove Parameter* and *Rename Method*. When a refactoring of one of these types is applied to a method in a class, it should also be applied to the corresponding methods of subclasses in most cases, and in some cases even of sibling classes. If a developer fails to do so, two kinds of errors can occur:

- *An interface method or an abstract method is no longer implemented in a subclass.* This kind of error results in compile time errors, and thus, is easy to detect.
- *The refactored method is inherited by a subclass instead of being overwritten.* However, the project can still be compiled without any problems and, thus, the developer may not notice the problem for a long time.

Our approach is applicable in two scenarios: First, we can search in existing software repositories for incomplete refactorings that have been done in the past and have not been corrected yet. In addition to this, we can assist the developer with the daily work: Every time the developer commits source code to the repository a tool can check the committed code for incomplete refactorings and warn if necessary.

The remainder of this paper is organized as follows: First, we explain in Section 2 how refactorings can be extracted from a software archive. Then, in Section 3 we show how to check for consistency of the reconstructed refactorings. Section 4 presents case studies of two open source projects. After that, we outline related work in Section 5. Finally, Section 6 summarizes our findings.

2. UNCOVERING REFACTORINGS

In this section we present a technique to detect refactorings on the level of methods in a software archive managed by CVS [1]. At first, we explain how we preprocess the repository data to get easy and fast access to it. After that, we take a closer look on how to retrieve classes and methods of JAVA files. We need this information in the following step where we analyze if methods have been refactored by renaming them, or adding resp. removing parameters. After that, we discuss in few words how we deal with additional changes to refactored methods and with ambiguous refactorings. Finally, we explain how we relate refactorings to

MSR'05, May 17, 2005, Saint Louis, Missouri, USA

complete configurations of the software project. A more detailed and formal description of this technique is described in [3].

2.1 Preprocessing the CVS Data

Unfortunately, the direct access on the data is much too slow, and furthermore, some information has to be recovered from different places of the repository. Thus, the first step of our technique is to extract the repository completely, recover information where necessary, and store this data in a relational database. The details of this extraction step are described in [8]. After the extraction we can access the following information:

Versions. A *version* describes one revision of a file in the `CVS` repository (e.g. file `org/epos/epos.java` in revision 1.4). For each revision in the repository we store information about the committer, the log message, the timestamp, the state, the predecessor revision if one exists, and the text.

Transactions. A *transaction* is the set of versions that have been committed to the repository at the same time by the same developer. As `CVS` splits commits that contain more than one file into single check-ins for each file and does not store which of these check-ins have been issued together, we use a sliding time window heuristic to recover transactions quite precisely.

Additionally, we need information about particular *configurations*. A configuration is a set of versions of distinct files. In our application, we are only interested in *active configurations after transactions*. An active configuration after a transaction is the set of versions a developer can access in his working directory after performing the transaction.

If we want to examine a new commit to the repository instead of searching existing failures, it is sufficient to consider the set of versions belonging to the current commit and to build the active configuration after this committed transaction.

2.2 Parsing Syntactical Blocks of Versions

To gather information about which classes and methods are contained in a `JAVA` file (and thus, may be affected by refactorings) we use a light-weight parser that identifies a) classes in versions and b) methods in classes. The classes are identified by its fully-qualified name while methods are identified by their signature, e.g. `parseInt(String):int`. If we compare the classes of a version v to the classes of its predecessor version v', we note by $\mathsf{COMMON_C}(v, v')$ the set of classes that exist in both v and v'.

For each class $c \in \mathsf{COMMON_C}(v, v')$ we can now compute the sets of added, removed, and common methods by comparing the methods contained in the class in v with the methods in the corresponding class in v':

- $\mathsf{ADDED_M}(v, v', c)$ method that have been added to c;
- $\mathsf{COMMON_M}(v, v', c)$ methods that are contained in c in both versions v' and v;
- $\mathsf{REMOVED_M}(v, v', c)$ methods that have been deleted from c.

2.3 Identifying Local Refactorings

To find refactorings performed in single classes we iterate over the set V_r of all versions of `JAVA` files in the repository. As refactorings describe changes with respect to the predecessor version, we ignore versions that have no predecessor. For all remaining $v \in V_r$, we take the predecessor version v' and test in all classes $c \in \mathsf{COMMON_C}(v, v')$ if we can find one of the following refactorings:

Rename Method.

