Detecting and Visualizing Refactorings from Software Archives

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

We perform knowledge discovery in software archives in order to detect refactoring on the level of classes and methods. Our REFVIS prototype finds these refactorings in CVS repositories and relates them to transactions and configurations. Additionally, REFVIS relates movements of methods to the class inheritance hierarchy of the analyzed project.

Furthermore, we present our visualization technique that illustrates these refactorings. REFVIS provides both a class hierarchy layout and a package layout and uses color coding to distinguish different kinds of refactorings. Details on each can be displayed on demand using mouse-over tooltips.

Finally, we demonstrate by case studies on two open source projects how REFVIS facilitates understanding of refactorings applied to a software.

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[9]. In all non-trivial software projects applying refactoring is a standard task of programmers and many integrated development environments support (semi-)automated refactoring. Imagine the following situation: a programmer leaves a project for some months and rejoins afterwards. After resuming the work he realizes that the code is unfamiliar now: classes have been moved to other packages, methods to other classes, methods have been renamed, or their parameter lists or visibilities have been changed. The programmer feels quite frustrated as it will take a long time to understand the changed code.

In this paper, we show how to help the programmer in detecting and understanding refactoring applied to a software archive. In particular, our REFVIS tool currently detects and visualizes refactorings of two different types:


The remainder of this paper is organized as follows: At first, we describe in Section 2 how refactorings can be extracted from a software archive. Then, in Section 3 we introduce our technique to visualize these refactorings. Section 4 presents the case study we performed on the archives of two open source projects. After that, Section 5 outlines related work by other researchers. Finally, Section 6 resumes the conclusions we have discovered and Section 7 closes with an overview of our future work.

2. Uncovering Refactorings

In this section we present a technique to detect refactorings in a software archive managed by CVS [3]. 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 get the syntactical blocks of JAVA files. We need this information in the following step where we analyze if these blocks have been refactored.

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 [22]. 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 version in the repository we also
store information about the committer, the log message, the branches the version belongs to, the timestamp of the check-in, the state (e.g., “dead” for actually deleted revisions), and the predecessor revision if one exists. The text (i.e. for program files the source code) of a version can be retrieved on demand.

Transactions. A transaction is the set of versions that have been committed to the repository at the same time by the same developer. Unfortunately, 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. Thus, we use a sliding time window heuristic to recover transactions quite precisely. For every transaction we additionally store the timestamps of the transaction start and end.

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. Briefly, the active configuration after a transaction is the set of versions a developer can access in his working directory after performing the transaction. In Section 2.8 we define such configurations precisely.

CVS repositories can contain branches where alternative versions of files exist. In this paper we only consider versions in the main development line which is called trunk by most projects. As branches that contain well proved changes normally get merged into the trunk, this restriction does not cause to miss refactorings the developers considered worthwhile. The drawback, however, is that refactorings that have been done in separate transactions on a branch are displayed together because they are contained in a single merge.

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 these syntactical blocks. Let \( V \) be the set of all versions, \( C \) be the set of all class names, and \( M \) be the set of all method signatures. Then, the parser is represented by the following two functions:

- \( \text{parse}_C : V \to \mathcal{P}(C) \) returns the names of the classes contained in a version (or the empty set if the version belongs to a non-JAVA file).
- \( \text{parse}_M : V \times C \to \mathcal{P}(M) \) takes a version and a class name and returns the signature of each method in the class. The signature consists of the name, the parameter list, and the return type of the method.

If we apply these functions to a version \( v \) and its predecessor version \( v' \), we can compute the sets of added, removed, and common classes:

- \( \text{ADDED}_C(v, v') = \text{parse}_C(v) \setminus \text{parse}_C(v') \)
- \( \text{REMOVED}_C(v, v') = \text{parse}_C(v') \setminus \text{parse}_C(v) \)
- \( \text{COMMON}_C(v, v') = \text{parse}_C(v') \cap \text{parse}_C(v) \)

Note, that “common classes” does not mean that these classes are unchanged with respect to the predecessor version, but only that the class is present in both versions. The same holds analogously for the term “common methods”.

