Historage: Fine-grained Version Control System for Java

Hideaki Hata
Osaka University
Osaka, Japan
h-hata@ist.osaka-u.ac.jp

Osamu Mizuno
Kyoto Institute of Technology
Kyoto, Japan
o-mizuno@kit.ac.jp

Tohru Kikuno
Osaka University
Osaka, Japan
kikuno@ist.osaka-u.ac.jp

ABSTRACT

Software systems are changed continuously for adapting to the environment, correcting faults, improving performance, and so on. For in-depth analysis related to software evolution, it is informative to obtain the histories of fine-grained source code entities. This paper presents a tool named Historage that can provide entire histories of fine-grained entities in Java, such as methods, constructors, fields, etc. A characteristic of Historage is the ability of tracing entity histories including renaming changes. We applied our technique to five open source software projects to quantitatively evaluate the renaming change identification.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Version control*; D.2.9 [**Software Engineering**]: Management—*Software configuration management*

General Terms

Management

Keywords

fine-grained version control, software repository, fine-grained analysis, software evolution

1. INTRODUCTION

Software configuration management (SCM) system data have been mined and analyzed for many research purposes because they contain rich information on real software activities and products; for example, bug prediction based on historical data [15, 16, 23], code clone management [6, 10, 19]. File-level histories can be easily collected from SCM systems, but it is not easy to collect fine-grained entity histories.

The concept of method-level version control in object-oriented programming can be seen in Orwell SCM system [24]. Though several tools have been proposed to support fine-grained version control for development, no such a tool has been actually integrated into widely used SCM systems [5]. These systems intend to control fine-grained entity histories during the development. Since existing repositories remain at file-levels, what we have to do is constructing a fine-grained entity history storage with the data from the existing file-level SCM systems.

Better are required for future research in software evolution [22]. There are several related tools proposed and used in research. *BEAGLE* is a framework incorporating subtools from software metrics software visualization, and relational databases [13, 26]. On the point of fine-grained entity histories, it performs *origin analysis* to identify change types including renaming, moving, splitting, and merging. However, it targeted selected release revisions for applying *origin analysis*. *C-REX* is an evolutionary extractor [14]. It intends to record fine-grained entity changes over the development periods. Though *C-REX* targets entire revisions, it cannot identify renaming. *Kenyon* is designed to facilitate software evolution research [1]. It supports CVS, Subversion, and ClearCase SCM systems and conducts preprocessing tasks for fine-grained change analysis. Though it stores entire revisions, change types are limited to adding, deleting, and modifying. *APFEL* collects fine-grained changes in relational databases [31]. It investigates fine-grained changes at the token level. Though revisions are stored entirely, renaming is not identified.

In this paper, we address the problem of achieving, in storage, fine-grained entity histories as rich as file histories in SCM systems. For providing fine-grained histories, we consider that both storing entire histories (revisions) and identifying renaming (not only providing differences [21]) are important. However there is no tool satisfying both requirements. We propose a system for constructing fine-grained version control storage that satisfies both requirements. We empirically evaluate our system with real open source software projects.

This paper is organized as follows. Section 2 presents our technique for constructing fine-grained version control system and implementation. Empirical evaluation is provided in Section 3. Section 4 discussed related work about rename identification techniques and finally, we conclude in Section 5.

2. HISTORAGE

2.1 Overview

Overview of our system for providing fine-grained entity histories is presented in Figure 1. From file-level snapshots, each entity content (text) is extracted and stored independently. Renaming between two revisions is identified between two revisions. Then each entity history is presented including renames.

We use Git, which is one source code management systems, as a storage. Recently Git attracts some researchers [2, 17]. Bird et

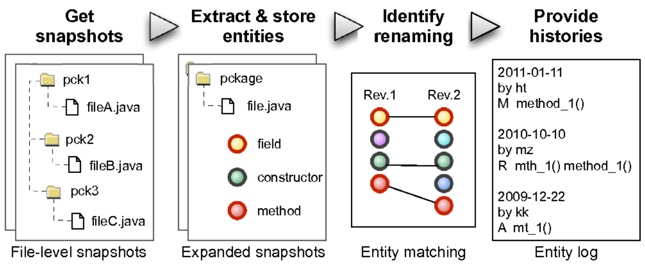


Figure 1: Providing fine-grained entity histories from file-level repositories.