If we find a method $m_1 \in \mathsf{REMOVED_M}(v, v', c)$ and a method $m_2 \in \mathsf{ADDED_M}(v, v', c)$ that have exactly the same text, the same return type, the same parameters, but different names we consider this as a *Rename Method* refactoring.

Add Parameter.

If we find a method $m_1 \in \mathsf{REMOVED_M}(v, v', c)$ and a method $m_2 \in \mathsf{ADDED_M}(v, v', c)$ that have the same name and the same return type, but m_2 has additional parameters with respect to m_1, we consider this as an *Add Parameter* refactoring. The *Remove Parameter* refactoring is recognized analogously.

2.4 Impure and Ambiguous Refactorings

A major problem in parsing the version archive for refactorings is that often the refactorings are *impure*: The developer has not only performed the refactoring, but has changed other things at the same location at the same time, or the developer has performed two different refactorings on the same entity. Our approach is not capable to find *Rename Method* and *Add/Remove Parameter* refactorings that have been applied to the same entity in the same transaction. In addition, these refactorings cannot be recognized if other changes have been simultaneously performed to the method signature (i.e. name, parameter list, and return type). However, if only the body of the method has been changed together with the refactoring, we still detect *Add/Remove Parameter* refactorings. As a consequence, it can happen that we fail to detect some inconsistent refactorings: Assume that in class `A` a method has been renamed and at the same time an additional parameter has been added. Furthermore, assume that class `B` is a subclass of `A`. Thus, the corresponding method in `B` has to be refactored the same way. But, as we cannot detect the refactoring in `A` we also cannot detect if `A` has been refactored but `B` has been missed.

Unfortunately, it is not always possible to unambiguously identify all refactorings as the following example illustrates: a class contains the methods $\mathtt{m}(\mathtt{t}_1, \mathtt{t}_2) : \mathtt{t}_\mathtt{r}$ and $\mathtt{m}(\mathtt{t}_2, \mathtt{t}_3) : \mathtt{t}_\mathtt{r}$ and after a new transaction it contains instead of these two the new method $\mathtt{m}(\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_3) : \mathtt{t}_\mathtt{r}$. Now it is undecidable if this is an *Add Parameter* refactoring from $\mathtt{m}(\mathtt{t}_1, \mathtt{t}_2) : \mathtt{t}_\mathtt{r}$ by adding $\mathtt{t}_3$ or from $\mathtt{m}(\mathtt{t}_2, \mathtt{t}_3) : \mathtt{t}_\mathtt{r}$ by adding $\mathtt{t}_1$. In such cases, we take all matching refactorings into account.

2.5 Relating Refactorings to Configurations

To detect errors it is not sufficient only to look at the classes that have been currently changed and may contain refactorings, but also at the other classes that have been part of

the project and (maybe erroneously) not have been updated when the developer has performed the check-in to the repository. Thus, additionally to the changed versions, for each file that has not been changed in a transaction we take the most recent version into account and parse it for its classes and methods. We call the set of changed versions and most recent versions of non-changed files the *configuration active after transaction t*.

2.6 Reconstructing the Class Hierarchy

Before we can check if refactorings have been applied consistently to related classes with respect to the class hierarchy, we have to construct the inheritance tree of the examined JAVA project. Thus, we iterate over the set of transactions and build for each the configuration active after it. For each such configuration we know the set of *project classes* – these are the JAVA classes in the workspace of the developer after the transaction. Thus, the inheritance tree for both *implements* and *extends* relations is built by iterating over these project classes, parsing them for the declarations after the `implements` resp. `extends` keyword and relating the found class references (using the import declarations) to a) the classes of the JAVA standard API and b) the classes contained in JAR files.

3. CHECKING CONSISTENCY

In this section we explain how we check if the reconstructed refactorings of types *Rename Method* and *Add/Remove Parameter* have been applied consistently to the software project. For each refactoring we compute a list of other possible candidates in the current configuration and check if the refactoring has been applied also to them. If not, we regard this as a possible inconsistency.

