Fine-Grained Analysis of Change Couplings

Beat Fluri, Harald C. Gall, and Martin Pinzger
s.e.a.l. – software evolution and architecture lab
Department of Informatics
University of Zurich, Switzerland
{fluri,gall,pinzger}@ifi.unizh.ch

Abstract

In software evolution analysis, many approaches analyze release history data available through versioning systems. The recent investigations of CVS data have shown that commonly committed files highlight their change couplings. However, CVS stores modifications on the basis of text but does not track structural changes, such as the insertion, removing, or modification of methods or classes. A detailed analysis whether change couplings are caused by source code couplings or by other textual modifications, such as updates in license terms, is not performed by current approaches.

The focus of this paper is on adding structural change information to existing release history data. We present an approach that uses the structure compare services shipped with the Eclipse IDE to obtain the corresponding fine-grained changes between two subsequent versions of any Java class. This information supports filtering those change couplings which result from structural changes. So we can distill the causes for change couplings along releases and filter out those that are structurally relevant. The first validation of our approach with a medium-sized open source software system showed that a reasonable amount of change couplings are not caused by source code changes.

1. Introduction

Version control systems such as CVS contain a huge amount of historical information of a software system collected during its evolution. Recent investigations of CVS data, such as performed by Fischer *et al.* [3, 6] or Zimmermann *et al.* [12] have shown that commonly committed files highlight their change couplings. Change couplings present important facts that are used to reason about the evolution of software systems.

Current approaches rely on the following definition of change coupling: Two files exhibit a change coupling if they are commonly committed, *i.e.,* at the same time, by the same author, and with the same modification description. Following these definition change couplings are computed from the modification reports stored in CVS repository.

However, CVS tracks changes only on a text basis. Detailed information of which class, method, or field was affected by a change is not stored. The effect is that change couplings include also couplings that are due to non-structural changes, such as changes in the license term. Such non-structural change couplings have to be filtered because they indicate false couplings between files. For instance, predicting source code changes is simpler with filtered change couplings. In [11] Ying *et al.* present an approach to predict source code changes. For each change coupling they calculate the source code couplings between the corresponding files. Filtering non-structural change couplings has the advantage, that only files which were structurally changed must be further investigated for source code couplings. It is not necessary to calculate source code couplings of all change coupled files. Furthermore, the possibility to investigate only frequent change couplings between a set of files reduces also the amount of change couplings to be investigated [6].

Frequent change couplings indicate a strong coupling between the corresponding modules, sub-modules, or files as well as possible shortcomings in the design of a software system [5]. Filtering non-structural change couplings reduces the amount of misleading change couplings and, therefore, reduces the effort to investigate all change couplings.

In this paper we focus on both issues and present an approach that facilitates the filtering of change couplings according to structural changes and the frequency of change couplings. For the computation of the structural changes between subsequent releases of Java source files we use the source comparison facilities of the Eclipse IDE. They retrieve fine-grained change information, such as the insertion, deletion, and modification of methods and classes based on the abstract syntax tree. With the CVS interface

Proceedings of the 2005 Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)
0-7695-2292-0/05 $20.00 © 2005 IEEE

of Eclipse we have automated the structural change extraction of an Eclipse project. The change coupling information stored in our release history database (RHDB) are used to cluster transactions of change coupling groups. The structural change information is then used to filter those change couplings which result due to structural changes. The first validation of our approach showed that a reasonable amount of change couplings are not caused by source code changes.

The remainder of this paper is organized as follows: Section 2 gives a brief overview of related work in the area of software evolution analysis and change extraction. In Section 3 we describe the process of analyzing change coupling groups. Section 4 presents a validation of our approach on the compare plugin of Eclipse. We conclude the paper in Section 5 and indicate areas for future research.

2. Related Work

In previous work Gall *et al.* [5] analyzed the history of changes in software systems to detect the hidden dependencies between modules. Fischer *et al.* [4] extended the concept of logical coupling and defined a filtering mechanism and a data scheme for such an integration. The data scheme is the initial version of the Release History Database (RHDB) that we adapted for the ArchView approach. Fischer *et al.* [3] analyzed the evolution relation to bug reports to track the hidden dependencies between system features. By instrumenting the code the authors showed how features are scattered over the project tree and how features are logically coupled over releases. An extension of this approach with a number of specific visualization techniques is described in [2]. This approach allows an engineer to uncover hidden dependencies among different features over many releases.

Based on CVS data Krajewski *et al.* [6] discovered logical (*i.e.,* change) couplings: developers checking in and out files within certain periods of time and the relationship of these files discovered dependencies that are difficult to detect by other means and pointed to several bad code smells by means of visualizations using JGraph.