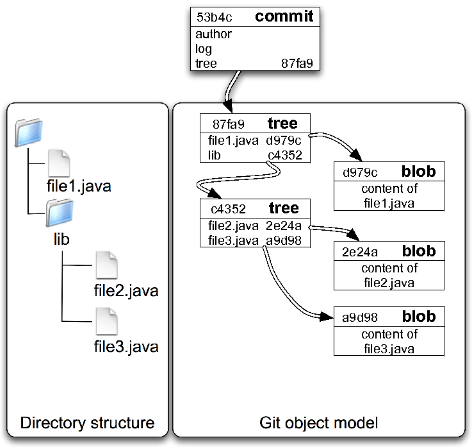


Figure 2: How a snapshot is stored in Git.

al. reported both its promise and peril [2]. Though Git is known for decentralization of source code management, we found that Git architecture is also effective for our purpose.

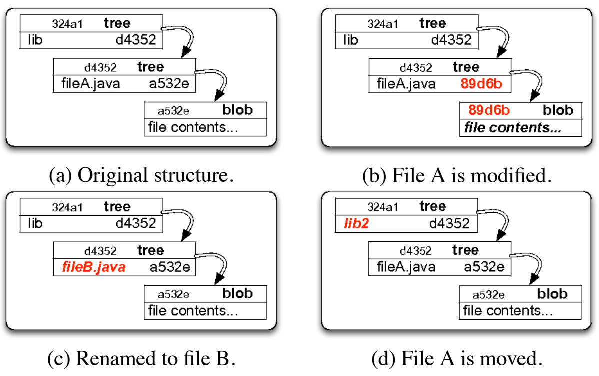
2.2 Preliminary – Git

Git is a content-addressable file system [3]. Git controls file contents, directory structures, file histories, commit logs, etc., by managing Git objects. Each object is stored in Git object database and is compressed and named by the SHA-1 (Secure Hash Algorithm) value of its contents.

Storage of snapshots Figure 2 shows how directory structure of a snapshot is stored and managed with Git object model. The left side of Figure 2 represents sample directory structure at the time of a commit and the right side shows a Git object model that reflect the directory structure. Each blob object, which represents a file, is referred by a tree object. A tree object, which represents a directory, refers blobs and trees. The top tree object is referred by a commit object, which contains the author and log of the commit. As shown in Git object model in Figure 2, each object is identified with SHA-1 value.

Identify changes Here, we explain how Git identify change types with Figure 3. Figure 3 (a) shows an original directory structure in Git object model. It shows that `fileA.java` exists in the directory named `lib`. The content of `fileA.java` is stored in a *blob*, which is named by SHA-1 value: `a5352e`. The `lib` directory is represented as *tree* named `d4352`, and the name of the directory is stored in the `324a1` *tree*.

If the `fileA.java` is modified, the Git object model changes to Figure 3 (b). Since the file content is changed, the corresponding *blob* is also changed. Figure 3 (c) represents the rename of the file.



(a) Original structure. (b) File A is modified.

(c) Renamed to file B. (d) File A is moved.

Figure 3: How changes are detected in Git.

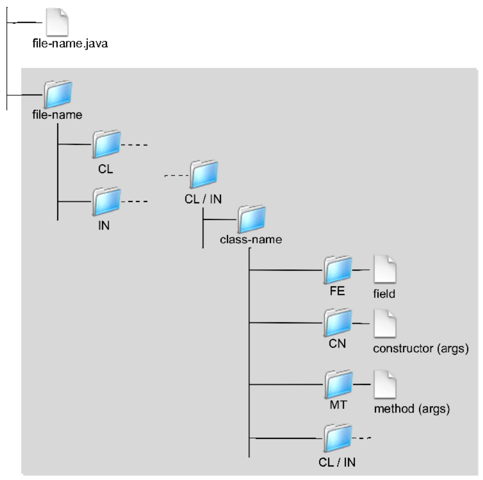


Figure 4: Directory structure for fine-grained entities.

This can be identified because same *blob* SHA-1 value is linked to different file name, `fileB.java`. The Figure 3 (d) represented a directory structure after moving the `fileA.java`. This can be detected because the directory, which has different name `lib2`, contains the `fileA.java`.