The list of possibles candidates is computed as follows: Let ref be a refactoring changing the parameter list of method m in class c in configuration conf_1 from params_1 to params_2 in configuration conf_2. If in configuration conf_2 exists a superclass, subclass or sibling class[1] of class c that contains the method m with parameter list params_1 then this method is a possible candidate for a missing refactoring. Analogously, we compute further candidates by taking refactorings renaming a method into account.

The methods found as candidates split into two different categories:

- **Methods in subclasses:** If methods in a class are refactored it is likely that overwritten methods in subclasses should be refactored the same way. For this type of error candidates one can distinguish between two subtypes: If the superclass is an interface, the unrefactored subclasses do not implement the interface correctly anymore and thus, cannot be compiled successfully. Otherwise, the affected subclass inherits the refactored method from the superclass and additionally, holds its own unrefactored method. Thus, the class can be compiled, but may behave incorrectly.

[1] a sibling class of class c is an arbitrary subclass of a superclass of class c.

- **Methods in sibling classes:** There are also cases when sibling classes have to be updated although the superclass has not to be updated. For example, in one transaction of JEDIT at the same time in both classes `EnhancedMenuItem$MouseHandler` and `EnhancedCheckBoxMenuItem$MouseHandler` the method `mouseClicked` has been renamed to `mouseReleased`. As both classes are subclasses of the standard JAVA class `MouseAdapter` the actions implemented in these methods are not instantly triggered any more when the mouse has been clicked, but not until the mouse has been released again. Clearly, this refactoring should be applied at the same time to all classes extending from `MouseAdapter` to preserve a consistent user interface.

As we examine the whole software archive and start with the oldest configurations, it is possible that the missing parts of a detected inconsistent refactoring have been added some transactions later (this means the refactorings has been performed using multiple transactions). Hence, for each candidate we iterate over all configurations with a timestamp later than the considered configuration and look if the refactoring has been applied to the corresponding method. If we find such a later configuration, the inconsistency (and thus the error) has been resolved.

4. CASE STUDIES

We applied our techniques to detect inconsistent refactorings to the software archives of the open source projects JEDIT and TOMCAT. We found five candidates for erroneous transactions in JEDIT and seven in TOMCAT. For JEDIT two of these candidates are unrefactored methods in subclasses – both have been refactored in later transactions – and three are methods in sibling classes. For TOMCAT three candidates are methods in subclasses and four are methods in sibling classes. None of these unrefactored methods have been updated later. In the following paragraphs we explain some noticeable transactions in more detail.

4.1 Unrefactored Methods in Subclasses

We found an example of the subclasses type in Transaction 876 of JEDIT: An additional parameter has been added to the method `foldLevelChanged` of the interface `BufferChangeListener`. This interface is implemented by two classes: `BufferChangeAdapter` and `JEditTextArea$BufferChangeHandler`. Thus, these classes should have been updated accordingly to ensure that they can be compiled. Interestingly, only `JEditTextArea$BufferChangeHandler` has been initially updated. One month later the other class has been refactored, too, allowing it to be compiled again.

Also in JEDIT we found a second example where the developers seem to have noticed their mistake and corrected it some transactions later: In Transaction 1241 a developer has added a boolean parameter to the method `addTokenHandler` in the class `DefaultTokenHandler`. One day later, in the following transaction, the subclass `DisplayTokenHandler` has been updated accordingly. Note, that in this example the superclass is not an interface. This means, even the incorrect configuration was syntactically correct and did not produce compiler errors.

