Toward Reusing Code Changes

Yoshiki Higo*, Akio Ohtani*, Shinpei Hayashi†, Hideaki Hata‡, and Kusumoto Shinji*
*Osaka University, Yamadaoka 1–5, Suita, Osaka 565–0871, Japan
Email: {higo,a-ohtani,kusumoto}@ist.osaka-u.ac.jp
†Tokyo Institute of Technology, Ookayama 2–12–1, Meguro-ku, Tokyo 152–8552, Japan
Email: hayashi@se.cs.titech.ac.jp
‡Nara Institute of Science and Technology, Takayama-cho 8916–5, Ikoma, Nara 630–0192, Japan
Email: hata@is.naist.jp

***Abstract*—Existing techniques have succeeded to help developers implement new code. However, they are insufficient to help to change existing code. Previous studies have proposed techniques to support bug fixes but other kinds of code changes such as function enhancements and refactorings are not supported by them. In this paper, we propose a novel system that helps developers change existing code. Unlike existing techniques, our system can support any kinds of code changes if similar code changes occurred in the past. Our research is still on very early stage and we have not have any implementation or any prototype yet. This paper introduces our research purpose, an outline of our system, and how our system is different from existing techniques.**

***Keywords*—Change reuse, Source code analysis, Code clone**

I. Introduction

Source code reuse is one of the promising ways to develop software systems efficiently. By bringing code from other existing software systems, developers do not need to implement functions required in a new system from scratch. Moreover, reusing well-tested code realize that a new system becomes highly reliable with low cost. So far, many techniques and systems helping code reuse have been proposed and implemented [1], [2], [3].

Code reuse techniques are very useful when software systems require new functions. They help developers search available code and understand how to reuse it. However, they are not sufficient when a system requires code changes. During software evolution, source code are repeatedly changed due to various reasons such as bug fixes, function enhancements and refactorings. Some investigations revealed that the ratio of costs required after the first release of software systems are increasing gradually and over 90% of the total costs are spent after the first release [4]. Consequently, it is very important to support source code changes as well as implementations.

In this research, we are envisioning a novel system to support source code changes by reusing past code changes. Herein, change reuse means leveraging information of past code changes on a current code change, which developers are about to conduct. The purpose of our system is helping developers change code efficiently.

The following operations are required when a developer encounters a new bug in her/his software system:

- identifying which code is the cause of the bug,
- considering how to fix the identified code,
- conducting code changes, and
- testing whether the change correctly fixed the bug.

Assume that we are able to know that a similar bug had occurred in the past and then we are able to obtain the following information of the similar bug:

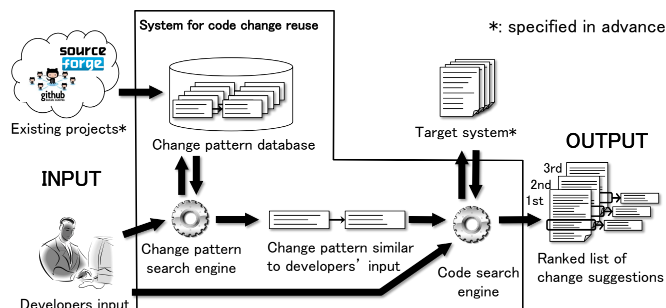

Fig. 1. Overview of proposed system

- a patch for fixing the similar bug (code before and after the bug fix), and
- test cases for checking the correctness of the bug fix.

If we have the code before the bug fix, we do not have to search the cause of the current bug by ourselves. We are able to utilize the code to search the cause [5]. We are also able to reuse the test cases for the current bug.

Code changes can be conducted more efficiently by leveraging the past code change information. Existing code reuse techniques help developers implement code for new functions meanwhile our novel system helps them change code for existing functions. Our technique is available when there is at least a past code change, which is similar to a code change that a developer is about to conduct. The authors have revealed that the same code changes are repeatedly conducted in every software system during its evolution [6]. Consequently, we are confident that our technique is useful for many code changes.

II. System for Reusing Code Changes

Figure 1 shows a system that we are envisioning. The system takes the information of a code change that a developer is about to conduct as its input. Then, the system suggests the code to be changed and how to change the code. If multiple suggestions are available, they are ranked based on their significances. The system consists of three components.

Change Pattern Database: including information of past code changes. In order to reuse them easily, they are stored into the database as patterns.

Change Pattern Search Engine: taking the information of a code change that a developer wants to conduct as its input, and searching code changes that are similar to it. Found similar code are passed to *code search engine*.

Code Search Engine: taking past code changes as its input, and searching which code of the developer's system should be changed. Identified code is suggested to developers.

