Inferring Restructuring Operations on Logical Structure of Java Source Code

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

Restructuring source code structure, such as moving and renaming classes or methods, are inevitable activities in software development, and are recommended for the improvements of maintainability. However, it has been not easy to understand properly what logical structural changes occur. This is because we can obtain only file-level and line-level information from source code management systems about changes. This paper presents a technique of such inferring restructuring operations on logical structure of Java source code. For inferring structural change operations, the core part is mapping elements between two revisions. Previous related studies tackle this problem based on the analysis of subgraph similarity, which takes lots of time. We find match candidates based on the similarity of element contents and identify matches with Bayesian inference based on empirical data. We report the result of empirical evaluation of our technique with open source software projects from Android and Eclipse. We see that our technique identify most element matches correctly and provide appropriate operations, and it took only a few seconds to analyze entire history of each project.

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

change analysis, refactoring, software evolution

1. INTRODUCTION

Software evolves dynamically. While developing and maintaining source code, restructuring source code structure including moving and renaming program elements is a common practice. From an empirical study, Murphy-Hill et al. reported that though pure refactorings (root-canal refactorings) rarely occurred, floss refactorings, which are refactorings with other types of programming activities, occurred frequently [18].

Though refactorings are recommended and restructuring source code occurs frequently, there is a problem on understanding those changes. Mailloux reported an experience of industrial software development from the initial implementation to several business phases [17]. In this report, it is described how configurations, bugs, changes, and so on were managed as the project grew. At the initial implementation, there was no change management. From the first release, informal one-to-one coaching and formal documentation began. As the project grew, an initial training was provided to developer team. However, it is reported that changes were so fast, then the initial training became obsolete. As seen in this experience report, it has been not easy to understand changes.

This paper presents a technique of inferring change operations, especially restructuring change operations on logical structure of Java source code. We target moving and renaming of program elements, such as packages, classes, fields, constructors, and methods. There are several studies providing change operations between two revisions. One approach is a record-and-replay technique [3, 12]. Though these approach are able to provide accurate change operations, it is not always possible to record change operations because developers do not always use refactoring tools [18]. The other studies based on matching techniques. Change operations are inferred based on identified program element matches. The objective of most previous studies were identifying what changes occur between two releases. Most approaches based on the subgraph isomorphism problem, which requires large time to analyze. Consequently, analyzing entire histories (every changes) is not practical because of large time consumption.

We propose a light-weight technique to overcome this limitation. Our technique do not directly infer restructuring based on finding subgraph isomorphism, but there are mainly two phases to infer restructuring. First, we identify matches between individual element using simple heuristics and then restructuring is inferred using Bayesian inference based on empirical data. Our technique is empirically evaluated with open source software projects in Android and Eclipse. From this evaluation, we see that our technique infers most change operations properly and takes only a few seconds to analyze entire history of each project.

The rest of this paper is organized as follows. Section 2 describes restructuring operations we target and clarifies that program element matching problem is the core of structure change operation inference. Section 3 discusses program element matching problem

Previous revision

```
package pck;

public class Cls_a {
 void mth_x () {
 }
}

public class Cls_b {
 void mth_y () {
 }
}
```

Changed revision

```
package pck;

public class Cls_a {
 void mth_p (int a) {
 }
}

public class Cls_c {
 void mth_q () {
 }
}
```

Figure 1: A change example between two revisions.

with related work. In Section 4, we explain our inference algorithm and evaluate our technique in Section 5. Finally, we conclude in Section 6.

2. RESTRUCTURING OPERATIONS

Motivation. Changes of software structure is inevitable and important, but it is difficult to understand such changes. In some paper, it is reported that the lack of change management is big risk of software quality and team management [17, 21]. Tools that help developers to understand restructuring changes should be required.

Definition. The problem addressed in this paper is proposing a technique that suggests restructuring operations applied to change software structure between two revisions of software. In source code, there are some structures, such as physical structures (directories and files), logical structures (packages, classes, methods, and so on), dependencies (define-use and overriding) [14]. This paper targets changes on logical structures. We treat *packages*, *classes*, *fields*, *constructors*, and *methods* as program elements in logical structures of Java source code. Targeted restructuring operations include *rename*, *move*, *hide*, and *unhide* following the terms in [22]. Rename means the identifier changes in this paper. So method renames is the changes of method signatures, that is changes on method name, parameters.