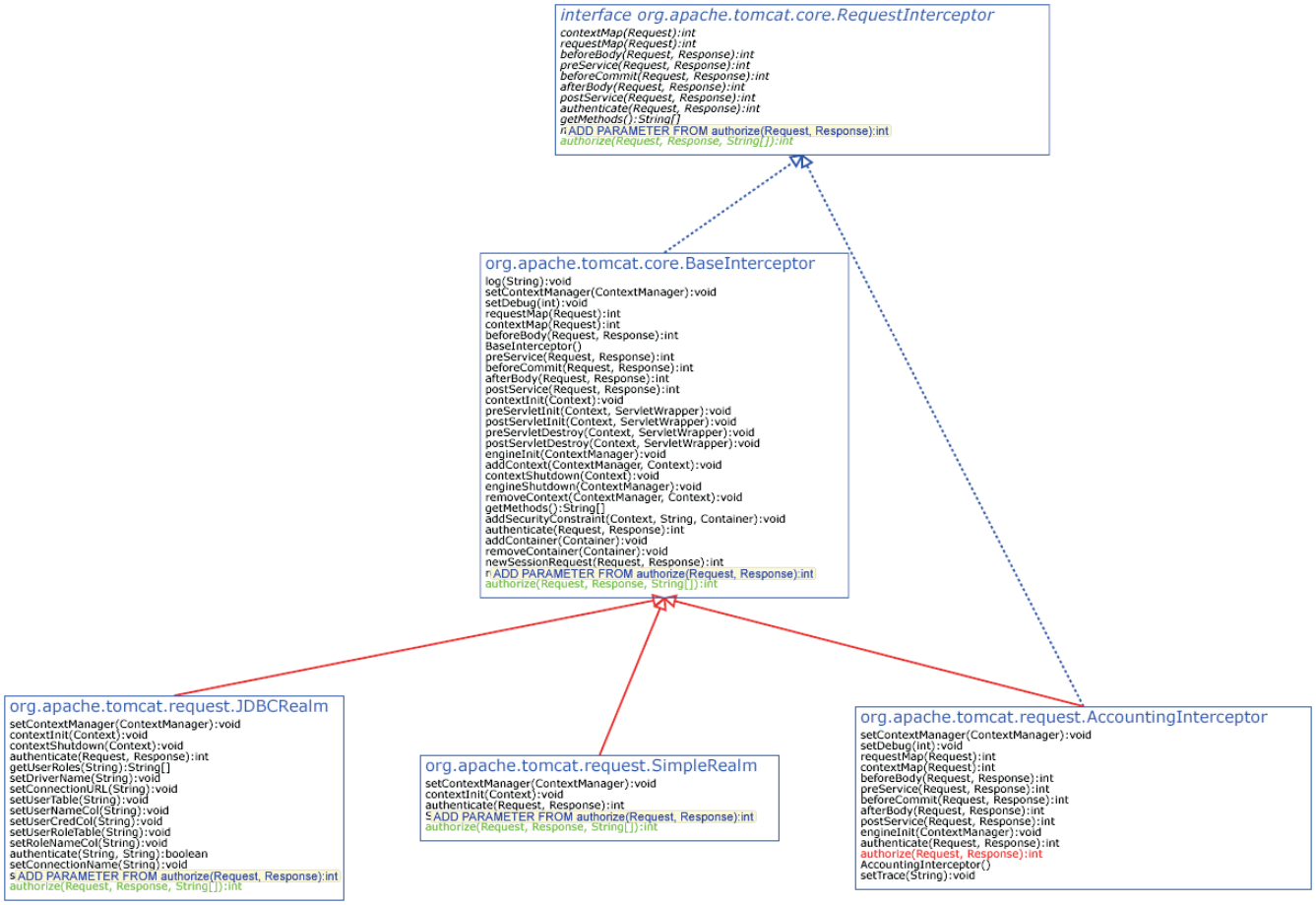

Figure 1: Add Parameter refactorings in transaction 1971 of TOMCAT.

But we also found transactions where the unrefactored class has not been updated later. For example, in Transaction 1475 of TOMCAT the method `handleTagBegin` has been extended by one parameter in different classes of the package `org.apache.jasper.compiler`, but not in the class `DumbParseEventListener`. The documenting comments in the source code of the class tell that the class is a testing class that should be removed some time.

This shows that our approach only finds candidate methods that may be buggy. Actually, these methods should be audited by a developer.

Figure 1 shows a more complex example for a maybe inconsistent *Add Parameter* refactoring in TOMCAT. In this visualization the involved classes and the inheritance relations between them are displayed in the UML notation, refactored methods are marked green (a tooltip describes the refactoring) and missing refactorings are colored red. In Transaction 1971 the method `authorize` of the interface `RequestInterceptor` has been extended by one parameter of type `String[]`. This new parameter provides the roles which the person that should be authorized owns (e.g. "admin"). Consequently, the class `BaseInterceptor` that implements this interface has been updated, too. Moreover, two of three subclasses of `BaseInterceptor` have also been changed: `SimpleRealm` and `JDBCRealm`. Surprisingly, the class `AccountingInterceptor` that both *implements* `RequestInterceptor` and at the same time *extends* `BaseInterceptor` has not been updated. Anyway, this class can be successfully compiled because the `authorize` method as needed because of the interface is inherited from the superclass `BaseInterceptor`.