For each common class \( c \in \text{COMMON}_C(v, v') \), we can now compute the added, removed, and common methods:

- \( \text{ADDED}_M(v, v', c) = \text{parse}_M(v, c) \setminus \text{parse}_M(v', c) \)
- \( \text{REMOVED}_M(v, v', c) = \text{parse}_M(v', c) \setminus \text{parse}_M(v, c) \)
- \( \text{COMMON}_M(v, v', c) = \text{parse}_M(v', c) \cap \text{parse}_M(v, c) \)

To get further information about the methods of classes contained in versions we define some auxiliary functions:

- \( \text{name}_M(v, c, m) \) returns the method name,
- \( \text{parameters}(v, c, m) \) returns the parameters (only types, not the names of the variables),
- \( \text{returntype}(v, c, m) \) returns the return type,
- \( \text{signature}(v, c, m) \) returns the signature,
- \( \text{body}(v, c, m) \) returns the body,
- \( \text{visibility}(v, c, m) \) returns the visibility
  (either public, protected, default, or private) of method \( m \) contained in class \( c \) in version \( v \).

In the following two sections we explain how we use this information about added, deleted, and common classes and methods to find refactorings on single files as well as on multiple files.

2.3. Refactorings on Single Files

To find refactorings performed on single files 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 if we can find one of the following refactorings:

Rename Method. For every class \( c \in \text{COMMON}_C(v, v') \) we consider all method pairs \((m_r, m_a) \in \text{REMOVED}_M(v, v', c) \times \text{ADDED}_M(v, v', c)\) that fulfill the following condition:

\[
\text{name}_M(v', c, m_\ell) \neq \text{name}_M(v, c, m_u) \\
\wedge \text{parameters}(v', c, m_\ell) = \text{parameters}(v, c, m_u) \\
\wedge \text{returntype}(v', c, m_\ell) = \text{returntype}(v, c, m_u) \\
\wedge \text{body}(v', c, m_\ell) = \text{body}(v, c, m_u)
\]

These pairs \((m_r, m_a)\) describe a Rename Method refactoring: The method with name \( \text{name}_M(v', c, m_\ell) \) in class \( c \) has been renamed to \( \text{name}_M(v, c, m_u) \).
**Hide Method.** For every class \( c \in \text{COMMON}_c(v,v') \) we consider all methods \( m \in \text{COMMON}_m(v,v',c) \) that fulfill the following condition:

\[
\begin{align*}
\text{signature}(v',c,m) &= \text{signature}(v,c,m) \\
\land \text{visibility}(v',c,m) &= "\text{public}" \\
\land \text{visibility}(v,c,m) &= "\text{private}"
\end{align*}
\]

Each such method \( m \) in class \( c \) has been hidden.

**Add Parameter.** For every class \( c \in \text{COMMON}_c(v,v') \) we consider all method pairs \( (m_r, m_a) \in \text{REMOVED}_m(v,v',c) \times \text{ADDED}_m(v,v',c) \) that fulfill the following condition:

\[
\begin{align*}
\text{name}_m(v',c,m_r) &= \text{name}_m(v,c,m_a) \\
\land \text{returntype}(v',c,m_r) &= \text{returntype}(v,c,m_a) \\
\land \text{parameters}(v',c,m_r) \subseteq \text{parameters}(v,c,m_a)
\end{align*}
\]

This means that at least one parameter has been added to the method \( m_r \) in the class \( c \). The **Remove Parameter** refactoring can be recognized analogously.

### 2.4. Refactorings on Multiple Files

To detect refactorings that affect more than one file, e.g., methods that are moved from one class (contained in one file) to another class (which may be located in another file), it is not sufficient to regard single versions as in the previous section. Instead, we have to consider complete transactions. This is sufficient because we assume that developers commit all the changes they have done in one single work step together. Thus, if something has been moved from one file to another, the corresponding new versions of both files belong to the same transaction.

In the following, we describe how we find refactorings performed in a transaction \( t = \{v_1, \ldots, v_n\} \).

**Move Class.** For every \( v_i, v_j \in t \) with previous versions \( v_i^t \) resp. \( v_j^t \) we look for class pairs \( (c_r, c_a) \in \text{REMOVED}_c(v_i,v_i^t) \times \text{ADDED}_c(v_j,v_j^t) \) with \( c_r = c_a \land \text{parse}_m(v_i^t,c_r) = \text{parse}_m(v_j^t,c_a) \).