Zimmermann *et al.* presented a fine-grained analysis approach for CVS data that considered all kinds of entities starting from the statement level [12, 13]. Their ROSE tool identifies common changes between syntactical entities rather than files or modules using textual differencing algorithms.

Most mechanisms of finding changes between two existing source code files use textual differencing algorithms. The most popular differencing algorithm tool is certainly the UNIX `diff`. It is based on a longest common subsequence algorithm (LCS). Textual differencing algorithms are limited to a line-level granularity. This implies several limitation. First, the changed source code entity, such as a method, is not determined. Second, if a source code entity moves within the text file, but does not change syntactically, the textual differencing algorithm states for a change anyway. These limitations can be addressed with comparing abstract syntax trees (AST). This technique is primarily used to improve merging [9]. Full AST differencing techniques are too fine-grained for our filter approach, therefore we use the light-weight AST differencing algorithm shipped with Eclipse.

Today, the term *configuration management system* is mostly reduced to CVS. But there are systems offering the possibility of fine-grained change analysis. For instance, Magnusson *et al.* developed a framework for fine-grained version control. Instead of only controlling files, they also put blocks, classes, procedures, and functions under version control [7]. Thus using this framework, fine-grained change information is available. Such systems are source code management systems and therefore not directly usable for fine-grained change analysis.

In [8] Maletic and Collard present an approach to analyze source code differences. The approach is called *meta-differencing* and uses an XML representation of the source code, *srcML*. With that representation the authors can embed meta-information such as hyperlinks into the XML representation. They use textual differences to locate the changed source code entities in the srcML document. Our structure compare relies on the parsing capabilities of Eclipse and gets rid of pure textual comparisons.

The *UMLDiff* approach presented by Xing and Stroulia detects structural modification on a class basis [10]. UMLDiff takes as input two UML class models represented in XMI and calculates their changes. However, UMLDiff does not track if the body of a method has changed, because an UML class model does not contain implementation details.

3. Analyzing Change Couplings

The Eclipse IDE [1] provides several plugins to support software developers in their daily work. Among other things, there are plugins to access CVS repositories and compare source code files. We exploit these interfaces to checkout the revisions of a file, perform structural comparisons between subsequent revisions, and store the extracted information in a database for further analysis. In this section, we describe this process in more detail.

3.1. The Process of Analyzing Change Couplings

The information flow and process steps of our change coupling analysis approach is depicted in Figure 1. The first step is concerned with retrieving the modification reports from the CVS repository and building the release his-

Proceedings of the 2005 Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)
0-7695-2292-0/05 $20.00 © 2005 IEEE

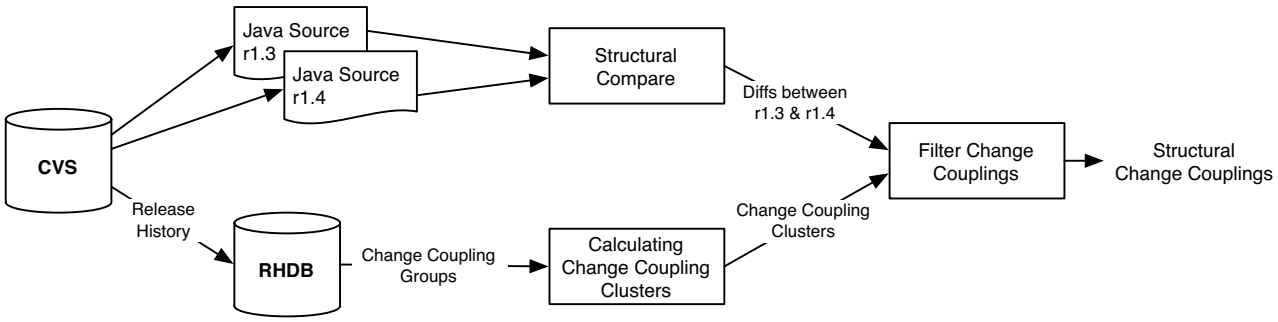

Figure 1. Information flow and process steps

tory database (RHDB). The building process includes the computation of the change coupling relationships between source code files [4]. Based on this information the change coupling clusters are calculated. On the upper chain, subsequent revisions of source files are retrieved from the CVS repository and structurally compared. Both, the structural differences and the change coupling clusters are input to the change coupling filter that outputs those change couplings that stem from structural changes. The process steps are described in more detail in the following sections.

