Supporting Empirical Studies by Non-Intrusive Collection and Visualization of Fine-Grained Revision History

Jacky Chan, Alan Chu and Elisa Baniassad
Programming Practices Laboratory
Department of Computer Science and Engineering
The Chinese University of Hong Kong
{jacky, achu, elisa}@cse.cuhk.edu.hk

ABSTRACT

This paper presents a code-revision history collection and visualization Eclipse plugin for use in empirical studies of programmers. The revision history is collected non-intrusively, and does not depend on CVS-checkin. The visualizations allow for detailed viewing of the code changes for an individual file, and also for an overview of the alterations to a project. In this paper we describe the plugin, and also give examples of our own uses for the plugin for empirical work.

1. INTRODUCTION

When developers are creating and modifying a code base, they carry out many gradual and incremental code change activities. Studying these activities for the sake of finding patterns and analyzing their nature requires non-intrusive monitoring of programmer behavior, intense scrutiny of programmer actions, and high-level visualizations of their patterns of activity. There are, however, currently no publicly available tools of which we are aware, for monitoring and visualizing developers' fine-grained code changes with enough detail to allow in-depth analysis.

In order to support our own empirical work, we developed an Eclipse plugin for non-intrusive collection and visualizations of revision histories. The visualizations need to serve three purposes:

1. depict a high-level view of the code changes made during software evolution to allow investigators to identify phases of development
2. capture language independent code changes
3. capture changes to the code that are not checked in explicitly by the programmer

While the plugin is tailored for our specific empirical questions, we believe that they have broader applicability for empirical studies of programmers.

The plugin consists of three components: an export of the local history maintained by Eclipse, a visualization that shows a high-level view of the history of an individual file, and a second visualization that shows an overview of the project's revision history.

In this paper we present the plugin's architecture, implementation, and usage, and discuss how each component of it has, and might be further used for empirical studies.

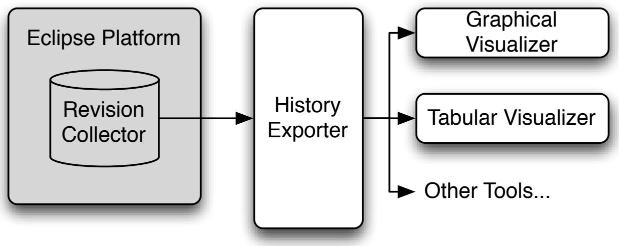


Figure 1: The Plugin Architecture

2. PLUGIN ARCHITECTURE

The architecture of our plugin is depicted in Figure 1. It consists of three components: a history exporter, a graphical visualizer and a tabular visualizer. Each of these depend on revision collection within the current version of Eclipse.

Revision Collector: Using particular Eclipse settings, revisions are captured by this component each time a programmer saves or compiles his/her code. The revisions are then stored into a local repository.

History Exporter: This component retrospectively extracts the Eclipse history of fine-grained language-insensitive code revisions from the local repository into a collection of XML files. These files can then be used for analysis by external tools.

Visualizers (graphical and tabular): These compnents provide visualizations of the revision histories. The first provides a history flow, and depicts fine-grained code changes as colored bars. The second is a tabular view, which provides a revision chart, showing the files that have been changed during revisions. Each of these summarizes developers' behavior, and thus facilitate the identification of developer patterns of activity.

2.1 Revision Collector

Code revisions are collected by the Eclipse built-in function "Local History" (available since Eclipse 3.0), which automatically stores each revision for the sake of undo and backup. Auto-save should be disabled so that revisions are punctuated by developer-intended saves or compiles.

To configure Eclipse 3.0+ to capture the correct information, the following settings must be applied[1]:

```
1. In Eclipse, select menu Window > Preferences...;
2. Select Workbench > Local History (for 3.0) or
   General > Workspace > Local History (for 3.1+);
3. Increase ``Days to keep files'' to the expected
   duration of the software project (e.g. 365);
4. Set ``Maximum entries per file'' to 10000;
5. Set ``Maximum file size (MB)'' to 100;
6. Select Workbench (for 3.0) or
   General > Workspace (for 3.1+);
7. Check ``Perform build automatically on resource
   modification'' (for 3.0) or
   ``Build automatically'' (for 3.1+);
8. Check ``Save all modified resources automatically
   prior to manual build'' (for 3.0) or
   ``Save automatically before build'' (for 3.1+);
9. Set ``Workspace save interval (in minutes)''
   to 9999;
10.Apply all the settings.
```

The Eclipse repository mechanism is implemented as a Memento [3] pattern, which stores the internal states of an object for later restoration. The Originator interface is the `IFile` interface; the Memento interface is the `IFileState` interface, each found in the package `org.eclipse.core.resources`. After the above settings have been applied, the full revision history can be retrieved by calling the `getHistory()` method defined in the `IFile` interface.