Overview. Figure 1 is an example of source code change. If a method mth_p is identified as a modified version of a method mth_x, that is, a match is found between the two methods, it is easy to interpret that the method is renamed from mth_x to mth_p and its parameter is changed. The method is not moved since the method exists in a same class Cls_a. If a match is found between a method mth_y and a method mth_q, it is easy to suggest that the method is renamed. However, this case is different from the previous case. The method mth_y exists in a class Cls_b and the method mth_q exists in a class Cls_c. If the class Cls_b and the class Cls_c are different one, we can recognize that the method is moved from the class Cls_b to Cls_c. If the class Cls_c is a renamed version of Cls_b, the method is not moved but just renamed. Changes on element names or parameters for *constructors* and *methods* are identified based on corresponding matches. To identify whether elements are moved or not, it is needed to investigate their parent element matches. In summary, we have to identify every program element matches.

3. PROGRAM ELEMENT MATCHING

The problem of identifying program element matches can be seen as a *link prediction problem* [6, 13], which is a problem of predicting the existence of a link between two nodes in a network as shown in Figure 2. For program elements, networks can be seen in logical structures, call dependency graphs, and so on. Existence of the link can be regarded as a match between two program elements. The link prediction problem can fall into two categories in accordance with the information used for prediction [13]:

Topological-information-based methods: nearby nodes are similar or not.

Node-information-based methods: attributes of nodes are similar or not.

There are many studies inferring change operations. Table 1 summarizes previous studies based on the two information

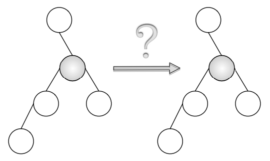


Figure 2: Link prediction problem.

Table 1: Information for program element matching

Studies	Topological info.	Node attributes
S. Kim et al. [16]	calls	name, text, metrics
Godfrey and Zou [7]	calls	name, metrics
Wu et al. [23]	calls	name
Fluri et al. [5]	structure	name
Dig et al. [2]	calls, structure	tokens
Weißgerber and Diehl [22]	structure	name, text
Prete et al. [20]	calls, structure	text
Xing and Stroulia [24]	structure	name
Dagenais and Robillard [1]	calls, structure	name

Origin identification. S. Kim et al. applied several method matching techniques for *origin analysis* identifying renaming and moving to open source software projects, and evaluated the effectiveness of the techniques [16]. They reported that though *clone detection* yields an accuracy value 67.4, *function body diff* achieved 90.2. 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 [7]. Splitting and merging correspondence analysis is also known as one-to-many and many-to-one matching. Wu et al. combined text similarity analysis on names and call dependency analysis for those method matching [23]. Fluri et al. proposed *change distilling*, a tree differencing algorithm [5]. *Change distilling* target not only method-level changes but also more fine-grained element changes. Name string similarities and tree similarities are calculated for matching.

Refactoring identification. Dig et al. proposed a technique for detecting refactorings based on identifying renaming packages, classes, methods, and moving methods [2]. Those changes are identified by using structural data, call-graph and tokens from entities. Weißgerber and Diehl presented a technique to detect changes that are likely to be refactorings [22]. Their matching technique is based on structure similarity and code clone analysis. M. Kim and Notkin proposed an approach, *LSdiff* to discover and represent systematic code changes [14]. They intended to infer what changes are occurred based on analyzed structure differences. The matches are analyzed based on a set of predicates that describe program elements, their containment relationships, and their structural dependencies. *REF-FINDER* proposed by Prete et al. extends predicate sets of *LSdiff* for identifying refactorings [20]. It supports sixty-three refactoring types. Though original *LSdiff* does not identify matches, *REF-FINDER* does.

Framework usage changes. Xing and Stroulia proposed an approach for API-evolution support, called Diff-CatchUP [25]. On the step of change identification, UML-diff, which is based on name similarity and code dependency similarity of program elements [24], 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 [1]. Their matching technique is based on structure similarity and out going call dependency similarity.