3.2. Structure Compare between Java Files

When a changed file is committed, the CVS repository stores a modification report and the number of added and/or deleted lines in the two subsequent revisions of the file. This information does not reveal much about the changes of the source code in the file. It is possible to get finer-grained modification information with the CVS *diff*. It prints out which regions in the files were added, changed, or deleted. Figure 2 shows an example where four lines were reduced to two lines, thus the four lines were changed. But, using solely this information for determining in which method the changed lines resist is still not a trivial task.

```
1967,1970c1964,1965
<     if (d != null && !d.fIsToken) {
<         updateDiffBackground(oldDiff);
<         updateDiffBackground(fCurrentDiff);
<     }
---
>     updateDiffBackground(oldDiff);
>     updateDiffBackground(fCurrentDiff);
```

Figure 2. Extract of a CVS diff

A potential solution to this problem is to represent the source code in a more organized way to facilitate comparison. For that we distinguish between two techniques: 1) abstract syntax tree (AST) comparison; 2) structure comparison. The first technique compares the ASTs of the two versions of a source file and derives the differences on the abstract syntax level. For instance, changed conditions in an if-statement can be detected. Concerning the fine-grained changes this is the desired approach but its implementation is not trivial and to our knowledge has not been realized yet.

The second technique determines changes on the level of source code entities, such as methods, fields, or classes. Although it misses detailed changes, it is sufficient to answer the question whether a Java source code entity has experienced changes in two subsequent revisions. In particular, we are interested in changes of source code entities such as class, interface, method, constructor, field, and initializer.

The compare plugin of Eclipse[1] distinguishes between two kinds of comparisons: 1) differences between hierarchically structured data, such as source code; 2) differences between sequences of comparable entities, such as text lines or tokens. For the structure comparison of Java classes, Eclipse structures the corresponding Java source files hierarchically into the entities import declaration, class, interface, method, constructor, and field. It then compares the files using these structures and displays the result in a tree layout. In addition, for each structural difference the corresponding text differences are displayed. Figure 3 shows the screenshot of the Java structure compare with the previous example (Fig. 2). In the topmost panel of Figure 3 we see the method that was changed, `setCurrentDiff(Diff,boolean)`. The text differences between the subsequent revisions of the method are depicted in the middle panel. The newer revision of the method is depicted on the left hand side.

Basically, we are interested in the structural change information of the topmost panel. Since Eclipse has well defined and described APIs of its plugins, we are able to make use of the comparison interfaces. In general, to compare two entities the following two steps have to be performed:

[1] `org.eclipse.compare`

Proceedings of the 2005 Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)
0-7695-2292-0/05 $20.00 © 2005 IEEE