After these settings have been applied, the Eclipse platform is able to collect local history for each source file, and the plugin we developed can be installed at any moment to facilitate the use of the revision history.

Another approach might have been to have an automatic CVS-check-in each time a file is saved or compiled, however, this would have resulted in revision-loss each time a file name is changed. We made use of Eclipse's Universal Unique Identifier (UUID) for the file, rather than the filename.

2.2 History Exporter

The History Exporter component retrieves the revision history from the local repository and converts it into XML for the other components to use. The wizard can be invoked by selecting File>Export, or by right-clicking the resources and selecting Export. Within the Export wizard, several categories will be available; the file history is found under the "Other" category. Then the developer can choose the resources to be exported, and choose the destination for the export. After the export, each source file has its own history XML file (named "<current-filename> .hist.xml").

[1]It may seem inconvenient apply these settings manually, however, applying the settings before plugin installation allows the plugin to be installed after the monitoring period is over.

The XML history format is depicted in Figure 2. The node `timestamp` contains the creation time of the revision, with precision up to milliseconds. The node `value` can be an empty node, or it can contain `CDATA` value, which is a serialized string of the revision.

```
<file>
    <history>
        <revision>
            <timestamp>1173065453000</timestamp>
            <value><![CDATA[XXXXXXXXX]]></value>
        </revision>
        <revision>
            ...
        </revision>
    </history>
</file>
```

Figure 2: Sample XML History File Format

The exported history files are then used as inputs for the two visualization components, or can be used by other external analysis tools.

2.3 Graphical Visualizer as History Flow

The graphical visualizer is intended to render information-rich graphics to provide an overview of revision history of an individual source file. This is done by delimiting revisions into lexeme blocks and linking them together into a history flow as introduced by [7]. The history flow visualization is then integrated into an Eclipse view.

2.3.1 Algorithm for Delimiting Revisions

We devised an n-revision differencing algorithm to identify fine-grained insertions and deletions in the revisions of a system. The algorithm extends the Heckel's differencing algorithm [5] to compare code bases by different revisions in a lexeme-by-lexeme basis rather than line-by-line. After running Heckel's differencing algorithm $n-1$ times on n revisions, lexemes are generated as in Figure 3. The algorithm is described below.

We define a system S having n revisions r_1, r_2, ..., r_n as a tuple of revisions $S = \langle r_1, r_2, ..., r_n \rangle$, and each revision r_i is a string of characters representing the code base of that revision. Each revision is tokenized into a tuple of lexemes, i.e. $lex(r) = \langle l_1, l_2, ... \rangle$ where $r \in S$, and l_j is a string of characters without whitespaces. For example, lex("The quick brown fox jumps over the lazy dog") = ⟨"The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"⟩.

The Heckel's differencing algorithm can then be defined as a function operating on tuples of lexemes of two revisions as its input, and the output of the algorithm is a pair of tuples of integers s and t representing the results of the algorithm with length $|lex(r_i)|$ and $|lex(r_j)|$ respectively. Formally,

$$\begin{gathered} H(r_i, r_j) = \langle s, t \rangle \text{ where} \\ r_i, r_j \in S, \\ s \in (\{x \in \mathbf{N} : 1 \le x \le |lex(r_j)|\} \cup \{-1\})^{|lex(r_i)|} \text{ and} \\ t \in (\{y \in \mathbf{N} : 1 \le y \le |lex(r_i)|\} \cup \{-1\})^{|lex(r_j)|} \end{gathered}$$

.