When file paths are changed, it is often the case with files that contents of the files are also modified. Even in such cases, Git is able to detect relationships between changes if the file contents are similar enough. This is performed by checking that the amount of deletion of original content and insertion of new content is larger than a threshold, which is set to 50% of the size of smaller files (original or modified). Therefore, if deletion or insertion is less than 50%, two files in parent and child commits are detected as moving or renaming. The threshold value can be changed.

2.3 Technique

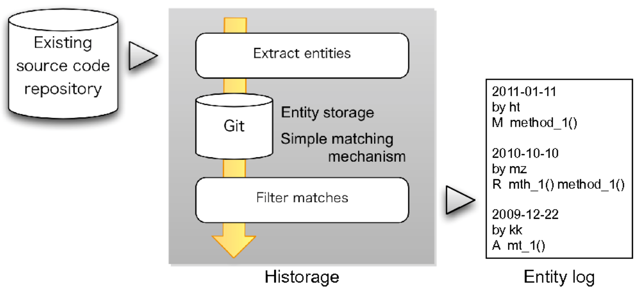
For storing fine-grained entity files, the directory structure is designed as Figure 4[1]. If there are fine-grained entities in a Java file, `fine-name.java`, each entity is additionally stored as a file.

Three kind of entity files are stored in three kind of directories, `FE` (for fields), `CN` (for constructors), and `MT` (for methods). And these directories are stored in a directory identified as class or interface name, which contains those entities as shown at the right part of Figure 4. Anonymous classes are ignored in this paper. Entire

[1]This is a prototypal structure. It is also reasonable to store class declarations for representing logical structures.

Table 1: Open source software projects for evaluation

Project	Description	First Commit	Last Commit	(# of .java)	Total Commits
WTP incubator	Subproject of Eclipse IDE	2007-11-10	2010-07-22	1,944	541
Hadoop	Utilities for distributed application framework	2009-05-19	2010-12-26	667	375
Subversion	SCM system	2000-03-01	2010-11-29	127	738
jEdit	Text editor	2001-09-02	2010-10-02	546	4,399
Android	Mobile operating system	2008-10-21	2010-12-23	2,690	25,965



```
2011-01-11
by ht
M method_1()

2010-10-10
by mz
R mth_1() method_1()

2009-12-22
by kk
A mt_1()
```

Figure 5: Historage architecture.

files and directories are stored in the `file-name` directory. Directories and files in gray space of Figure 4 are newly prepared for new directory structure.

The entities we target in this paper are named as follows for files:

Field: field name.

Constructor: constructor name and parameter list.

Method: method name and parameter list.

Changes of entity names correspond to file name changes, and moving of entities correspond to moving files. If an entity is deleted in a commit and reappear in a later commit, Git can output its history including disappearing periods.

As described in Section 2.2, renames are identified based on file content similarity. If two entities are highly similar, it is rational to detect them as corresponding entities. Because this matching technique is simple, there may be obvious mismatches, that is, matches between different entity types, such as a match between method and constructor, for example. These mismatches are distinguished easily by checking directory names whether they are same or not. We filter out these mismatches before providing entity histories.

2.4 Architecture

Figure 5 shows the architecture of Historage[2]. As shown in Figure 1, extracting and storing fine-grained entities are conducted on each snapshot. A snapshot in each revision can be obtained easily from Git. Even if existing repositories are not in Git system, it is possible to convert them to Git repositories from most SCM systems. For extracting the fine-grained entities in Java files, we use the source code analysis tool MASU[3], which is an open source tool. The threshold value for rename identification is set to 30% (as a option of Git commands) based on empirical study reported in Section 3 for filtering appropriate matching entities beyond renaming and moving.

[2]A prototype will be available at `http://www-ise4.ist.osaka-u.ac.jp/~h-hata/`.