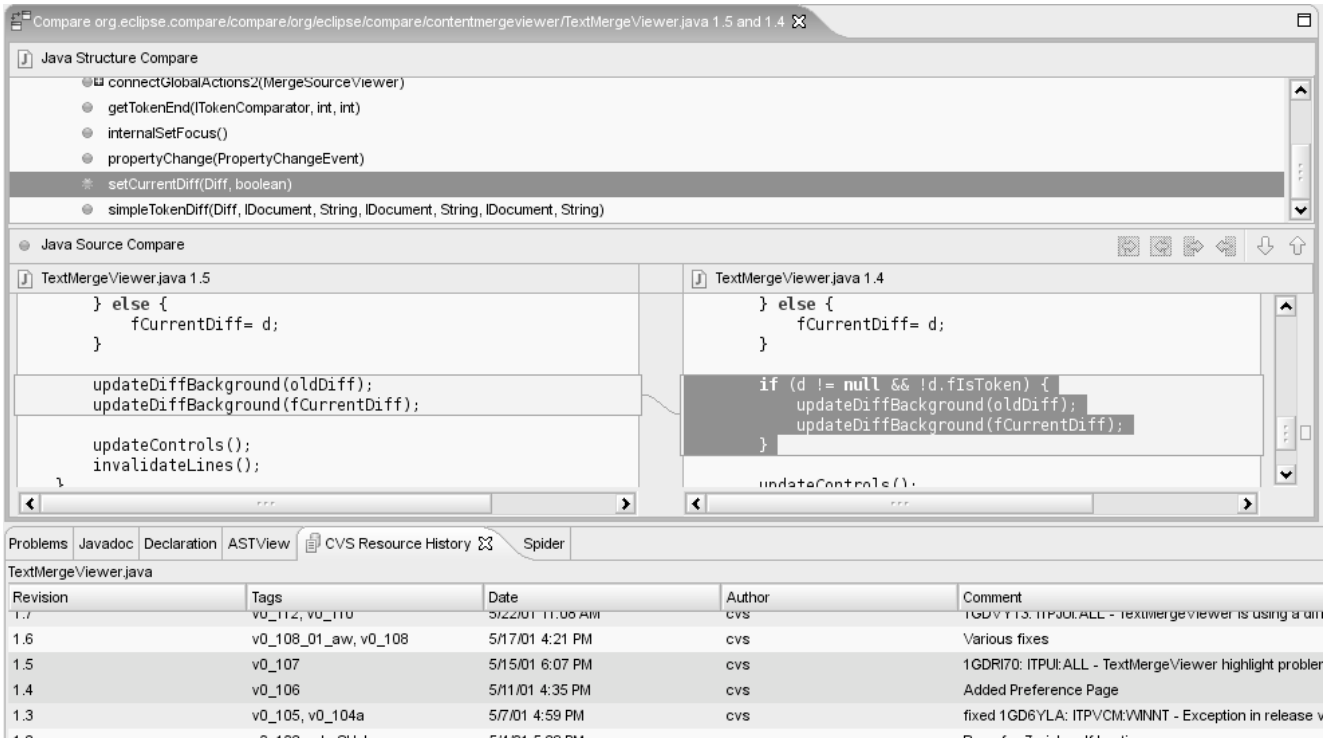

Figure 3. Structure and source code compare in Eclipse

1) the entities have to be converted into a format suitable for the differencing algorithm, *i.e.,* either into hierarchically structured data or certain sequences of characters; 2) the conversions are fed into the differencing algorithm to get the differences. The compare plugin of Eclipse provides implementations for the two algorithms: `Differencer` for hierarchically structured data; `RangeDifferencer` for sequences of comparable entities.

In case of structure comparison of two Java source files, the *Java Development Tools*[2] provides a class, `JavaStructureCreator`, for structuring Java source code into the entities: import declarations, classes, interfaces, methods, constructors, and fields. Then, using the class `Differencer`, the differences between the two structures are calculated. These differences are stored in a tree of `DiffNode` instances containing the kind of change, *i.e.,* changed (C), added (A) or deleted (D), as well as an instance of `JavaNode` representing the compared source code entity.

In addition, we are interested in the number of added and/or removed text lines of changed entities. For that, the regions corresponding to the changed entities are compared by using a line-by-line comparison. The compare plugin provides a line sequence generator for text documents: `DocLineComparator`. Since the Java node of the changed entity keeps track of its location in the source file, we initialize the generator only with the entity region to compare. Using the class `RangeDifferencer`, the differences between the two regions are stored in an array of `RangeDifference` instances.

Each identified structural difference is saved in our RHDB with the information: complete file path including the CVS module name, new revision number, change type, number of added lines, number of removed lines, entity type, and entity name.

Table 1 shows the stored information for the compare example in Figure 2, and 3 respectively. With the information in this example, we find out that the method `setCurrentDiff(Diff, boolean)` in the Java source file `TextMergeViewer.java` has changed from revision 1.4 to 1.5 and two lines of code were deleted.

3.3. Automated Structural Change Extraction

CVS allows us to checkout any revision of a file in a particular repository and CVS integration is well supported in Eclipse. The CVS plugin of Eclipse[3] provides various interfaces to use common CVS commands. We make use of this support to automate the structural change extraction process.

[2] `org.eclipse.jdt`

[3] `org.eclipse.team.cvs.core`

File	*Rev*	*changeType*	*+LOC*	*-LOC*	*entityType*	*entityName*
TextMergeViewer.java	1.5	C	0	-2	method	setCurrentDiff(Diff, boolean)

Table 1. Example of a structure change of `TextMergeViewer.java` between revisions 1.4 and 1.5

We developed an Eclipse plugin that performs the process described in Section 3.2 for all subsequent revisions of a Java source file. In particular it performs the following process steps:

1. It parses the CVS log information of the file under investigation to get all revision numbers.
2. Starting at the revision 1.1, it checks-out every revision and its subsequent revision of the file and performs a structure comparison.
3. It stores the comparison results in the RHDB.

With our plugin we can select a single file or a whole project on which the structure comparison is performed.

3.4. Change Coupling Groups

The RHDB approach [4] calculates the change couplings between files in a CVS repository using the relation analysis approach described in [6]. In short, change couplings are traced back to common commits of modified files that refer to a transaction. We define a *change coupling group as the set of files that were committed together*. But, to what extent the files were structurally changed, *i.e.,* are structurally coupled, is not clear. For instance, files which were committed together according to a change in the license terms are not structurally coupled, but contained in the same change coupling group.

In the history of all transactions, a certain change coupling group may occur more than once. With each occurrence of such a change coupling group the probability that modifications to a file of a group propagates to the other coupled files increases. Change coupling groups occurring only once are most likely to be accidental transactions.