In the reminder of this section, we describe the scheme of each component. Please note that the details of the components are still under consideration at this paper submission.

A. Change Pattern Database

Change pattern database includes code changes that were extracted from existing software repositories such as Git and Subversion. There should be a huge amount of code changes in this database because all the code changes are extracted. Some of them should be similar or identical to one another. If there are multiple code changes that are available at developer's context, ranking them appropriately is very important.

Consequently, we utilize a notion **change pattern**, which is a formalized pattern that shows how a code fragment was changed to another one. In this research, every code change is regarded as a change pattern, and two or more code changes are regarded as the same change pattern if they are identical or enough similar to one another.

We do not regard the whole added/deleted code in a commit as a code change because multiple code fragments could be changed in a commit. Some research studies have reported that there are commits including tangled changed in code repositories [7], [8]. We define that a code change is a correspondence between an added code chunk and a deleted one. Thus, there could be multiple code changes in a commit.

Currently, we are going to regard two code changes, c_1 and c_2, as the same pattern if they satisfy the following conditions:

$$ac(c_1) = ac(c_2) \quad \wedge \quad pc(c_1) = pc(c_2)$$

where $ac(c)$ means the code where code change c is applied, and $pc(c)$ means the changed code by the code change c.

At present, we have not decided how we represent pre/post-changed code yet. The simplest way is representing them as consecutive lines such as Unix `diff` command. An advantage of such a line-based representation is its high speed.

However, in such a representation, changes on consecutive lines are regarded as one code change. If one actual code change is conducted on multiple lines that are separated on source files, changes of them are not regarded as the same code change. Consequently, we are going adopt a graph-based representation. In this approach, a program dependence graph (in short, PDG) is constructed from the code before and after a given change on the changed method. Then, subgraphs extracted from the before and after PDGs. Extracted subgraphs consist of changed program elements. Even if multiple program elements located in separated lines are changed, they can be regarded as one code change if the elements are directly connected to each other in the PDGs.

In order to prioritize code changes when they are suggested to developers, they are characterized by using some metrics. At present, we are going to use the number of code changes belonging to a given change pattern and the number of projects where a given change pattern appear as metrics.

B. Change Pattern Search Engine

Change pattern search engine takes developers' query as its input and identifies change patterns related to it. Developers specify a code change that they want to conduct as a query. More concretely, in this research, a query should be a list of words representing a code change. For example, in cases of bug fixes, query words should be symptom of the buggy behavior or variable/method names appearing in the thrown exception. The system compares words in a given query to words in each change patterns. In change patterns themselves, user-defined names such as variable names have been replaced with special tokens to remove vocabulary differences. However, each change pattern has vocabulary information used in its original code. We are going to use vocabulary not only in changed code but also in signatures of the class and the method that include the changed code because an existing study reported that vocabulary in method's signature is the most informative [9].

C. Code Search Engine

Code search engine takes a change pattern, which is an output of *Change pattern search engine*, as its input, and it searches code that is identical/similar to the pre-changed code of the given pattern. Identified code is the locations to be changed, and the post-changed code of the given pattern is a suggestion how to change them. Searching a given piece of code from a large amount of source code is a kind of clone detections, and there exist some research studies [5], [10].

D. Scenario

Herein, we describe how developers can use our system on their change tasks. At the beginning of using our system, developers need to construct a change pattern database. When constructing a database, developers need to select existing repositories as resources of change patterns. There is no constraint for selecting repositories, but the followings should be reasonable standards to extract useful change patterns.

Same libraries: the first standard is using software repositories where the code uses the same libraries as developers' software. Existing studies reported that there are many stylized patterns to use APIs [11], [12], [13], [14]. Consequently, using such repositories will allow developers to reuse code changes that are related to API invocations for the libraries.

Same domains: the second standard is using software repositories that are the same domains as developer's software. Functions included in software systems depend on their domains, so that kinds of occurred code changes also depend on their domains. Using the same domains' software will be beneficial to extract useful code changes.

Same organization: the third standard is using software repositories that were developed in the same organizations. Organizations occasionally have their local rules for program coding. Using the same organization's software allows developers to reuse code changes that depend on such local rules.

Same developers: the fourth standard is using software repositories of the same developers. Developers learn their own implementation styles intentionally or unintentionally if they repeatedly develop the same kinds of functions. Using the same developers' software makes it possible to reuse code changes on such developer-dependent implementations.

Same software: the last standard is using the repository of the developer's software itself. This standard will be very

helpful for bug fixes caused by code clones. In other words, if a bug occurs in a code fragment that was overlooked by a developer in the past code change, our system should be able to detect it. Some existing research reported that developers occasionally overlook code fragments to be changed [6], [15]. Our system will be helpful to fix such bugs.