Discussion. As shown in Table 1, every study uses both methods for program element matching. As *topological-information–based methods* and *node-information-based methods* have different advantages and limitations, the combination of both methods is expected to achieve better results. Most studies mainly adopt *topological-information-based methods* and use *node-information-based methods* for program element matching.

Topological-information-based methods require unchanged or easily understandable neighborhood. Therefore, it is difficult to identify matching elements if there is no enough nearby elements or there are major changes. Wu et al. reported the limitations and insist that topological-information-based methods cannot be overcome them [23]. Fluri et al. reported following two limitations [5]:

- 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.

Because of these problems, previous techniques are not light-weight for analyzing entire histories. In addition, some studies report the difficulties of identifying moving operations [5, 24].

4. INFERENCE ALGORITHM

Our algorithm infers restructuring operations between two revisions of Java source code. Our algorithm consists of three parts: (1) finding candidates of program element matches, (2) identifying program element matches, (3) interpreting restructuring operations.

For program element matching in the part (1) and (2) , we use only node information because there are problems in using topological information as seen before.

4.1 Finding match candidates

We have proposed a system *Historage*[1] that can track program elements beyond renaming and moving [10, 11]. With this system, match candidates between program elements can be found based on the similarity of their text. We have found that it is possible to find most of match candidates in *methods*, *constructors*, and *fields* if contents are similar enough [11]. It is also possible to find match candidates between *classes* with the same technique.

[1]A tool to build Historage is available from `https://github.com/hdrky/git2historage`.

The percentages of the same content in the size of smaller content (original or new) are calculated as text similarity values. In the previous study [11], we immediately identify matches based only on the similarity value. If the value is larger than or equal to 30%, elements can be regarded as matches, and if the value is less than 30%, elements are regarded as independent elements.

Though this procedure works relatively well, we miss some matches if the similarity values are low, which is caused by major modification. The next part of our algorithm is introduced for decreasing such missing.

4.2 Identifying matches

Though the high similarity value is a good evidence for finding element matches, we can use other node information as additional evidence. These additional evidence should be valuable especially when the similarity value is low. In this part, we identify matches based on Bayesian inference. If we can obtain additional node information X, we can calculate the posterior probability of matches as follows:

$$P(match|X) = \frac{P(match)P(X|match)}{P(X)}$$

We identify matches if the posterior probability $P(match|X)$ is greater than or equal to 50%, where $P(X) = P(match)P(X|match) + P(match)P(X|\texttt{not}match)$. To build not a project-specific model but a general model, we will determine parameters based on empirical investigation of several open source projects.

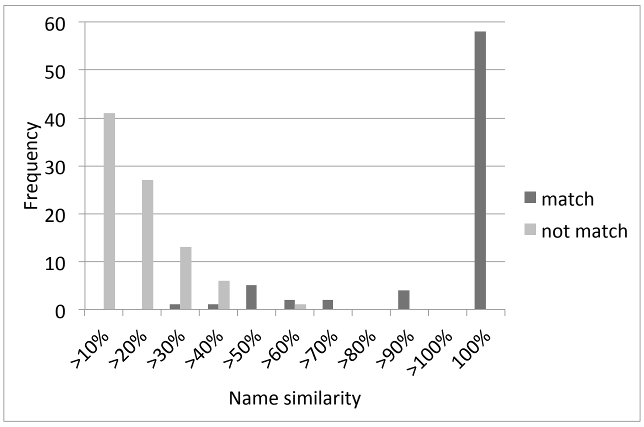
Prior probability. From the empirical study [11], if the similarity value is greater than or equal to 30%, more than 95% of matches are correct. There are not many matches if the similarity value is less than 30%. Based on this observation, we determine the prior probability as follows:

High text similarity: $P(match) = 95\%,\ \ P(\text{not } match) = 5\%$

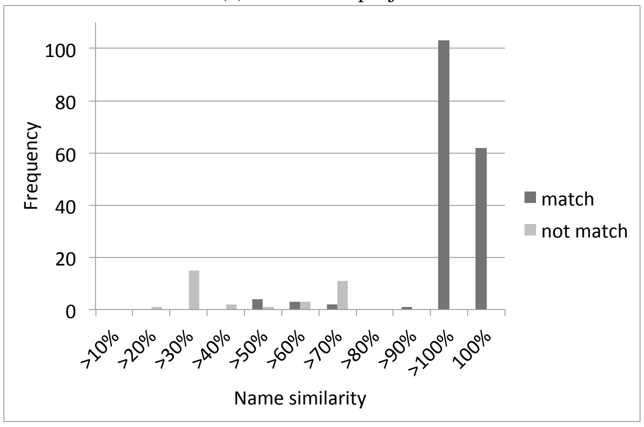
Low text similarity: $P(match) = 20\%,\ P(\text{not } match) = 80\%$