Therefore, we focus on the following aspects of change coupling analysis: Which coupling groups recur often and which of them were subject to structural changes.

3.5. Detecting Frequent Change Coupling Groups

A certain change coupling group may be committed frequently in two different ways. Either it is always committed alone and not with other files/groups, or it is a subgroup of another group. A change coupling group g is a change coupling sub-group of h if $g \subseteq h$. The cardinality $|\cdot|$ of a change coupling group is the number of contained files. In our approach we consider the maximal change coupling groups representing one commit (*i.e.,* transaction).

After a commit in CVS a file gets a new revision number. Consequently, a change coupling group gets a set of revision numbers. We define a revision vector as this set of revision numbers. A revision vector also corresponds to a transaction.

Let G be the set of all change coupling groups in the RHDB and $g \in G$. Let g be the change coupling group under investigation. The change coupling cluster P is the collection of all revision vectors of the change coupling group g in G. For the computation of the change coupling cluster, we check for each change coupling group $h \in G$ if g is a change coupling sub-group of h. After that is verified, the corresponding revision vector is added to the change coupling cluster.

This algorithm is in $O(|G|^2)$. Reducing G by exact matches, *i.e.,* if $g = h$ remove h from G, does not reduce the complexity $O(|G|^2)$ in general. Anyway, we have to remove exact occurrences after a pass, because otherwise a change coupling cluster may occur more than once.

We distinguish the occurrences of a particular change coupling group in a change coupling cluster by their revision vectors and represent a change coupling cluster as a matrix. For instance, the files A, B, and C build a change coupling group $g = \{A, B, C\}$. Assume, they were committed together three times. Their revision numbers were for A: 1.3, 1.5, and 1.9; for B: 1.2, 1.7, and 1.8; for C: 1.6, 1.10, and 1.12. The corresponding change coupling cluster is the following matrix:

$$\begin{array}{c} \\ A \\ B \\ C \end{array} \underbrace{\begin{bmatrix} 1.3 & 1.5 & 1.9 \\ 1.2 & 1.7 & 1.8 \\ 1.6 & 1.10 & 1.12 \end{bmatrix}}_{\text{revision vectors } r_1, r_2, r_3}$$

To summarize, a change coupling group consists of a set of files which were committed together more than once in the history of a software system. A change coupling cluster consists of a change coupling group with multiple revision vectors. We store a change coupling cluster in a matrix in which the revision vectors are its columns.

3.6. Attaching Structural Changes to Change Coupling Groups

According to the stored information of each change coupling cluster, we are able to get the structural change for each commit in a change coupling cluster P. Assume:

Proceedings of the 2005 Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)
0-7695-2292-0/05 $20.00 © 2005 IEEE

$$P = \begin{array}{c} \\ A \\ B \\ C \end{array} \begin{bmatrix} 1.3 & 1.5 & 1.9 \\ 1.2 & 1.7 & 1.8 \\ 1.6 & 1.10 & 1.12 \end{bmatrix}$$

- T is the set of all transactions in the change coupling cluster P. $t_i \in T$ is one particular transaction.
- R is the set of all revision vectors in the change coupling cluster P. $r_i \in R$ is the revision vector of t_i:

$$R = \begin{bmatrix} 1.3 & 1.5 & 1.9 \\ 1.2 & 1.7 & 1.8 \\ 1.6 & 1.10 & 1.12 \end{bmatrix} \quad i = 2 : r_2 = \begin{bmatrix} 1.5 \\ 1.7 \\ 1.10 \end{bmatrix}$$

- $g \in G$ is the coupling group represented by the change coupling cluster P: $g = \{A, B, C\}$
- $f_j \in g$ is a file in the change coupling group. Then, $r_{i,j}$ is the revision number of the file f_j after the transaction t_i:

$$i = 2, j = 3 : \quad f_3 = C, r_{2,3} = 1.10$$

- SC is the set of all structural changes which has the corresponding layout as Table 1.

F	*Rev*	*chT*	+	-	*eT*	*eN*
A	1.3	A	0	0	method	foo(byte)
A	1.5	C	3	0	method	foo(byte)
A	1.5	C	1	-4	constr.	A(int)
B	1.2	D	0	0	field	bar
B	1.2	C	5	-2	method	add(C)
B	1.8	A	0	0	field	bar
C	1.6	C	0	0	method	dispose()
C	1.10	C	1	0	constr.	C()
C	1.12	C	0	0	field	frame

Abbreviations: F = File, chT = changeType, +/- = +/- LOC, eT = entityType, eN = entityName, constr. = constructor.

