Have Java Production Methods Co-Evolved With Test Methods Properly?: A Fine-Grained Repository-Based Co-Evolution Analysis

Tenma Kita*, Hirohisa Aman†, Sousuke Amasaki‡, Tomoyuki Yokogawa‡ and Minoru Kawahara†

*Department of Computer Science, Faculty of Engineering, Ehime University
Matsuyama, Ehime, 790–8577 Japan
†Center for Information Technology, Ehime University
Matsuyama, Ehime, 790–8577 Japan
‡Faculty of Computer Science and Systems Engineering, Okayama Prefectural University
Soja, Okayama, 719–1197 Japan

***Abstract*—Any source code of a software product (production code) is expected to be tested to ensure its correct behavior. Whenever a developer updates production code, the developer should also update or create the corresponding test code to check if the updated parts still work correctly. Such a desirable co-evolution relationship between production and test code forms a logical coupling. Although the logical coupling is detectable through an association analysis on the code repository such as Git, the detection granularity is coarse because the conventional repository is at the file level. For observing those logical couplings as precisely as possible, this paper utilizes the finer-grained, Java method-level repository (FinerGit). Then the paper proposes a metric measuring the extent to which a production method has co-evolved with test methods and conducts a case study using ten open-source projects. The results show that most Java methods (98% on average) have co-evolved with test methods, but some have not; The proposed metric helps detect those methods having the potential risk that the developers might not test adequately.**

***Index Terms*—logical coupling, metric, co-evolution between production and test code, fine-grained code repository**

I. Introduction

Software products usually evolve through various source code changes, including new code additions, existing code deletions, or modifications [1]. Although source code changes are inevitable in the successful software evolution, they also risk introducing new bugs into the software product [2]. Hence, developers should test their products whenever they update the source code. To this end, when developers change the product's source code (*production code*), the developers should also create or modify the *test code* (unit test cases) corresponding to the updated production code [3], [4]. Moreover, even when developers fix bugs in production code, they should add or modify their test code because the existing test code has failed to detect those bugs. Therefore, it is ideal that any production code needs to co-evolve with the corresponding test code [5]. We can observe such a co-evolution as a *logical coupling* [6] between production and test code.

Logical couplings between two software artifacts (e.g., source files) are detectable through association analysis in the software repository [7]—*If one artifact's change appears in a commit, how likely is another artifact's change also to appear in the same commit?* The granularity of logical coupling detection depends on the unit of commits to the code repository. Since the commit unit of prevailing code repositories such as Git is a file, the logical coupling detection is also at the file level. However, because the implementation unit of a product's functionality or a test case can be finer-grained (e.g., method or function), a file-level analysis seems to be coarse and insufficient. To overcome this issue of granularity, we utilize a fine-grained code repository, FinerGit developed by Higo et at. [8], in this paper. FinerGit can convert the conventional file-level Git repository of Java software to a method-level one and allow us to analyze the logical couplings between production and test code at the Java method level.

Although there have been studies regarding fine-grained co-evolution cases (logical couplings) between production and test code [5], [9], [10], their main focuses were on understanding the evolution patterns. There may also be problematic production code that has not been co-evolved with test code in practice, and such a production code was out of scope in the above co-evolution pattern analyses because it had weak or no logical coupling with the test code and thus did not appear in the association rules. Nonetheless, to prevent unexpected regressions during software evolution, detecting such a production code having poor or no logical coupling with the test code is still significant. In this paper, to detect the above potentially problematic production code (method) automatically, we propose a metric for evaluating the logical coupling of a production method with test methods by utilizing a state-of-the-art fine-grained code repository.

II. Related Work

Luben et al. [9] emphasized the successful maintenance of test cases in test-driven development. They conducted case studies of co-evolution between production and test code using two software development projects. Their study regarded Java classes and JUnit classes as production and test code, respectively, and performed association analyses between them.

Then, they identified some logical couplings between Java production classes and JUnit test classes in those software products, but the strength of coupling differed from project to project. Although the work by Luben et al. is a valuable previous study of the production-test code co-evolution relationships, the analysis is at class-level and coarse-grained. To analyze those relationships from a finer-grained point of view, we utilize a method-level code repository in this paper.

Marsavina et al. [5] performed a fine-grained code change analysis to extract co-evolution patterns between production and test code. They utilized ChangeDistiller [11] to examine code changes in a detailed manner. Furthermore, they built (compiled) and ran the test programs for each revision of the studied software products and analyzed test coverages. Through those rich code analyses on five open-source development projects, Marsavina et al. found six primary co-evolution patterns, such as "a production class addition/removal tends to cause a test class addition/removal." Vidács and Pinzger [10] conducted a replication study of [5] and reported that they also identified the same six co-evolution patterns but the logical couplings between production and test code were weaker than that Marsavina et al. reported.