Evidence. For additional evidence, we collect following two node information: (i) names of program elements, (ii) existence of corresponding child elements. The similarity of names between two elements are widely used for matching [1, 5, 7, 16, 22–24] as seen in Table 1. The existence of corresponding child elements is an additional evidence from our observation. If there is a match between classes, which means the matched class is equal, there should be elements that exists in the previous and the new class. Though additional information (i) can be used for every program element types, (ii) can be used for only *class* and *package*.

(i) names of program elements. To compute the similarity of program element names (s_1 and s_2), we calculate their *longest com-*



(a) `Browser` project



(b) `Xpand` project

Figure 3: Name similarity and matches.

mon subsequence (LCS). We adopt the following expression proposed in [7] for the name similarity:

$$\frac{length(LCS(s_1, s_2) * 2)}{length(s_1) + length(s_2)}$$

Based on the name similarity, we want to determine the parameter of $P(\text{name sim.}|match)$ and $P(\text{name sim.}|\text{not } match)$. We investigate four open source software projects (Browser, Phone, EMF Compare, Xpand) to see the relation of the name similarity and the existence of program element matches. Figure 3 shows the distribution of program element matches based on the name similarity in two projects, Browser and Xpand. Though most matches have higher name similarities (more than or equal to 70%), there are a few matches that have middle name similarities (40% to70%). Not match candidates have low name similarities (less than 40%) and middle name similarities. We found similar distribution on every project. Based on these observation, we determine the parameter as follows:

$$\begin{aligned}
P(\text{name sim.}high|match) &= 0.85\\
P(\text{name sim.}middle|match) &= 0.1\\
P(\text{name sim.}low|match) &= 0.05\\
P(\text{name sim.}high|\text{not } match) &= 0.05\\
P(\text{name sim.}middle|\text{not } match) &= 0.15\\
P(\text{name sim.}low|\text{not } match) &= 0.8
\end{aligned}$$

(ii) existence of corresponding child elements. We investigate the existence of corresponding child elements for program element match candidates. From empirical investigation, we observed that there are a few matches without corresponding child elements, and there are few cases for not matches with child elements. We determine the parameters as follows:

$$\begin{aligned}
P(\text{exists child}|match) &= 0.9\\
P(\text{not exists child}|match) &= 0.1\\
P(\text{exists child}|\text{not } match) &= 0.05\\
P(\text{not exists child}|\text{not } match) &= 0.95
\end{aligned}$$

Built model. The name similarity and the existence of corresponding child elements can be seen as independent features. Therefore we a built naive Bayes classifier as follows:

$$P(\text{name sim., child}|match) = P(\text{name sim.}|match)P(\text{child}|match)$$

Using determined parameters, we build a classifier model. Instead of the detail posterior probability values, we show that when our model identify match candidates as matches. For *methods*, *constructors*, and *fields*, which do not have child elements, matches are identified if any one of the following conditions is satisfied:

- Text similarity value is greater than or equal to 30%.
- Name similarity value is greater than or equal to 70%.

For *classes* and *packages*, matches are identified if any one of the following conditions is satisfied:

- Text similarity value is greater than or equal to 30% and name similarity value is greater than or equal to 70%.
- Text similarity value is greater than or equal to 30% and there are corresponding child elements.
- Name similarity value is greater than or equal to 40% and there are corresponding child elements.

Program element match identification begins for *methods*, *constructors* and *fields*. After identifying these matches, it is easy to know there are corresponding child elements for *classes*. Then we identify matches for *classes*. Finally, we identify matches of *packages*. As seen in our classify model, we use only node attribute information, which should fit our intuition.