To get a deeper understanding of what happened in the described transaction we checked out the source code of the involved classes at the time of the transaction. We found that in the unrefactored class `AccountingInterceptor` the questionable method always returns the value "0" for "authorized". This is exactly what the updated method that is inherited from `BaseInterceptor` does. Thus, there was no need to update the method in this class because the inherited one does the right thing and the unrefactored one does the same and can be used as abbreviation.

4.2 Unrefactored Methods in Sibling Classes

As we have explained in Section 3 in some cases it is also necessary to update sibling classes although the superclass of a class has not been refactored. In JEDIT we found three and in TOMCAT four transactions where a class has been refactored on method level but the siblings have not been updated. None of these have been updated later in the history.

As siblings do not inherit from each other, they can implement completely different methods. Thus, again the found transactions are only candidates and have to be checked by a developer.

For example in Transaction 3240 of `TOMCAT`, a parameter has been added to the method `addTagRules` in the class `ProfileLoader`. This class is a subclass of `BaseInterceptor` which has two other subclasses: `ContextXMLReader` and `ServiceXMLReader`. In these classes the described refactoring has not been performed.

5. RELATED WORK

General information on refactoring are presented in Fowler's book [2]. Demeyer et. al presented some metrics-based heuristics [6] to detect refactorings in successive configurations of software systems. They primarily concentrated on movements, splits, and merges of methods and performed case studies on three open source projects. In contrast to our work, they did not access the software archive to get successive configurations. Software metrics like the McCabe complexity [4] can also give hints about files and classes that are likely to contain errors. Ostrand and Weyuker [5] used a different approach to detect and predict possible problems in a software. They mined bug databases in order to predict fault-prone files.

6. CONCLUSIONS AND OUTLOOK

In this paper we introduced an approach to reconstruct refactorings on method level from software archives. Then we explained how to check if they have been consistently applied. The case studies on the open source projects `JEDIT` and `TOMCAT` show that our approach allows to detect candidates for incomplete refactorings. In two cases, the candidates have been applied later and so it turned out that they have been really missing. In two other cases, they have also been missing, but have not been applied, because they concerned classes, which have no longer been under active development. The remaining cases have to be inspected by the developer to decide whether the refactorings are really missing or not.

For future work we plan to investigate how to reconstruct more complex types of refactorings from software archives. Moreover, there are additional refactorings that could be checked for consistency without altering our current architecture but are not yet implemented: For example the *Extract Interface* refactoring that creates an interface containing the methods of a class and lets the class implement this interface could be detected by comparing the method declarations of new interfaces to the methods of existing classes. In addition to this, the *Generalize Type* refactoring that changes the type of an object to a more general type could be recognized using the class hierarchy.

As our approach can be used to support developers in their daily work to prevent errors in refactoring tasks, it would be desirable to let our algorithms run in the background while the developer is working. To achieve this we are planning to integrate it into a development environment like `ECLIPSE`.

Furthermore, we plan to extend our case study to more software projects of different size, complexity, and age.

7. REFERENCES

[1] P. Cederqvist. Version Management with CVS. `http://www.cvshome.org/docs/manual/`.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2001.

[3] C. Görg and P. Weißgerber. Detecting and Visualizing Refactorings from Software Archives. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC 2005)*, St. Louis, Missouri, U.S., 2005.

[4] T. J. McCabe. *A Complexity Measure*. IEEE Transactions on Software Engineering, Vol. 2, 1976.

[5] T. J. Ostrand and E. J. Weyuker. A Tool for Mining Defect-Tracking Systems to Predict Fault-Prone Files. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, Edinburgh, Scotland, U.K., 2004.

[6] O. N. Serge Demeyer, Stéphane Ducasse. Finding Refactorings via Change Metrics. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2000)*, Minneapolis, Minnesota, U.S., 2000.

[7] The Eclipse Foundation. Eclipse Homepage. `http://www.eclipse.org`.

[8] T. Zimmermann and P. Weißgerber. Preprocessing CVS Data for Fine-Grained Analysis. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, Edinburgh, Scotland, U.K., 2004.