The previous studies [5], [10] yield profound insights into co-evolution relationships between production and test code through their detailed and rich change analyses. Moreover, they performed manual qualitative analyses of co-evolution patterns and found some problematic production methods (production code) that were not covered by any test code. Although poor-tested production code is out of their co-evolution pattern extractions, it is also vital to automatically detect such a problematic production code to make the developers aware of them for successful software evolution. Thus, we aim to detect problematic methods having poor or no logical coupling with the test methods by quantifying the extent to which a production method has co-evolved with test methods. We can easily evaluate logical couplings between production and test methods without any rich code analysis performed in the previous work [5], [10] since we utilize a Java-method-level code repository [8] rather than the conventional file-level repository in this paper.

III. Logical Coupling Between Production and Test Methods

This section briefly describes the logical coupling between production and test methods. Notice that our approach relies on the assumption that the developers commit their co-change events to the Git repository in a disciplined manner.

A. FinerGit: A Fine-Grained Repository For Java Methods

Higo et al. [8] developed "FinerGit," a Git-based tool that allows easy version control of Java methods. The tool converts a Git repository managing Java source files into another Git repository with the following two features.

(1) Manage a Java method as a file: FinerGit extracts Java methods from source files and stores the methods in separate files. Each file contains a single method, and the file name corresponds to the method's signature as "*className#methodSignature*.mjava." Then, the tool produces a new Git repository to represent all the past code changes regarding Java methods as "commits" of the corresponding ".mjava" files. We can explore the change history of Java methods using the conventional Git commands in the produced repository. Therefore, we can efficiently perform the co-evolution analysis of Java methods without any rich program analysis techniques.

(2) Store a method in the one-token per line style: A mjava file maintains the corresponding method in the one-token per line style. One-token per line style aids the Git repository in successfully tracking the method renames and signature changes. Git repository has a function to keep track of renaming a file at a commit. Suppose both a file deletion and a file addition occur at the same commit, and the similarity between those two files is higher than a certain threshold[1]. In that case, Git regards it as a file "renaming" rather than a pair of file deletion and addition. For evaluating the similarity between two files, the repository computes the hash value for each line (or 64-byte block) of the files and compares their hash-value distributions. However, such a Git's rename-detection mechanism is likely to miss method-renaming cases and method signature-changing cases because many small methods consist of a few lines of code. Hence, FinerGit adopts the one-token per line style to overcome such a challenge.

B. Logical Coupling Between Production and Test Methods

Suppose M is the set of production methods, and T is the set of test methods. Now we consider a co-evolution between a production method $m_p (\in M)$ and a test method $m_t (\in T)$ with the following association rule: if the developers change m_p, they also change m_t. We denote that rule by "$m_p \Rightarrow m_t$." Moreover, we denote the number of co-change commits where the developers changed both m_p and m_t, by "$\sigma(m_p \wedge m_t)$." Notice that we can track the method-renaming and signature-changing cases mentioned in Section III-A and include them in the count of co-change commits. Similarly, let $\sigma(m_p)$ be the number of commits changing m_p. Then the following value is called the *confidence* of $m_p \Rightarrow m_t$:

$$conf(m_p \Rightarrow m_t) = \frac{\sigma(m_p \wedge m_t)}{\sigma(m_p)} \ .$$

The confidence value indicates a likelihood of the co-change occurrence when m_p is changed. Hence, we can regard $conf(m_p \Rightarrow m_t)$ as a conditional probability of co-change, $P(m_t \text{ is changed} \mid m_p \text{ is changed})$.

We can detect logical couplings between methods by focusing on the confidence value. A confidence value presents the strength of logical coupling. Notice that the above logical coupling detection relies on the developers' commit manners; if they did not commit their production methods and the corresponding test methods simultaneously, we fail to detect their couplings. We need to employ an advanced detection approach to cover such a *delayed* (time-lagged) logical coupling, and it is our significant future work.

[1] Git's default threshold is 50%.

C. Evaluation Metric

Whenever a developer adds a new production method or updates an existing one, the developer should also add or edit the test method corresponding to the added/updated parts to avoid unexpected regressions. We propose a metric for evaluating the certainty that a production method's change links to one or more test methods' changes below.