After a database is created, developers can use the system whenever they want. The system has been designed to help developers identify code fragments to be changed and detemine how the code fragments are changed. This system can also be used in other situations. For example, we assume that a developer has identified a code fragment to be changed by herself. In this case, if she inputs the identified code fragment to the system, she will get examples showing how to change the code fragments. She also will get other locations to be changed in the same way. Another situation where this system is useful is cases that a developer has identified and changed code fragments. In this situation, she can leverage this system to check if her changes are consistent to the changes that were conducted on similar code fragments in the past.

We are going to implement the system as a plugin of IDEs such as Eclipse and IntelliJ IDEA. Our plugin will have a text field to input queries. After receiving a query from a developer, the plugin suggests which code and how it should be changed on IDE's text editors. Integrating our system to IDEs should be very helpful for developers because they do not need to move away from IDEs for searching available past changes.

III. Preliminary Study

We have already confirmed that the pattern-based suggestion works well within a project [6]. However, in order for the proposed system to be useful, the same code changes must occur cross projects. Consequently, we conducted a preliminary study to investigate the following questions.

RQ1: How often do the same code changes occur cross different projects?

RQ2: How often do cross-project code changes[1] appear in a project?

A. Data Collection

Projects in *Apache Software Foundation* (in short, *ASF*) were selected as targets. All the *ASF* projects are being managed in a single repository[2]. Each project corresponds to a subdirectory of the root directory of the repository. We mined code changes in the past five years (from the beginning of 2010 to the end of 2014). In the period, there were 324,992 commits that changed at least one Java source file and 122 projects were changed at least once.

[1]Cross-project code change means a code change belonging to a change pattern that appears in two or more projects.

TABLE I. Detected change patterns and code changes

		Change patterns		Code changes	
		Number	Percentage	Number	Percentage
Within-project		828,616	92.8%	1,845,048	51.5%
Cross-project	2–10	59,658	6.68%	548,464	15.3%
	11–20	2,810	0.32%	172,327	4.81%
	21–30	856	0.096%	110,453	3.08%
	31–40	384	0.043%	90,026	2.51%
	41–50	189	0.021%	80,389	2.24%
	51–60	112	0.013%	69,658	1.95%
	61–70	71	0.0080%	64,250	1.79%
	71–80	50	0.0056%	91,887	2.57%
	81–90	28	0.0031%	74,971	2.09%
	91–10	22	0.0025%	120,732	3.37%
	over 100	13	0.0015%	312,603	8.73%
	subtotal	64,193	7.2%	1,735,760	48.5%

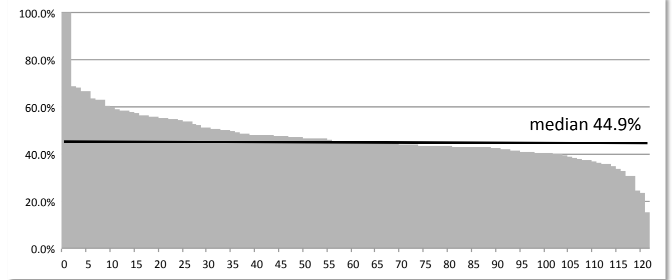

Fig. 2. Ratio of cross-project code changes for each project (the projects are sorted in the descending order of the ratio)

B. Data Analysis

In this preliminary study, we utilized our previous technique [6] to detect change patterns. It mines code changes from code repositories. In order to explain change patterns, we define program statements, code fragments, and code changes.

A **program statement** is represented as a sequence of program tokens, which are identified by lightweight source code analysis. User-defined identifiers are replaced with special tokens with parameterized matching technique, which is sometimes used in clone detection [16]. If two statements have the same sequences of tokens, they are regarded as identical.

A **code fragment** is a sequence of program statements. If two code fragments have the same sequences of program statements, they are regarded as identical.

A **code change** is a transformation from a code fragment to another code fragment. It is represented as a pair of a pre-changed code fragment and a post-changed code fragment. If two code changes have the same pre-changed and post-changed code fragments, they are treated as an identical change pattern. Even if a code change is not the same as any of the other code changes, it is treated as a change pattern.

C. Result and Discussion

Table I shows the number of detected change patterns and code changes. The first row shows within-project change patterns and code changes. There are about 830K within-project patterns, and they occupy 92.8% of all the patterns. The within-project patterns consists of 1.8M code changes, which occupy 51.5% of all the changes.