This means the class \( c_r \) has been moved from \( v_i \) to \( v_j \) where it is now known as \( c_a \).

**Move Method.** For every \( v_i, v_j \in t \) with previous versions \( v_i^t \) resp. \( v_j^t \) we first look for all class pairs \( (c_1, c_2) \in \text{REMOVED}_c(v_i,v_i^t) \cup \text{COMMON}_c(v_i,v_i^t) \times \text{COMMON}_c(v_j,v_j^t) \cup \text{ADDED}_c(v_j,v_j^t) \) with \( c_1 \neq c_2 \). For these \( (c_1, c_2) \) we consider all method pairs \( (m_r, m_a) \in \text{REMOVED}_m(v_i,v_i^t,c) \times \text{ADDED}_m(v_j,v_j^t,c) \) with \( \text{signature}(m_r) = \text{signature}(m_a) \).

This means that the method \( m_r \) in the class \( c_1 \) has been moved to the class \( c_2 \).

### 2.5. Incorporating the Class Hierarchy

By taking the class hierarchy into account, we can determine if moved methods have actually been pushed down, pulled up, or none of both. In the following we describe our approach to extract the class hierarchy for a transaction only on source level (thus, without compiling and analyzing the byte code).

First, we compute the set \( P \) of **project classes** in the configuration created by the transaction. Thus, we parse all versions of JAVA files in the configuration for their classes. Next, we parse all JAR files in the configuration. The classes contained in these JAR files form the set \( E \) of **classes in external libraries**. Additionally, we know the set \( J \) of **classes in the standard JAVA API**.

Next, we determine for each class \( s \in P \) the set \( S \) of the direct superclass of \( s \) and of the interfaces which \( s \) implements (based on the extends, resp. implements keyword, the import declarations, and the set \( P \cup E \cup J \)).

Let \( C \) be a set of classes and \( \mathcal{P}(C) \) denote the set of superclasses of \( C \). The method \( m \) has been **pulled up** in the class hierarchy\(^1\).

The **Push Down Method** refactoring can be recognized analogously by switching \( s \) and \( t \).

To gather further information about classes related to refactored classes, we also extract the aggregated classes of each class. Therefore, we look at the types of the fields – these are the global variables of the class – and retrieve the corresponding fully-qualified classes using again the import declarations, and the set \( P \cup E \cup J \).

### 2.6. Detecting Impure 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 artifact. In this section we explain to which degree we are able to detect such **impure refactorings**.

**Changes in a Moved Class.** Analyzing a transaction for **Move Class** refactorings we test if a class that has been added to a package contains the same methods as a class that has been removed from another package. Therefore, we

---

\(^1\) We define \( f^1 = f \) and \( f^n = f \circ f^{n-1} \).
cannot detect class movements if one or more of the methods have been locally refactored, new methods have been added, or methods have been removed.

**Multiple Refactorings on the Same Location.** As **Hide Parameter** is the only refactoring where the visibility of a method is tested, this refactoring can be detected together with all other method refactorings on the same method. As the **Rename Method** refactoring requires the parameters of the “old” and the “new” method to be equal, we cannot detect these refactorings when a method has been renamed and at the same time parameters have been added or removed.

**Refactorings on Methods with Changes in Method Body.** Our condition to find rename refactorings requires the new method to have the same body as the old one. This means, we cannot find renamed methods when at the same time something within the method has been changed.

### 2.7. Ambiguous Refactorings

Unfortunately, it is not always possible to identify unambiguously all refactorings. Look at the following example: a class contains the methods \( m(t_1, t_2) : t_r \) and \( m(t_2, t_3) : t_r \), and after a new transaction it contains instead of these two the new method \( m(t_1, t_2, t_3) : t_r \). Now it is undecidable if this is an **Add Parameter** refactoring from \( m(t_1, t_2) : t_r \) by adding \( t_3 \) or from \( m(t_2, t_3) : t_r \) by adding \( t_1 \). In such cases, we take all matching refactorings into account.

### 2.8. Relating Refactorings to Configurations

In our visualization we show not only the classes that have been changed and may contain refactorings, but also the other classes that have been part of the project when the developer has performed the check-in to the repository. Thus, the visualization has to take the whole configuration that has been active after a transaction into account. We compute for each transaction \( t \) the configuration active after it.