4.3 Interpreting restructuring operations

After identifying every program element matches, we interpret restructuring operations. Renaming is easily known between matches. As seen in Section 2, moving can be identified after clarifying whether parent elements are same (matched) or different (not matched). Though some paper describes the limitations of identifying moving operations [5, 24], there is no such limitation in our technique. Now our technique support the following restructuring operations: move, rename, parameter change, access modifier change (hide or unhide).

Figure 4 presents an example of inferred restructuring operations. In a package `com.android.phone`, there are changes of two methods as follows:

```
Package com.android.phone
  PhoneApp.java
    Class PhoneApp
      Method displayCallScreen() -> private displayCallScreen(): 1. hide
  PhoneUtils.java -> CallNotifier.java
    Class PhoneUtils -> CallNotifier
      Method showIncomingCallUi() -> private showIncomingCall(): 2. move & rename & hide
```

Figure 4: An example of inferred restructuring operations.

Table 2: Target project data.

Project		Initial	Last	# Changes
Android	Browser	2008-10-21	2011-05-03	1,517
	Contacts	2008-10-21	2011-04-04	2,082
	Phone	2008-10-21	2011-05-31	2,253
Eclipse	ECF	2004-12-03	2011-05-17	5,251
	EMF Compare	2007-04-03	2011-05-24	860
	Xpand	2007-11-10	2011-05-31	637

1. A method `displayCallScreen()` is **hidden** by being attached a `private` access modifier.

2. A method `showIncomingCallUi()` that existed in a class `PhoneUtils` is **moved** to a class `CallNotifier`, and is **renamed** `showIncomingCall`, and **hidden** by being attached a `private` access modifier.

These information should be useful for further research on fine-grained level, such as software evolution analysis, historical information based fault-prone/failure-prone module prediction, code clone management, and so on. Text-based output like Figure 4 may not be human readable. Appropriate visualization is one of required future work.

5. EVALUATION

In this section, we evaluate the accuracy of our technique and the performance of analysis time. We investigate the accuracy of identifying program element matches because inference of restructuring operations depends on this identification. As shown in Table 2, we select six open source software projects from Android and Eclipse to empirically evaluate our technique. Each project is developed more than two years and is committed (changed) about 500 to $5,000$ times. These projects are written in Java and Git repositories are available.

5.1 Program element matching

We manually investigate every match candidates. To evaluate with *Recall* measure, we need to prepare reference set that should be collected from every potential matches, which is very hard task. Hence we measure *CRecall*, which is the number of identified correct matches divided by the number of correct matches in match candidates. From our large inspection, there are few cases that there are correct matches that are not identified as match candidates. *Precision* is the number of identified correct matches divided by the number of all identified matches. Since we manually identify correct matches, we may introduce some bias.

Table 3 summarizes the result of each project. The result is divided in two tables based on the text similarity values, that is, (a) for text similarity value is greater than or equal to 30% and (b) for text similarity value is less than 30%, because match identification with low text similarity is more difficult than with high text similarity. From Table 3 (a), which is a summary of matching with high text similarities, we can see that all matches are identified (every CRecall value is 1.00) and there are a few false positives (precision values range from 0.94 to 1.00). Table 3 (b) is a summary of matching with low text similarities. This case is relatively difficult because there are not many correct matches in entire match candidates as seen in the second and the fifth row of Table 3. As seen in Table 3 (b), there are no false positives (high precision except for matches of classes and packages in Contacts project). CRecall values ranges from 0 to 1.00. Matches with low text similarities can be identified based on the new evidence introduced in Section 4.2. We can see that naive Bayes inference framework works relatively well.

Table 3: Matching evaluation.