In this example, not every file was structurally changed in all transactions. As we show in Section 4, this issue is quite common.

- $S_{i,j} = \{s_k \in SC, k = 1, ..., |SC| \mid f_j = f_k, r_{i,j} = Rev_k\}$. That is the set of all structural changes of the file f_j from revision $r_{i,j-1}$ to revision $r_{i,j}$:

$$i = 2, j = 1 : f_1 = A, r_{2,1} = 1.5 \Rightarrow$$

$$S_{2,1} = \{s_k \in SC, k = 1, ..., 9 | f_k = A, Rev_k = 1.5\}:$$

F	*Rev*	*chT*	+	-	*eT*	*eN*
A	1.5	C	3	0	method	foo(byte)
A	1.5	C	1	-4	constr.	A(int)

Then, the set of all structural changes for one commit t_i is

$$\Sigma(t_i) = \{j \in \{1, ..., |g|\} \mid S_{i,j}\}$$

For instance: $\Sigma(t_2)$:

F	*Rev*	*chT*	+	-	*eT*	*eN*
A	1.5	C	3	0	method	foo(byte)
A	1.5	C	1	-4	constr.	A(int)
C	1.10	C	1	0	constr.	C()

3.7. Browsing Structural Changes of Change Coupling Clusters

We developed a change coupling cluster browser to get a quick overview of the change coupling groups, change coupling clusters, and their structural changes. Figure 4 shows a screenshot of our plugin. It is divided into four tables. The table on the left hand side contains the change coupling groups represented by *GroupID*. For each change coupling group the number of files is given in the column *# Files*. The last column labeled *Occurrences* states how often the change coupling group was committed. Clicking on a change coupling group, the contained files are listed in the top-middle table and the corresponding revision vectors of the change coupling cluster are shown in the top-right table. Clicking on a revision vector the structural changes of the files at the given revisions are listed in the bottom table. Referring to our example, it lists the changes of the methods in the file `ContentMergeViewer.java`, `StructureDiffViewer.java`, and `DiffTreeViewer.java`.

3.8. Filtering Change Coupling Groups

In this paper we focus on filtering false-positives in change coupling groups, *i.e.,* files which were committed according to non-structural changes. These are changes in license terms and whitespaces between methods. Changes in the comment of a method, *e.g.,* Javadoc, or adding whitespace within a method are treated as structural changes. Structure comparison in Eclipse treats changes equally. We use this to filter out revision vectors of change coupling clusters where the corresponding files were not structurally changed.

In particular, we filter two kinds of non-structural changes: first, revision vectors where none of the corresponding files were changed, *i.e.,* $\Sigma(t_i) = \emptyset$. Second, revision vectors in which at least one of the corresponding files were not changed, *i.e.,*:

$$\exists j : \forall i, k : r_{i,j} \neq Rev_k \qquad j = 1, ..., |g|,\ k = 1, ..., |SC|$$