Let \( V \) be the set of all versions (as before), \( V_r \subseteq V \) be the set of the versions contained in a repository, \( T \) be the set of all transactions, and \( T_r \subseteq T \) be the set of transactions that exist in the repository. Moreover, let \( \text{filename:} V \to \text{String} \) be the function that returns the fully qualified filename of a version, \( \text{time:} V \to \text{Timestamp} \) be the function that returns the timestamp of a version, and \( \text{starttime:} T \to \text{Timestamp} \) be the function that returns the timestamp of the start of a transaction.

Then, we compute the set of files affected by a transaction \( t \in T_r \): \( \text{files}(t) = \bigcup_{v \in \text{files}(t)} \text{filename}(v) \).

The configuration \( \text{conf}(t) \) active after transaction \( t \in T_r \) consists of the versions performed in the transaction, as well as the latest version with a timestamp before the start time of the transaction for the files that are not affected by \( t \):

\[
\text{conf}(t) = t \cup \{ v \in V : \text{filename}(v) \notin \text{files}(t) \\
\land \text{time}(v) < \text{starttime}(t) \\
\land \{ \exists v' \in V_r : \text{filename}(v') = \text{filename}(v) \\
\land \text{time}(v) < \text{time}(v') < \text{starttime}(t) \})
\]

### 3. Visualizing Refactorings

In this section we present our technique to visualize refactorings. We visualize on the level of classes and methods, and not on the level of source code. Classes are represented by boxes containing the class name (including the package name) and a list of the signatures of the methods in this class. Interfaces are similarly represented. Our **REFVIS** tool provides two different views on the classes of a configuration: the class hierarchy view and the package view. In the hierarchical view, the relations between classes are represented by edges with arrows conforming to the standard UML notation. Additionally, the different types of relations are color coded: extends edges are red, implements edges blue, and aggregates edges black.

But first, we show how to pick the interesting refactorings out of the whole bunch retrieved by the knowledge discovery in the archive.

#### 3.1. Filtering and Context

Usually a programmer is not interested in all refactorings, but only in those which are involved in the part of the project he is working on, or those out of a certain range of configurations. To achieve this goal **REFVIS** offers a large variety of filter possibilities that can be individually enabled or disabled:

- filter on the level of configurations:
  - consider only configurations that contain (particular) refactorings and/or are out of a certain period of time,
- filter on the level of classes:
  - consider only classes that contain (particular) refactorings and/or are contained in particular packages and/or contain particular methods,
- filter on the level of methods:
  - consider only methods involved in (particular) refactorings.

Just as important as filtering is to build the appropriate context of refactorings. Sometimes it is useful to integrate refactored classes in their inheritance tree. The visualization of **Push Down Method** respectively **Pull Up Method** refactorings, for example, should not only show the target class of the refactoring, but the source class as well. Therefore **REFVIS** offers the possibility to extend the set of filtered classes to all classes that are contained in an inheritance tree of a filtered class. Furthermore, it is also possible
to integrate classes that aggregate or are aggregated by filtered classes.

3.2. Visualizing Refactored Classes

To visualize refactorings contained in a class we use two different techniques. First, we use the following color coding scheme to mark the type of refactoring:

- a shade of red for moved classes/methods (red moved class or method, orange pulled up method, dark orange pushed down method),
- a shade of green for changed parameters (dark green removed parameter, light green added parameter),
- a light gray for hidden methods,
- magenta for renamed methods,
- black for methods without a refactoring.

Second, we use tooltips that show a short description of all refactorings in a class if the mouse pointer is moved over the concerned class.

3.3. Interactive Views

REFVIS provides two different views: the hierarchical view showing the class hierarchy and the class aggregations, and the package view. The package view is appropriate if we want to know which packages are affected by refactorings in a configuration, whereas the hierarchical view provides more structural correlation of refactorings (Figure 3, for example, shows that the parameter lists of several subclasses of a class are extended by a parameter of type \texttt{Component} resp. \texttt{View}). The layout of the package view is mostly more compact than the layout of the hierarchical view.

The output of REFVIS is a Scalable Vector Graphics (SVG) file that includes a script that offers free zooming and scrolling features. These provide the user with the possibility to get an overview over all classes affected by refactoring (by zoom out), as well as to have a closer look at a certain class (by zoom in).