Heckel's differencing algorithm works by identifying if the x^{th} lexeme in the lexeme tuple of the i^{th} revision, i.e. $lex(r_i)$

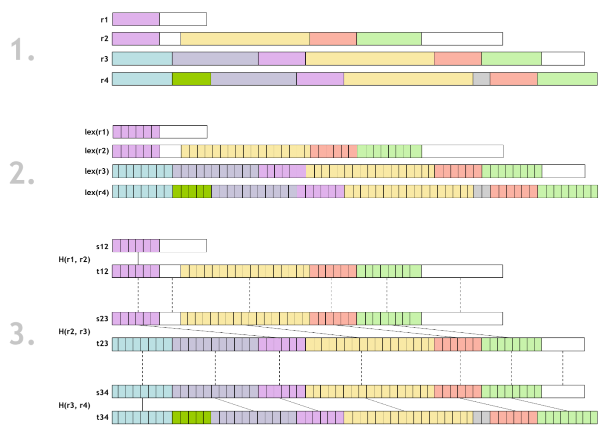


Figure 3: The Steps of the n-revision Differencing Algorithm: 1) Get the revisions; 2) Tokenize revisions into lexemes; 3) Run Heckel's differencing algorithm for every revision pairs

is "identical" to the y^{th} lexeme in the lexeme tuple of the j^{th} revision ($lex(r_j)$), then $s(x) = y$ and $t(y) = x$. The algorithm can also identify the x^{th} lexeme in $lex(r_i)$ as not appearing in $lex(r_j)$, and it will set $s(x) = -1$. Similarly, if the y^{th} lexeme in $lex(r_i)$ is identified as not appearing in $lex(r_j)$, then $t(y) = -1$.

n-revision differencing algorithm utilizes Heckel's differencing algorithm by running it on the revisions of the code base sequentially in a pair-wise manner, i.e.

$$\langle H(r_1, r_2), H(r_2, r_3), ..., H(r_{n-1}, r_n)\rangle =$$
$$\langle\langle s_{12}, t_{12}\rangle, \langle s_{23}, t_{23}\rangle, ..., \langle s_{(n-1)n}, t_{(n-1)n}\rangle\rangle$$

.

The sequence of results is then combined to form a history sequence for each lexeme in each revision. The history sequence traces the location of the lexeme retrospectively to the revision in which the lexeme was inserted. The notation $h(i, j)$ denotes the history sequence of the j^{th} lexeme in the i^{th} revision in the system, and $h(i, j)$ can be defined recursively as

$$h(i,j) = \begin{cases} h(i-1, t_{(i-1)i}(j)) \bullet \langle j\rangle & t_{(i-1)i}(j) \neq -1 \\ \langle j\rangle & t_{(i-1)i}(j) = -1 \end{cases}$$

where $\bullet$ concatenates two tuples, i.e. $\langle s_1, ..., s_m\rangle \bullet \langle t_1, ..., t_n\rangle = \langle s_1, ..., s_m, t_1, ..., t_n\rangle, 1 \leq i \leq |S|$ and $1 \leq j \leq |lex(r_i)|$.

The lexemes in each revisions are then grouped into lexeme blocks such that each adjacent lexeme in the lexeme block has adjacent history sequence. For example, the lexemes having the history sequences $\langle 2, 4, 5, 7\rangle$, $\langle 1, 3, 4, 6\rangle$, and $\langle 3, 5, 6, 8\rangle$ would all belong to the same lexeme block.

The predicate $same(h_1, h_2)$ is defined to be true if and only if the two lexemes having the history sequences h_1 and h_2 are in the same lexeme block:

$$same(h_1, h_2) \overset{\text{def}}{=} (|h_1| = |h_2|) \wedge (h_1(i) - h_2(i) = n)$$
$$\text{where } h_1, h_2 \text{ are two history sequences,}$$
$$\exists n \in \mathbf{Z}, \text{ and } \forall i \in [1..min(|h_1|, |h_2|)]$$

We can then group the lexemes in one revision into lexeme blocks by using the $same(h_1, h_2)$ predicate. The function $block(i, j)$ returns a set representing the lexeme block which contains the j^{th} lexeme in i^{th} revision.

$$block(i,j) = \{k \in [1..|lex(r_i)|\ ] : same(h(i,j), h(i,k))\}$$
$$\text{where } 1 \leq i \leq |S| \text{ and } 1 \leq j \leq |lex(r_i)|$$

For each revision i we can find all lexeme blocks by using the following function.

$$revision\_blocks(i) = \{block(i,j) : \forall\ j \in [1..|lex(r_i)|\ ]\}$$

2.3.2 Viewing History Flow in Eclipse

Lexeme blocks are visualized as a history flow [7] like the one depicted in Figure 4. The implemention of history flow under Eclipse uses Apache Batik[2] as the back-end canvas engine. It can then be panned, and zoomed, or exported to SVG format.