(a) text similarity $\geq$ 30%

	Method et al.[†]			Else[‡]		
Project	Num.*	CRec.	Prec.	Num.*	CRec.	Prec.
Browser	66/66	1.00	1.00	2/2	1.00	1.00
Contacts	162/162	1.00	1.00	10/10	1.00	1.00
Phone	102/102	1.00	1.00	6/6	1.00	1.00
ECF	894/897	1.00	1.00	125/125	1.00	1.00
EMF Compare	84/85	1.00	0.99	7/7	1.00	1.00
Xpand	178/189	1.00	0.94	2/2	1.00	1.00

(b) text similarity $<$ 30%

	Method et al.[†]			Else[‡]		
Project	Num.*	CRec.	Prec.	Num.*	CRec.	Prec.
Browser	8/79	0.88	1.00	0/13	–	–
Contacts	9/33	0.89	1.00	2/3	0	0
Phone	13/31	1.00	1.00	0/0	–	–
ECF	98/287	0.95	1.00	9/22	0.89	1.00
EMF Compare	20/38	0.70	1.00	0/2	–	–
Xpand	7/20	0.29	1.00	0/0	–	–

[†]: methods, constructors, and fields.
[‡]: classes and packages.
*: number of matches / number of match candidates.

Most of program element matches are identified well. To decrease false positives and false negatives for more improvement, there are some possible plans as follows:

- Readjust parameters of naive Bayes models.
- Find new evidences for naive Bayes models.
- Build different models for different program element types.

5.2 Performance

We show a rough comparison of performance based on reported papers. At this time, we do not compare our technique with pre-

Table 4: A rough comparison of performance

Studies	Time for one change analysis
Wu et al. [23]	a few minutes
Dig et al. [2]	several minutes
Prete et al. [20]	several seconds to a hour
Xing and Stroulia [24]	several seconds to a hour
Dagenais and Robillard [1]	several hours
This paper	**less than a second**

vious techniques with same hardware platforms and same target projects. Table 4 summarize the reported time for analyzing one change between two revisions. Note that though previous studies analyze medium-size or large-seize projects, our target projects are relatively small-size. Previous techniques require from a few minutes to several hours to analyze one change, which may be difficult to analyze entire histories (hundreds to thousands of changes). Our technique took less than a second for one change and only two seconds for entire changes of each project in Table 2. Major reason of this difference is that our technique consists of simple methods using only the information of elements (nodes), though other studies mainly use topological-information methods, which require high cost.

6. CONCLUSION

This paper presents a technique for inferring restructuring change operations on Java source code. Though previous techniques used topological information for program element matching, which have some limitations, our technique uses only node information. From empirical evaluation with six open source software projects from Android and Eclipse, it is revealed that our technique identifies program element matches with high accuracy. In addition, our technique require only a few seconds for analyzing entire histories.

With our technique, it is possible to analyze fine-grained and detail project histories. There are some possible application of this technique, such as fault-prone/failure-prone module prediction based on histories [8, 9, 19] and code clone history analysis [4, 15]. Improvements of program element matching and comparison with other techniques on same environment and same target projects are future work of our research.

7. ACKNOWLEDGMENTS

This research is supported by Grant-in-Aid for JSPS Fellows (No.23-4335).

8. REFERENCES

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

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

[3] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-aware configuration management for object-oriented programs. ICSE '07, pages 427–436, 2007.

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

[5] 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.

[6] L. Getoor and C. P. Diehl. Link mining: a survey. *SIGKDD Explor. Newsl.*, 7:3–12, December 2005.

[7] 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.

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

[9] 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.

[10] H. Hata, O. Mizuno, and T. Kikuno. Reconstructing fine-grained versioning repositories with git for method-level bug prediction. IWESEP '10, pages 27–32, 2010.

[11] H. Hata, O. Mizuno, and T. Kikuno. Historage: fine-grained version control system for java. IWPSE-EVOL '11, pages 96–100, 2011.

[12] J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support api evolution. ICSE '05, pages 274–283, 2005.

[13] H. Kashima, T. Kato, Y. Yamanishi, M. Sugiyama, and K. Tsuda. Link propagation: A fast semi-supervised learning algorithm for link prediction. SDM '09, pages 1099–1110, 2009.

[14] M. Kim and D. Notkin. Discovering and representing systematic code changes. ICSE '09, pages 309–319, 2009.

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

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

[17] M. Mailloux. Application frameworks: how they become your enemy. SPLASH '10, pages 115–122, 2010.

[18] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. ICSE '09, pages 287–297, 2009.

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

[20] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. ICSM '10, pages 1–10, 2010.

[21] J. Streit and M. Pizka. Why software quality improvement fails: (and how to succeed nevertheless). ICSE '11, pages 726–735, 2011.

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

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

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

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