4. Case Studies

In this section we apply the presented refactoring discovery and visualization techniques to two open source projects: JEDIT and TOMCAT.

Table 1 shows the number of recognized refactorings in both projects, broken down to the kind of refactoring.

Figure 1 shows the distribution of the detected refactorings in the transactions\(^2\) of JEDIT respectively TOMCAT.

This information is particularly useful to quickly identify those transactions in which we detected many refactorings. For JEDIT Transaction 1836 is the hot spot with a total of 71 refactorings. This example shows that in one single transaction the structure of the system can change a lot.

In the following sections we go into detail for some refactorings that refer to interesting changes in the program structure or behavior. These refactorings should be regarded by a developer who was, for example, absent for some time, to regain an understanding of the system.

Since printed paper is quite inappropriate to show interactive, zoomable, and scrollable visualizations, the SVG files corresponding to the following figures are available at http://rw4.cs.uni-sb.de/~goerg/IWPC05/.

### Making an Abstract Class Concrete

In Figure 2 we see two classes of Configuration 934 of TOMCAT. The most interesting refactorings are the method movements to Response. All these methods have originally been part of \texttt{ResponseImpl}. A look at the source code shows, that actually the concrete class \texttt{ResponseImpl} has been deleted and the former abstract class \texttt{Response} has become concrete. Therefore, the methods of \texttt{ResponseImpl} have been moved to \texttt{Response}.

### The Effects of Local Refactorings

Figure 3 displays Transaction 1232 of JEDIT. This transaction demonstrates very nicely that a developer must take attention to sub- and superclasses when doing local refactorings on methods of a class. For example, if he misses to update a class that inherits from the refactored class, the program will either not compile or behave incorrectly be-

\(^2\) The first 838 transactions of JEDIT respectively the first 754 transaction of TOMCAT do not contain refactorings, because they reflect the initial imports of the files into the repository.
Figure 1. Distribution of refactorings.

Figure 2. Making an abstract class concrete.

Figure 4. Trying and undoing a refactoring.

Renaming Methods to Change Event Handling

Figure 5 shows a part of Configuration 1123 of JEDIT. In the corresponding transaction the mouseClicked() method of the inner MouseHandler classes of

3 We actually found such a buggy refactoring in Transaction 876 of JEDIT: A parameter has been added to a method in an interface but not all implementing classes have been updated. Thus, the resulting configuration did not compile. The visualization of this refactoring can be found on our web site.
both EnhancedMenuItem and EnhancedCheckBoxMenuItem have been renamed to mouseReleased(), i.e. the body of these methods have not been changed. This means the methods still do the same. But, as those mouse adapters inherit from java.awt.event.MouseAdapter, they are now triggered at a different event.

Adding an Intermediate Class to the Hierarchy

Figure 6 shows that in Transaction 3926 of TOMCAT some methods have been pulled up from SimpleRealm and JDBCRealm to the direct superclass RealmBase which is a subclass of BaseInterceptor. In fact, the class RealmBase has been added to the project in this transaction. The obvious reason is that the developers wanted to create one class that is a superclass of all Realm classes but not other classes. As the single Realm classes extended BaseInterceptor before, the new RealmBase class has now to do so.

Converting an Abstract Class to an Interface

In Transaction 822 of TOMCAT several methods have been pushed down from the interface Tag to the implementing class TagSupport and its sub-class BodyTagSupport (see Figure 7). In fact, Tag has been an abstract class before and has been converted to an interface but not all methods of the abstract class have been adopted in the interface. Thus, the developer has created new classes implementing this interface: BodyTagSupport and TagSupport where the first of them extends the latter. The methods of the former abstract class that have not been adopted in the new interface have been moved to these classes as needed and have been made concrete.

Specializing Overloaded Methods

In Figure 8 which shows a part of TOMCAT Configuration 1087 we see that several methods in two
classes have been renamed in the corresponding transaction. Note, that previously these methods all had the name `add`. Thus, `add` had been overloaded and had been used to add users, groups, as well as roles. The renaming breaks this overloading into specialized methods which make it more clear that different kinds of addition operations are performed in each case. As a side note, bear in mind, that the class `FileRealmGroup` aggregates the class `FileRealmDatabase`, which makes it obvious that for the consistency of the method namings the refactoring should be and has been performed to both classes at the same time.