The history flow view shows the time-lapse of code changes. Horizontal bars represent code revisions; the most recent revision is at the bottom. The length of the bar indicates the length of the code base. Each bar consists of one or more lexeme blocks, each separated by a small space. A vertical link connects lexeme blocks that originated from an earlier revision. If a developer adds code, the lexeme block representing that code will not have a link to the previous horizontal bar. Each revision is assigned a random color; the lexeme blocks representing the code added in that revision are filled in that color. Deleted code can be recognized by lexeme blocks with no vertical links extending from them. Code within the lexeme blocks can be viewed by pointing at them.

Figure 4 gives an example of how history flow changes interactively as the code changes. Code in the figure has been colored to reflect their representative lexeme-block.

1. In the first revision, a class `ETX07` was created (dark blue).

2. Next, the programmer declared an instance variable `log` for logging activity and a constructor (purple).

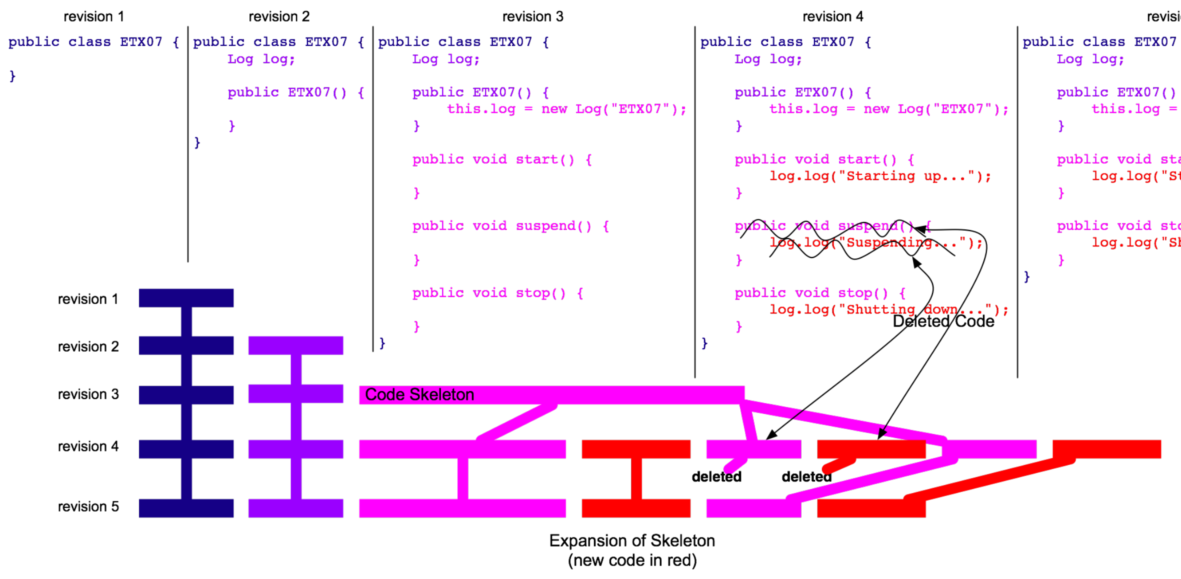
3. The programmer then initialized the instance variable `log` in constructor by instantiating the `Log` object, and also declared three more methods `start()`, `suspend()` and `stop()`. (magenta)

4. Next, logging functionality (red) was introduced into three of the methods in revision 4. These new lexeme blocks are inserted in the corresponding locations into the long magenta lexeme block from Revision 3.

5. Finally, the programmer deleted the method `suspend()`. This is indicated by the two lexeme blocks in Revision 3, with no linked lexeme blocks in Revision 4.

Here, we can see that initially, the code length increases showing implementation of functionality, and then decreases during a refactoring/reworking phase. Also, the activity pattern evident for this code base is that the developer first implemented the skeleton of their methods, and then expanded the code base to add implementation details. If there is any question about the contents of a revision, the code contained in a lexeme block can be viewed by mouse-over. This visualization affords a high-level view of each change the programmer makes, without the need to investigate each revision in detail manually.