[3]`http://sourceforge.net/projects/masu/`

Table 2: Rename identification results in five open source software projects

Project		Sum†	Correct (%)		Measure (%)
WTP incubator	mismatches	62			Rec. 96.7 Prec. 99.6
	$s < 30$	366	99	27.0	
	$30 \leq s < 100$	436	426	**97.7**	
	$s = 100$	2,641	2,641	**100.0**	
Hadoop	mismatches	32			Rec. 88.1 Prec. 100
	$s < 30$	152	43	28.3	
	$30 \leq s < 100$	141	141	**100.0**	
	$s = 100$	178	178	**100.0**	
Subversion	mismatches	41			Rec. 96.4 Prec. 99.7
	$s < 30$	148	88	59.5	
	$30 \leq s < 100$	528	521	**98.7**	
	$s = 100$	1,820	1,820	**100.0**	
jEdit	mismatches	254			Rec. 94.4 Prec. 99.9
	$s < 30$	1,229	347	28.2	
	$30 \leq s < 100$	1,461	1,457	**99.7**	
	$s = 100$	4,421	4,421	**100.0**	
Android	mismatches	203			Rec. 99.8 Prec. 99.98
	$s < 30$	1,125	98	8.7	
	$30 \leq s < 100$	912	903	**99.0**	
	$s = 100$	61,278	61,278	**100.0**	

†: entity pairs exist in January, 2010 for the `Android` project, and entire entity pairs for the other projects.
s: similarity.

3. EVALUATION

In this Section, we empirically investigate the usefulness of our fine-grained version control system, *Historage*.

3.1 Target Projects

As shown in Table 1, we select five open source software projects: Eclipse WTP incubator (WTP incubator),Apache Hadoop Common (Hdoop),Apache Subversion (Subversion),jEdit,and Android framework classes and services (Android).These projects are written in Java and Git repositories are available. We cloned the Git repositories on the 27th December, 2010.

The disk space overhead compared with original repositories and constructed *Historage* depends on projects. It varies from nearly equal to a few times on Git database.

3.2 Rename Identification

We investigated every matching pairs of fine-grained entities in the repositories (the number of commits are shown in Table 1) except for the Android project. As there are more than $180,000$ matching pairs in the Android project, we limited the pairs to those existing on January, 2010, for the Android project. Entity pairs are classified according to similarity values, which are calculated by Git, to see the impact of the threshold and investigated the effectiveness of rename identification. We determine by hand if a matching is correct or not.

Table 2 shows the results in the five projects. Mismatches are matches between different entity types. It is possible to distinguish them automatically. Shown in bold fonts, the percentage of correct matches when similarity is greater than or equal to 30% is higher

Table 3: Rename identification results for entity types in WTP incubator project

Entity		Sum†	Correct (%)		Measure (%)
	$s < 30$	33	6	18.2	
Field	$30 \leq s < 100$	45	39	86.7	Rec. 99.4
	$s = 100$	1,142	1,142	**100.0**	Prec. 99.4
	$s < 30$	21	11	52.4	
Constructor	$30 \leq s < 100$	59	59	**100.0**	Rec. 94.4
	$s = 100$	129	129	**100.0**	Prec. 100
	$s < 30$	312	82	26.3	
Method	$30 \leq s < 100$	332	328	**98.8**	Rec. 95.4
	$s = 100$	1,370	1,370	**100.0**	Prec. 99.8

†: entire entity pairs.
s: similarity.

than 97% in every project. This means that we can provide more than 97% correct rename histories of fine-grained entities. However there seem to be actual renaming when similarity is less than 30%, it is now difficult to distinguish by our system. The recall and precision values, where *Historage* provide rename changes with the threshold value 30%, are presented in Table 2.

Though Table 2 reports the results of every fine-grained entity change types together, we can conduct more detailed analysis. Table 3 represents the rename identification results for each fine-grained entity type in the WTP incubator project. We can see that the percentages of correct pairs are different depending on the entity types. For example, the result on `field` is relatively low. We think this is because it is more difficult to compare the similarity with the small contents of fields. On the contrary, change type identification of `constructor` achieved relatively high result. We think this is because there is a small number of potential constructor pairs compared to method pairs. Similar results can be seen in the other projects.

With the investigation of the results, we found that automatic rename identification by Git and our filtering works relatively well. Distinguishing actual renames when similarity values are less than 30% is parts of our future work.

4. RELATED WORK

There are many studies about identifying changes.