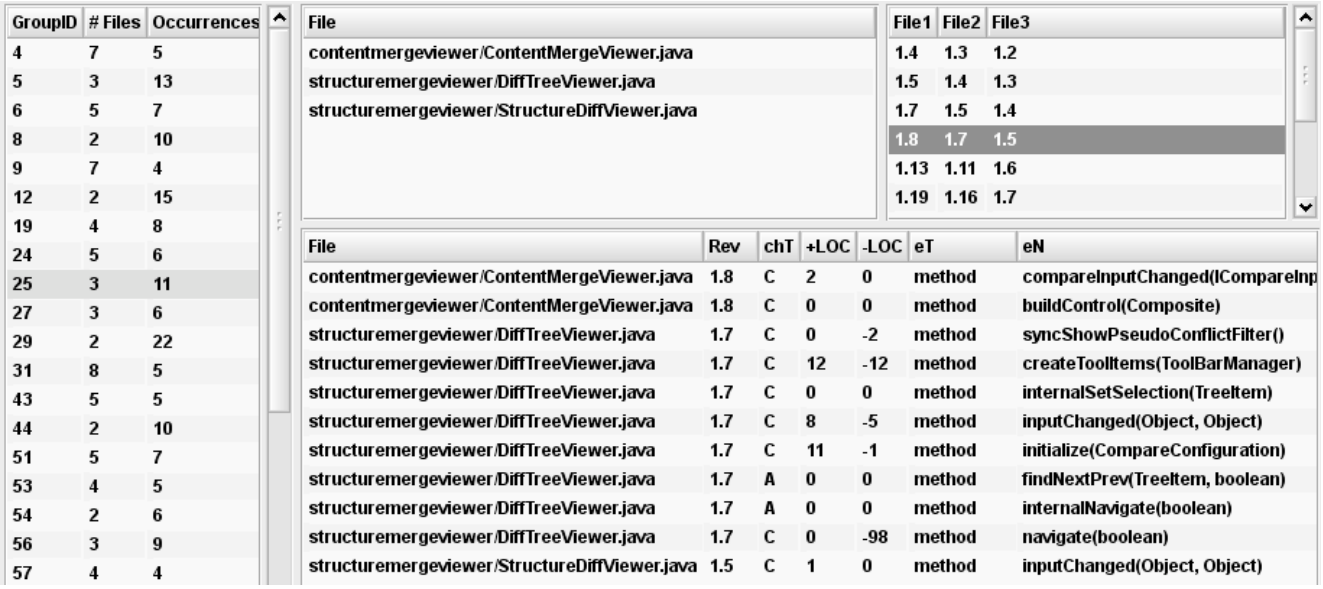

Figure 4. Screenshot of the Change Coupling Cluster Browser

4. Case Study

For the validation of our change coupling analysis approach we conducted a case study with the compare plugin of Eclipse, `org.eclipse.compare`. In particular, we were interested in analyzing the impact of non-structural changes on the amount of change couplings.

The compare plugin consists of seven packages. They provide interfaces to organize data for the differencing algorithm and predefined content viewer to display the differencing results. Three of the packages have the infix `internal`. They contain already implemented data creators used within Eclipse. For instance the class `DocLineComparator` which we use to structure the Java source files into text lines, belongs to one of the *internal* packages.

For the analysis we used the CVS data available on April 18th, 2005. The compare plugin project started on May 2nd, 2001. Table 2 gives an overview of the source size of the plugin at the checkout day.

Metric	*#*
# Packages	7
# Files	117
# Classes	120
# Interfaces	27
# Methods	1022
# Fields	465
# LOC	25914

Table 2. Size-metrics of the compare plugin

4.1. Change Coupling Clusters

We started with the population of the release history database. Table 3 summarizes a few numbers from the RHDB: The plugin contains 322 files; in its history 160 Java files were in the repository. 117 of them were alive at the checkout day. Although the largest five groups contain 53, 36, 28, 18, and 16 files, the average cardinality of the change coupling groups is only 5, *i.e.,* 5 files were committed together on average.

Description	*#*
MRs imported into the RHDB	3086
MRs concerning Java files	1852
Change coupling groups	384
Average # files per groups	5
Change coupling clusters	116
Revision vectors	805
Average # revision vectors in clusters	6

Table 3. Summary of the change coupling cluster extraction

4.2. Structure Compare

We performed a structural change extraction of all Java source files of the compare plugin. In total, 4826 structural changes were found, but interestingly 7 out of the 117 files did not structurally change over the four years. Table 4

shows the distribution of the structural changes into the entities used by structure generator of Eclipse. We extracted addition and deletion of inner classes, because their changes are tracked with the changes of their body. Obviously, the most changes are performed in methods. But, we observed that import declaration changed almost as frequently as fields. Adding import declarations to a Java source file is mostly an impact of another change in the source code, *e.g.,* when a new class is used the first time. Deleting import declarations must not be performed immediately when classes are no longer used. Therefore, we performed the analysis with and without changes of import declarations.

Entity	*#*
Total	4826
Method	3094
Field	853
Import declaration	547
Constructor	298
Inner classes (A/D)	18
Initializer	16

Table 4. Distribution of the structural changes

4.3. Filtering Change Couplings

To analyze which change couplings are not caused by source code changes, change coupling clusters are filtered in two different ways: 1) revision vectors in which none of the corresponding files were structurally changed; 2) revision vectors in which at least one of the corresponding files was not structurally changed. Table 5 and 6 list the results of our approach: 1) 255 out of 805 revision vectors show no structural changes; 2) 416 out of 805 revision vectors show at least one commit of a file without any structural change. This is 31.7%, and 51.7% respectively, of all revision vectors.

If the import declarations are omitted, the values are actually bigger: 1) 262 (32.5%); 2) 438 (54.4%). According to the first filtering, 6 change coupling clusters disappear; the second filtering drops 41 coupling clusters. That are 5%, and 35.3% respectively of all change coupling clusters. Omitting the import declaration changes affects only the first filtering: 42 (36.2%).

4.4. Discussion

The result of the case study is that more than half of all transactions were not caused by structural changes. A possible explanation may be that files are bound to an author, thus strong code ownership is used. The author works all day on his/her files and performs a commit at the end of a working day. Then it is obvious that files that are not structurally coupled are committed together. On the other hand, it may also be possible that the handling of the software licensing of Eclipse changed all the time, and the findings in this case study are unique. In the latter case, a non-Eclipse case study will provide more and in-depth insights.