The rows from the second to the bottom show cross-project change patterns and code changes. The cross-project patterns are grouped based on the number of projects where they were found. For example, the second row shows change patterns appeared in at least 2 projects but no more than 10 projects. The table shows the more projects change patterns appear in, the less the number of them becomes. However, the number of

[2]http://svn.apache.org/repos/asf/

Code BEFORE change

```
if ( other . value ! = null ) return false ;
```

Code AFTER change

```
if ( other . value ! = null ) {
return false ;
}
}
else if ( ! value . equals ( other . value ) ) {
return false ;
```

Fig. 3. Change pattern example

code changes does not follow this trend. Each group of cross-project changes occupies at least 1% of all the changes. The bottom row shows a surprising result, 0.0015% of the patterns occupy 8.73% of the changes.

Figure 2 shows the ratio of cross-project code changes for each project. The minimum value is 15.4% for project *infrastructure* (2/13). The maximum is 100% for *kafka* (1/1) and *phoenix* (120/120). Median for the 122 projects is 44.9%. There is a tendency that the ratio of the projects including a large number of code changes is close to the median value.

Figure 3 shows an actual cross-project change pattern. Exactly the same changes occurred in 4 projects (*directmemory*, *incubator*, *myfaces*, and *portals*); the figure shows original tokens without special tokenization. This pattern included 8 code changes, the first one occurred at 19/Mar/2010 and the last one occurred at 8/Nov/2012. These changes were conducted in code where two objects are compared. The code before the change checked if the comparison target is *null* or not. However, it was insufficient as an object comparison. The code was changed to perform another comparison by using *equals* method if the comparison target was not *null*. Although this pattern includes some syntax inconsistencies due to the lack of pattern refinement, it indicates the existence of useful patterns in the extracted ones.

Our answer to the RQ1 is as follows: cross-project code changes occupy only 7.2% of all the change patterns, however they cover 48.5% of all the code changes.

Our answer to the RQ2 is as follows: all the projects include cross-project code changes. The median of the ratio of cross-project changes against all the changes is 44.9%.

The answers mean that a small number of cross-project change patterns cover about half of the code changes and all the projects include cross-project code changes. Thus, if the system can utilize cross-project change patterns appropriately, it would be able to support code changes for many projects.

IV. Related Work

Several existing studies have found some evidences that our system will be a promising technique for reducing cost of code changes. The authors reported that there are the same change patterns that are applied to source code repeatedly [6]. Nguyen et al. reported there are large number of repetitive code changes both within-project and across-project [17]. Negara et al. reported that different developers implemented code with the same change patterns when they were gave the same tasks [18]. Barr et al. found that the content of a new code often can be assembled out from the content of existing code [19].

Wang et al. proposed a technique to help developers change code fragments at multiple places without missing any one [20]. Their technique does not require the code change history of the target software. However, queres to search code fragments are relatively complex. For example, in literature [20], the following query is introduced as an example:

function A, function B, variable C, variable D; A contains "malloc" or contains "realloc", B contains "malloc" or contains "realloc"; C dataDepends A, D dataDepends B, C onestep dataDepends D, C is FieldOf D; want C.

Meng et al. developed a tool named LASE to help developers locate code fragments to be changed and determine how to change them [21]. LASE takes two or more code fragments that developers have already identified and changed. Then, it identifies other code fragments to be changed and suggest how to change them. This technique is useful after developers identified and changed two or more code fragments by themselves. On the other hand, our technique aim to help developers identify and change code fragments even if they have not identified any of the code fragments to be changed yet. Consequently, the situation for our proposed technique is different from the one for LASE.

There are some existing techniques that extract code changes by using abstract syntax trees [22], [23]. In their techniques, code changes are represented as changes on subtrees. By recognizing patterns of subtree changes, the techniques identify what a given change means. For example, renaming variable names and extracting a code fragment as a new method can be automatically identified.

Some existing research studies succeeded to generate automatically bug fix patches to a certain extent. GenProg generates candidates for a fixed program by changing a buggy program slightly with genetic programing techniques [24]. PAR uses past patches that humans wrote for fixing bugs [25]. It identifies bug fix patterns from the human-written patches and try to fix a current bug. SemFix is a tool that automatically repairs programs [26]. It derives repair constraints from test cases and solves the repair constraints to generate a valid repair. This technique can generate a repair even if its similar or identical code does not exist in the program. Those techniques require many test cases to fix target programs. Chandra et al. proposed a technique for identifying wrong conditions that cause bugs in program source code [27]. Their technique also can suggest how it should be changed.