One-to-one matching techniques Based on the matching technique survey by M. Kim and Notkin, one-to-one software entity matching techniques are summarized as follows: *entity name matching*, *string matching*, *syntax tree matching*, *control flow graph matching*, *program dependence graph matching*, *binary code matching*, *clone detection*, and *origin analysis tools* [18]. S. Kim et al. applied several method matching techniques for *origin analysis* limited to renaming and moving to open source software projects, and evaluated the effectiveness of the techniques [20]. They reported that though *clone detection* yields an accuracy value 67.4, *function body diff* achieved 90.2.

Splitting and merging Splitting and merging of software entities are targeted by *origin analysis*. Godfrey and Zou proposed a technique of inferring such events based on matching procedures using multiple criteria including names, signatures, metric values, and call dependencies [13]. Splitting and merging correspondence analysis is also known as one-to-many and many-to-one matching [28]. Wu et al. combined text similarity analysis and call dependency analysis for those method matching [28].

Systematic structural changes Recognizing structure changes including refactorings and object-oriented design changes is one of hot topics of change analysis. These analyses are based on techniques of matching object-oriented program elements. With the differences of program elements, it is inferred what structure changes are occurred. *RefactoringCrawler* detect refactorings based on identifying renaming packages, classes, methods, and moving methods [7]. Those changes are identified by using structural data, call-graph and tokens from entities. *MolhadoRef* [8,9] is a semantics-based and refactoring-aware SCM system [12]. It adopts *RefactoringCrawler* [7] and uses refactoring logs to support merging. Weißgerber and Diel presented a technique to detect changes that are likely to be refactorings [27]. Their matching technique is based on structure similarity and code clone analysis.

Table 4: Change identification techniques and using data

Technique	Graph	Feature
S. Kim et al. [20]	calls	name, text, metrics
Godfrey and Zou [13]	calls	name, metrics
Wu et al. [28]	calls	text
Dig et al. [7]	calls, structure	tokens
Weißgerber and Diel [27]	structure	name, text
Xing and Stroulia [29]	structure	name
Dagenais and Robillard [4]	calls, structure	name
This paper	-	text

Framework usage changes Xing and Stroulia proposed an approach for API-evolution support, called Diff-CatchUP [30]. On the step of change identification, UML-diff, which is based on name similarity and code dependency similarity of program elements [29], is used. After identifying changes, plausible API replacements are proposed. Dagenais and Robillard presented a technique to recommend adaptive changes for clients of framework code based on structure change analysis [4]. Their matching technique is based on structure similarity and out going call dependency similarity.

Discussion Though there are some variations, change identification is a kind of matching problems. In computer vision research area, similar problems are known as the *correspondence problem* and techniques are classified in following two classes [25]:

Graph-based methods: checking if correlations on graph structures are similar or not.

Feature-based methods: finding features and seeing if they are similar or not.

Table 4 summarized the studies based on this classification. As shown in Table 4, every study except for this paper uses both methods for change type identification. As *graph-based methods* and *feature-based methods* have different advantages and limitations, the combination of both methods is expected to achieve better results. Most studies mainly adopt *graph-based methods* and use *feature-based methods* for improving method correspondence problems.

Graph-based methods require unchanged or easily understandable correlated parts. Therefore, it is difficult to identify corresponding entities if there is no enough correlated part or there are major changes. Wu et al. reported the limitations and insist that graph-based analysis cannot be overcome them [28]. Though it is different entity (AST node) analysis, Fluri et al. proposed an algorithm based on *graph-based methods* and reported following two limitations [11]:

- Mismatching can propagate. Not only mismatching for each targeting entity, correlate entities can be mismatched.
- The worst-case complexity increase. To decrease mismatching, complex algorithm is needed and this increase the worst-case complexity.

Since our tool is based only on feature-based method and do not analyze call-dependencies, it is difficult to recommend alternative method calls for example. In addition, it is not possible to identify multiple refactorings, now. However, there is no limitations of graph-based methods and our matching and filtering technique relatively work well. With *Historage*, fine-grained analysis is possible for repository mining research, such as bug prediction [15,16,23] and code clone management [6,10,19].

5. CONCLUSIONS

Since software repositories are great sources of software development data, repository mining based research has the possibility of powerful empirical analysis. For research based on SCM systems, fine-grained change analysis is a desirable approach for an in-depth study compared to file-level change analysis. This paper presents an automatic technique for constructing fine-grained version control system from an existing SCM repository. By utilizing the Git architecture, our system stores entire revisions, and can identify rename changes of fine-grained entities, and provide accessible system for further research. We empirically evaluate our system with five open source software projects and found that our system works well with those projects. Our future work includes more in-depth comparison with other approaches and rename identification when similarity values are low.