Effective filtering of change couplings with such impact on the number of change coupling groups will support software evolution analysis in various fields. For instance, the number of false-positives in predicting source changes can be reduced with our approach.

5. Conclusions and Future Work

Versioning systems provide lots of data about a source code base and its modifications, but their major drawback is that the kind and the scope of changes have to be analyzed by an engineer since only lines added and/or deleted are reported textually. Software evolution analyses based on such release history data, therefore, remain on the level of added and deleted lines of code, but do not effectively report on the structure of changes such as classes or methods and their respective changes. Our recent investigations of CVS data have shown that commonly committed files highlight important change couplings. But detailed information whether change couplings are caused by source code couplings or by other textual modifications, such as updates in license terms, are missed by current approaches.

In this paper, we presented an approach that uses the structural compare services shipped with the Eclipse IDE to obtain the fine-grained changes between subsequent versions of Java classes. This information enables to identify those change couplings which result due to structural changes. So we can distill the causes for change couplings along releases and filter out those that are structurally irrelevant. Our approach has been realized as an Eclipse plug-in and has been validated on a medium-sized open source software system. Results so far show that a significant amount of change couplings are not caused by structural source code changes.

The results of this work allows us to perform fine-grained investigation of change couplings and a more detailed combination of release history and source code data for software evolution analysis. We are able to refine our change coupling analysis and its tooling. Furthermore, since Eclipse is not limited to Java we have to generalize our plugin to support other programming languages, such as C/C++.

To validate the results of the case study presented in this paper, we plan to perform further case studies (because the frequent non-structural changes could be project specific). Future work will concentrate on detecting patterns in sequences of structural change. We plan to investigate if patterns which led to complex source code can be

Proceedings of the 2005 Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)
0-7695-2292-0/05 $20.00 © 2005 IEEE

Description	#	%
# Revision vectors (transactions)	805	100%
# Revision vectors without any structural change	255	31.7%
# Revision vectors with at least one non-structural change	416	51.7%
# Omitted change coupling clusters in the first case	6	5%
# Omitted change coupling clusters in the second case	41	35.3%

Table 5. Filtering change couplings with import declarations

Description	#	%
# Revision vectors (transactions)	805	100%
# Revision vectors without any structural change	262	32.5%
# Revision vectors with at least one non-structural change	438	54.4%
# Omitted change coupling clusters in the first case	6	5%
# Omitted change coupling clusters in the second case	42	36.2%

Table 6. Filtering change couplings without import declarations

predicted within other change sequences. For that, finer-grained source code changes extraction must be performed. Another perspective is to investigate the impact of our approach on predicting change propagation.

References

[1] Eclipse. http://www.eclipse.org.

[2] M. Fischer and H. Gall. Visualizing feature evolution of large-scale software based on problem and modification report data. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):385–403, November/December 2004.

[3] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)*, pages 90–99, Victoria, British Columbia, Canada, November 2003. IEEE Computer Society.

[4] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM)*, pages 23–32, Amsterdam, The Netherlands, September 2003. IEEE Computer Society.

[5] H. Gall, K. Hayek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 190 – 198, Bethesda, Maryland, USA, November 1998. IEEE Computer Society.

[6] H. Gall, M. Jazayeri, and J. Krajewski. Cvs release history data for detecting logical couplings. In *Proceedings of the 6th International Workshop on Principles of Software Engineering (IWPSE)*, pages 13–23, Helsinki, Finland, September 2003. IEEE Computer Society.

[7] B. Magnusson, U. Asklund, and S. Minör. Fine-grained revision control for collaborative software development. In *Proceedings of the ACM SIGSOFT symposium on Foundations of Software Engineering*, pages 33–41, Los Angeles, CA, USA, 1993. ACM Press.

[8] J. I. Maletic and M. L. Collard. Supporting source code difference analysis. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM)*, pages 210 – 219, Chicago, Illinois, USA, September 2004. IEEE Computer Society.

[9] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, May 2002.

[10] Z. Xing and E. Stoulia. Understanding class evolution in object-oriented software. In *Proceedings of the 12th International Workshop on Program Comprehension (IWPC)*, pages 34–43, Bari, Italy, June 2004. IEEE Computer Society.

[11] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transaction on Software Engineering*, 30(9):574–586, September 2004.

[12] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE)*, pages 73–83, Helsinki, Finland, September 2003. IEEE Computer Society.

[13] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 563–572, Edingburgh, Scotland, UK, May 2004. IEEE Computer Society.