Splitting Classes

In Transaction 3057 of TOMCAT a lot of move method refactorings from the class `DefaultMatcher` have been
moved to the class Matcher and its subclasses have been performed. We checked out the source code of the corresponding configuration and its predecessor configuration and found that actually the class DefaultMatcher has been removed and its functionality has been distributed over the several Match classes. Moreover, functionality that is needed by all these new classes commonly has been moved to the new class Matcher which has become a superclass of all Match classes.

Refactorings on Multiple Packages

In Transaction 1942 of TOMCAT a parameter of type Request has been added to methods of seven different classes. Using the package layout visualization we can see that these classes are located in two different packages, namely org.apache.tomcat.core and org.apache.tomcat.session.

5. Related Work

Refactorings and Program Comprehension. General information on refactoring are presented in Fowler’s book [9]. Demeyer et al. presented some metrics-based heuristics [19] 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 and did not visualize the discovered refactorings.

But exploring refactorings is only one method to gain comprehension on how a program is evolving. Xing and Stroulia [20] categorized classes to evolution profiles dependent on the differences of the UML diagrams of subsequent revisions of a class. In their taxonomy a class belongs to categories such as active, shrinking, and volatile. Moreover, they use data mining to unveil co-evolution patterns between these categories.

Knowledge Discovery in CVS Repositories. Data extraction from CVS repositories is already covered by several tools, e.g. SOFTCHANGE [11, 12] and BLOOP [18, 7]. Ahmed Hassan and Richard Holt discussed the design issues of such tools in [14]. In our previous work [22] we presented an approach how to mirror a CVS repository into a relational database and implemented it in the APPEL [5] framework.

Several researchers addressed the mining of fine-grained changes. Harald Gall et al. [10] and James Bieman et al. [2] analyzed relations between classes while Zimmermann et al. mined for binary [21] and n-ary [23] association rules on function-level.

Visualizing Software Evolution. There has been some work on visualizing the evolution of software so far. CODECRAWLER [16] provides many visualizations to illustrate changes over time, among these CLASSBLUEPRINT [17] that shows the inheritance between and the structure of classes. SEESOFT [8] introduced space-filling visualizations for metrics related to lines of code. Using a slider the user can control an animation over the evolution of the system.


6. Conclusions

In this paper we introduced an approach to search for refactorings performed in the lifetime of the project in software archives. This approach allows to detect refactorings concerning classes and methods in versions of object-oriented source code files and relates Move Method refactorings to the class hierarchy.

Then, we presented a technique to visualize these refactorings. This visualization includes

- color coded representations of different kinds of refactorings and an optional description which is available by tooltips,
- free zoomable and scrollable layouts of classes affected by refactorings - this provides an overview as well as a close-up view,
- two different views on the classes: the hierarchical view showing the class inheritance and the package view.

Finally, our case studies on the open source projects JEDIT and TOMCAT show that

- our visualization technique is appropriate to illustrate the structural changes of a software system that have been done within a particular transaction,
- the detected refactorings refer to interesting and complex structure changes that have to be regarded to understand the system. Being aware of these changes is very valuable to developers that have been absent from a project for some time and have to get used to it again.

7. Future Work

In this section we shortly describe our future plans on improving the described techniques.

Presently, we only find refactorings which affect the class hierarchy or the structure of classes and methods. However, there are many different kinds of refactorings that
deal, for example, with error handling or clearness of the code, and require taking the source code of methods respectively classes, and finer grained artifacts into account. Moreover, applying program analysis techniques can yield further data like call graphs. This would enable to find other kinds of refactorings, e.g. Remove Middle Man.

At the moment we often fail to detect impure refactorings. We hope to improve the recall of our methods by using clone detection algorithms [1] to identify moved code.

As to the visualization, we currently only show which refactorings have been performed – but not exactly how. Thus, we aim to present on demand exactly those source code fragments that contributed to a refactoring. Moreover, we currently show only the configuration that appears after the refactoring has happened – but not the transition itself. For the future we are planning to use our graph animation techniques [6, 13] to do so.

Acknowledgments. Michael Burch, Stephan Diehl, and Sandra Gessner gave helpful comments on earlier revisions of this paper. Mathias Pohl implemented the package view and Stephan Zimmer assisted in generating the SVG files.

References