6. ACKNOWLEDGMENTS

We would like to thank Dr. Livieri Simone in Osaka University for valuable comments from early versions of this paper, and thank the MASU developers in Osaka University for providing the tool and valuable advice. Authors would like to thank anonymous reviewers who gave us many valuable suggestions to the paper. This research is supported by Grant-in-Aid for JSPS Fellows (No.23-4335).

7. REFERENCES

[1] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey. Facilitating software evolution research with kenyon. ESEC/FSE-13, pp. 177–186, 2005.

[2] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. MSR '09, pp. 1–10, 2009.

[3] S. Chacon. *Pro Git*. Apress, Berkely, CA, USA, 1st edition, 2009.

[4] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. ICSE '08, pp. 481–490, 2008.

[5] A. De Lucia, F. Fasano, R. Oliveto, and D. Santonicola. Improving context awareness in subversion through fine-grained versioning of java code. IWPSE '07, pp. 110–113, 2007.

[6] M. de Wit, A. Zaidman, and A. van Deursen. Managing code clones using dynamic change tracking and resolution. ICSM '09, pp. 169 –178, 2009.

[7] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. ECOOP '06, pp. 404–428, 2006.

[8] D. Dig, K. Manzoor, R. E. Johnson, and T. N. Nguyen. Effective software merging in the presence of object-oriented refactorings. *IEEE Trans. Softw. Eng.*, 34:321–335, May 2008.

[9] D. Dig, T. Nguyen, and R. Johnson. Refactoring-aware software configuration management. Technical report, UIUCDCS, 2006.

[10] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. ICSE '07, pp. 158–167, 2007.

[11] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33:725–743, November 2007.

[12] T. Freese. Refactoring-aware version control. ICSE '06, pp. 953–956, 2006.

[13] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Eng.*, 31:166–181, February 2005.

[14] A. E. Hassan and R. C. Holt. C-REX: An evolutionary code extractor for c, 2004.

[15] A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. ICSM '05, pp. 263–272, 2005.

[16] H. Hata, O. Mizuno, and T. Kikuno. Fault-prone module detection using large-scale text features based on spam filtering. *Empirical Softw. Eng.*, 15:147–165, April 2010.

[17] I. Herraiz, G. Robles, and J. M. Gonzalez-Barahona. Research friendly software repositories. IWPSE-Evol '09, pp. 19–24, 2009.

[18] M. Kim and D. Notkin. Program element matching for multi-version program analyses. MSR '06, pp. 58–64, 2006.

[19] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. ESEC/FSE-13, pp. 187–196, 2005.

[20] S. Kim, K. Pan, and E. J. Whitehead, Jr. When functions change their names: Automatic detection of origin relationships. WCRE '05, pp. 143–152, 2005.

[21] J. I. Maletic and M. L. Collard. Supporting source code difference analysis. ICSM '04, pp. 210–219, 2004.

[22] T. Mens. The ERCIM working group on software evolution: the past and the future. IWPSE-Evol '09, pp. 1–4, 2009.

[23] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. ICSE '05, pp. 284–292, 2005.

[24] D. Thomas and K. Johnson. Orwell-a configuration management system for team programming. OOPSLA '88, pp. 135–141, 1988.

[25] E. Trucco and A. Verri. *Introductory Techniques for 3-D Computer Vision*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.

[26] Q. Tu and M. W. Godfrey. An integrated approach for studying architectural evolution. IWPC '02, pp. 127–136, 2002.

[27] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. ASE '06, pp. 231–240, 2006.

[28] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim. Aura: a hybrid approach to identify framework evolution. ICSE '10, pp. 325–334, 2010.

[29] Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. ASE '05, pp. 54–65, 2005.

[30] Z. Xing and E. Stroulia. Api-evolution support with diff-catchup. *IEEE Trans. Softw. Eng.*, 33:818–836, December 2007.

[31] T. Zimmermann. Fine-grained processing of CVS archives with APFEL. eclipse '06, pp. 16–20, 2006. ACM.