[2]http://xmlgraphics.apache.org/batik/



revision 1
```
public class ETX07 {

}
```

revision 2
```
public class ETX07 {
    Log log;

    public ETX07() {

    }
}
```

revision 3
```
public class ETX07 {
    Log log;

    public ETX07() {
        this.log = new Log("ETX07");
    }

    public void start() {

    }

    public void suspend() {

    }

    public void stop() {

    }
}
```

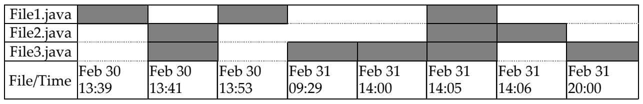
revision 4
```
public class ETX07 {
    Log log;

    public ETX07() {
        this.log = new Log("ETX07");
    }

    public void start() {
        log.log("Starting up...");
    }

    public void suspend() {
        log.log("Suspending...");
    }

    public void stop() {
        log.log("Shutting down...");
    }
}
```

revision 5
```
public class ETX07 {
    Log log;

    public ETX07() {
        this.log = new Log("ETX07");
    }

    public void start() {
        log.log("Starting up...");
    }

    public void stop() {
        log.log("Shutting down...");
    }
}
```

Figure 4: Annotated History Flow Mapped with Java Code in Multiple Revisions

2.4 Tabular Visualizer as Revision Chart

The Revision chart is a tabular representation of revisions of all the project's files in chronological order (Figure 5). Files are sorted by revision order.[3]



File1.java	▩		▩			▩		
File2.java		▩				▩	▩	
File3.java		▩		▩	▩	▩		▩
File/Time	Feb 30 13:39	Feb 30 13:41	Feb 30 13:53	Feb 31 09:29	Feb 31 14:00	Feb 31 14:05	Feb 31 14:06	Feb 31 20:00

Figure 5: A Portion of Revision Chart

The revision chart view allows developers to view project-wide revision histories. This gives insight into the relationships between files (whether files are changed together frequently), and also to visualize the progress of the project.

We have used this revision chart for analysis of several code bases, including the Crossia[4] code base, whose revision chart is shown in Figure 6. This revision chart shows three patterns that we also found in revision histories for other code bases: the sloped-line at the beginning of the chart, the cluster in the middle, and the scatter in the latter portion of the chart. In all the cases we considered, the sloped-line indicated a cascade of file-creation, with no implementation. The middle cluster represented the main development phase, and the scatter portion at the end indicated the maintenance phase.

While these are very general assessments, they show that this visualization can be helpful in assessing overall programmer behavior. We are currently exploring the potential of this visualization for the sake of deeper and more detailed analysis on programmer activity patterns.

[3]Currently, this view is not fully integrated into the Eclipse IDE, but will be in subsequent versions.
[4]http://www.crossia.com/: Crossia is a web-based online job-finder and social-networking site. It spans over 500 system files, with over 50K LOC and 2.5K revisions.

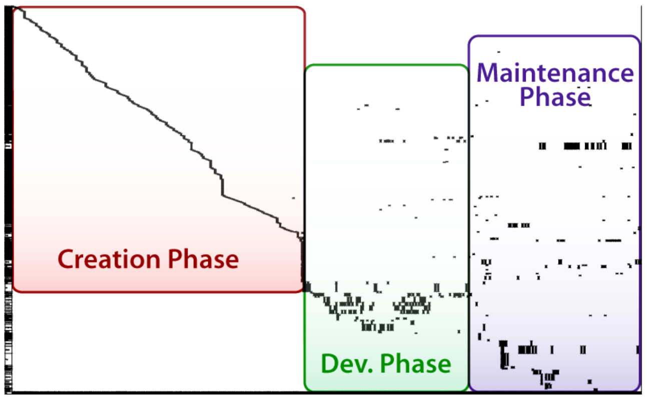


Figure 6: Annotated Revision Chart of Crossia

3. RELATED WORK

Eick et al [2] first coined the term *Software Visualization.* They developed SeeSoft as a line-oriented view where code files resemble vertical lines and colors show recent changes. The code difference is line-based, and revisions were from a legacy versioning system called ECMS. Voinea et al [8] developed CVSscan with multiple-view visualization using revisions from CVS repositories. It visualizes one file at a time, where each revision is shown in a column. Code in revisions are compared to get the global line positions and visualized according to those positions. Colors are used to represent different meanings such as code status, authors and other metrics. These tools differ from ours in that our tool is based on Heckel's diff [5] our differencing is lexeme-based. Our tool also uses Eclipse's local-history function to collect data finer-grained than ECMS and CVS, and does not require developer intervention beyond ensuring that Eclipse's default history values are maintained.

Zimmermann [9] presented an Eclipse plugin APFEL which relies on CVS repositories. Revisions are delimited by tokens in abstract syntax tree in Java, such as modifiers, method calls and keywords. The case study in his paper was conducted for identifying crosscutting concerns, pairing of variable names and renaming variables. The intent of this work is very related to our own, however, CVS-checkin information does not provide fine-grained change tracking within revisions, and we wished to perform analysis of developer behavior that is not necessarily punctuated by check-ins.