At present, in the field of software engineering, genetic programming attracts much attention. It is used for not only bug fixes but also other purposes. For example, Langdon et al. and White et al. proposed techniques that utilize genetic programming to improve performance of human-written programs [28], [29]. There are also some techniques that automatically identify refactoring opportunities [30], [31], [32], [33].

V. Conclusion

In this paper, we introduced our system that helps developers change source code. The system utilizes past code changes in other or same software and it supports any kinds of code changes such as bug fixes, function enhancements, and refactorings. The preliminary study shows that cross-project change patterns occupy about half of code changes. Consequently, if we are able to utilize them appropriately, our system will be very helpful for developers.

REFERENCES

[1] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A Search Engine for Finding Highly Relevant Applications," in *Proceedings of the 32nd International Conference on Software Engineering*, 2010.

[2] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking Significance of Software Components Based on Use Relations," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, 2005.

[3] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding Relevant Functions and Their Usage," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[4] J. Koskinen, "Software Maintenance Costs," http://web.archive.org/web/20040202105224/http://www.cs.jyu.fi/~koskinen/smcosts.htm.

[5] Y. Higo, Y. Ueda, S. Kusumoto, and K. Inoue, "Simultaneous Modification Support Based on Code Clone Analysis," in *Proceedings of the 14th Asia-Pacific Software Engineereing Conference*, 2007.

[6] Y. Higo and S. Kusumoto, "How Often Do Unintended Incosistencies Happend? –Deriving Modification Patterns and Detecting Overlooked Code Fragments–," in *Proceedings of the 2012 28th IEEE International Conference on Software Maintenance*, 2012.

[7] K. Herzig and A. Zeller, "The Impact of Tangled Code Changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013.

[8] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto, "Hey! Are You Committing Tangled Changes?" in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014.

[9] A. Corazza, S. D. Martino, V. Maggio, and G. Scanniello, "Investigating the Use of Lexical Information for Software System Clustering," in *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, 2011.

[10] I. Keivanloo, J. Rilling, and P. Charland, "Internet-scale Real-time Code Clone Search Via Multi-level Indexing," in *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, 2011.

[11] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API Patterns As Partial Orders from Source Code: From Usage Scenarios to Specifications," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2007.

[12] R. P. L. Buse and W. Weimer, "Synthesizing API Usage Examples," in *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[13] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A Graph-based Approach to API Usage Adaptation," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2010.

[14] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and Recommending API Usage Patterns," in *Proceedings of the 23rd European Conference on Object-Oriented Programming*, 2009.

[15] L. Barbour, F. Khomh, and Y. Zou, "An Empirical Study of Faults in Late Propagation Clone Genealogies," *Journal of Software: Evolution and Process*, vol. 25, no. 11, 2007.

[16] B. S. Baker, "Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance," *SIAM Journal on Computing*, vol. 26, no. 5, 1997.

[17] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A Study of Repetitiveness of Code Changes in Software Evolution," in *Proceedings of the 28th International Conference on Automated Software Engineering*, 2013.

[18] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, "Mining Fine-grained Code Changes to Detect Unknown Change Patterns," in *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[19] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The Plastic Surgery Hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[20] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu, "Matching Dependence-related Queries in the System Dependence Graph," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2010.

[21] N. Meng, M. Kim, and K. S. McKinley, "LASE: Locating and Applying Systematic Edits by Learning from Examples," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[22] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Montperrus, "Fine-grained and Accurate Source Code Differencing," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014.

[23] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, 2007.

[24] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each," in *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[25] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic Patch Generation Learned from Human-written Patches," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[26] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program Repair via Semantic Analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[27] S. Chandra, E. Torlak, S. Barman, and R. Bodik, "Angelic Debugging," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[28] W. B. Langdon, M. Modat, J. Petke, and M. Harman, "Improving 3D Medical Image Registration CUDA Software with Genetic Programming," in *Proceedings of the 2014 Conference on Genetic and Evolutionary Computation*, 2014.

[29] D. R. White, A. Arcuri, and J. A. Clark, "Evolutionary Improvement of Programs," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 4, 2011.

[30] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and Application of Extract Class Refactorings in Object-oriented Systems," *Journal of Systems and Software*, vol. 85, no. 10, 2012.

[31] A. Han and D. Bae, "Dynamic Profiling-based Approach to Identifying Cost-effective Refactorings," *Information and Software Technology*, vol. 55, no. 6, 2013.

[32] J. Henkel and A. Diwan, "CatchUp!: Capturing and Replaying Refactorings to Support API Evolution," in *Proceedings of the 27th International Conference on Software Engineering*, 2005.

[33] N. Tsantalis and A. Chatzigeorgiou, "Identification of Move Method Refactoring Opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, 2009.