Independently, but simultaneously to our work, Harald Kästel-Baumgartner[5] developed an Eclipse plugin using history flow to visualize code changes. The plugin retrieves revision data from CVS repositories and is imported into its own database. The code difference is line-based. Code lengths are represented in vertical axis while revisions are in horizontal axes. It looks very similar to our own history flow. However, the file history views focus on highlighting authorship of code rather than on propagation of code changes.

D'Ambros and Lanza [1] visualized CVS and Bugzilla data to uncover the relationship between evolving software and the way it is affected by software bugs. They developed two visualizations, in which one of them called the TimeLine View. It is similar to our revision chart because it marks revisions in files through time. It also marks bug reports on the view, with colors to show authorship. However, how the file orders is not mentioned, while file list sorted by chronological order in our view shows temporal locality among files.

Hayashi [4] developed an Eclipse plugin for monitoring developers' coding behavior to support refactoring activities. His approach detects every keystroke and stores a modification whenever there is an abstract syntax tree change in Java. Robbes's SpyWare [6] detects and stores atomic changes such as creation and deletion of a node in abstract syntax tree in Smalltalk, and composite changes such as refactorings and bug fixes. It also has a line-chart to show the results of different metrics. Their work can collect fine-grained revisions non-intrusively, but are limited to only one programming language, while our tool is language-insensitive.

4. SUMMARY AND FUTURE WORK

In this paper, we proposed a new non-intrusive approach for collecting and visualizing fine-grained code file revision history information. The approach is provided through an Eclipse plugin with three components: a history exporter, individual-file history flow delimited by lexeme blocks, and project-wide revision charts.

We believe with the support from the plugin, researchers can investigate programmer behavior, and allow analysis of how development processes relate to developer activity phases (for instance,*do programmers engage in refactoring more aggressively when engaging in agile processes?*) and exploration of cultural and demographic questions about developers, allowing for comparison of development patterns between developers with differing cultural background, gender, or educational background.

Additional features will be implemented for the plugin in the future, where some are listed below:

Incorporation of Developers' Activities The current plugin only supports a local repository in Eclipse, such that usually only one developer's activity is monitored for in-process analysis. We are working on synchronizing multiple local repositories with a centralized repository.

Integrated Visualizations in Eclipse History flow is implemented based on Apache Batik, which is a Swing application, while Eclipse is an SWT application. Their interaction is not smooth, and its performance still needs to be improved. The Revision chart is presented as HTML for the sake of prototyping, and it will be well integrated with Eclipse in a view in upcoming versions.

History Analysis as a Guide We intend to explore how the visualizations provided can be used to aid software development tasks.

[5] http://www.filehistory.de/

5. ACKNOWLEDGEMENTS

We would like to thank Edmund Liu and Henry Tang from Crossark Ltd. for providing the histories for our case study.

6. REFERENCES

[1] M. D'Ambros and M. Lanza. Software bugs and evolution: A visual approach to uncover their relationship. Proceedings of 10th European Conference on Software Maintenance and Reengineering, pages 227–236, 2006.

[2] S. Eick, J. Steffen, and E. Sumner Jr. Seesoft-A Tool for Visualizing Line Oriented Software Statistics. IEEE Transactions on Software Engineering, 18(11):957–968, 1992.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[4] S. Hayashi, M. Saeki, and M. Kurihara. Supporting Refactoring Activities Using Histories of Program Modification. IEICE Transactions on Information and Systems, 89(4):1403–1412, 2006.

[5] P. Heckel. A technique for isolating differences between files. Commun. ACM, 21(4):264–268, 1978.

[6] R. Robbes. Mining a Change-Based Software Repository. Proceedings of the Fourth International Workshop on Mining Software Repositories, 2007.

[7] F. Viégas, M. Wattenberg, and K. Dave. Studying cooperation and conflict between authors with history flow visualizations. Proceedings of the SIGCHI conference on Human factors in computing systems, pages 575–582, 2004.

[8] L. Voinea, A. Telea, and J. van Wijk. CVSscan: visualization of code evolution. Proceedings of the 2005 ACM symposium on Software visualization, pages 47–56, 2005.

[9] T. Zimmermann. Fine-grained Processing of CVS Archives with APFEL. In Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange, New York, NY, USA, October 2006. ACM Press.