Here we define the following *extended* confidence value $conf^*(m_{p,i})$ since there may be two or more corresponding test methods for one production method:

$$conf^*(m_{p,i}) = 1 - \prod_{j=1}^{|T|} \{1 - conf(m_{p,i} \Rightarrow m_{t,j})\} \ ,$$

where $m_{p,i}$ is the i-th method in our production method set (M), and $m_{t,j}$ is the j-th method in our test method set (T). $conf^*(m_{p,i})$ is the probability that a change of $m_{p,i}$ links to at least one test method's change. In other words, a low confidence value indicates that the production method has *not* co-evolved with any test methods successfully.

$conf^*(m_{p,i})$ evaluates the logical coupling of the production method $m_{p,i}$ with any test methods $m_{t,j}$ ($\forall j$). Furthermore, we also need to consider the logical coupling of the production method $m_{p,i}$ with other production methods because there may be indirect relationships between production and test methods. For example, suppose there is a production method $m_{p,k}$ calling other production methods $m_{p,k+1}$ and $m_{p,k+2}$. In that case, the developer might prepare test methods only for the callee methods ($m_{p,k+1}$ and $m_{p,k+2}$) because those test methods indirectly test the caller method ($m_{p,k}$). We propose the following metric covering both the direct and the indirect relationships between production and test methods:

$$Tconf(m_{p,i}) = 1 - \prod_{k=1}^{|M|} \{1 - conf(m_{p,i} \Rightarrow m_{p,k}) \cdot conf^*(m_{p,k})\} \ .$$

In the above equation, "$conf(m_{p,i} \Rightarrow m_{p,k}) \cdot conf^*(m_{p,k})$" presents the joint probability of the following (i) and (ii): (i) $m_{p,i}$ co-evolves with $m_{p,k}$, and (ii) $m_{p,k}$ co-evolves with at least one test method. Thus, "$1 - conf(m_{p,i} \Rightarrow m_{p,k}) \cdot conf^*(m_{p,k})$" represents the probability that $m_{p,i}$ does not co-evolve with any test methods via $m_{p,k}$. By considering the joint probability of them for all production methods, our metric $Tconf(m_{p,i})$ evaluates the likelihood that the production method $m_{p,i}$ co-evolves with at least one test method in either direct or indirect manners. In other words, a production method with a low $Tconf$ value might be problematic as it has not been co-evolved with test methods.

IV. Case Study

We conducted a case study using ten Apache top-level open-source projects (see Table I) to demonstrate how the proposed metric $Tconf$ helps detect the problematic Java methods that have not been co-evolved with test methods. We report and discuss the study results in this section.

TABLE I
Studied Projects

Project Name (Git repository)	Data Collection Period	# of Commits	# of Methods
Curator (curator.git)	2011-07-14 – 2021-12-15	2,739	3,876
Fineract (fineract.git)	2012-04-20 – 2021-12-17	6,368	21,141
Flume (flume.git)	2011-08-09 – 2022-01-14	1,832	5,105
Maven (maven.git)	2003-09-01 – 2021-12-17	11,502	7,481
Parquet (parquet-mr.git)	2012-08-31 – 2021-12-16	2,320	6,574
PDFBox (pdfbox.git)	2008-02-10 – 2021-12-20	10,539	11,768
Ranger (ranger.git)	2014-08-14 – 2021-12-17	3,760	13,979
RocketMQ (rocketmq.git)	2016-12-20 – 2022-01-19	1,955	8,742
Shiro (shiro.git)	2005-07-14 – 2022-01-15	2,210	4,088
Zookeeper (zookeeper.git)	2008-03-19 – 2021-11-27	2,371	7,498

A. Results

We extracted 71,730 Java production methods and 18,522 test methods from the studied projects; we regarded the Java methods in Java source files whose paths include "test" as the test methods and considered the remaining ones as the production methods. Then, we computed $Tconf$ values for each production method. As a result, most values were approximately 1.0, and the average $Tconf$ values were 0.9557 – 0.9996 for all projects[2].

Table II presents the percentage of methods classified by three categories: "$Tconf \simeq 1$," "$0.5 \leq Tconf \leq 0.999$," and "$0 \leq Tconf < 0.5$," where we regard "$0.999 < Tconf \leq 1$" as $Tconf \simeq 1$ by considering round-off errors of floating point numbers in our $Tconf$ computation. As a result, about 98% of production methods show $Tconf \simeq 1$ on average. Those results indicate that most production methods have co-evolved with test methods in the ten studied projects.

However, Table II shows that some production methods have low $Tconf$ values, i.e., they have not co-evolved with test methods. Here, we focus on the production methods whose $Tconf$ values are less than 0.5; Since the co-change probability is less than 0.5, such a production method is not likely to co-evolve with test methods, and it might be a problematic method from the viewpoint of testing. The percentages of such problematic methods were less than 1% in nine out of ten projects (see Table II). Only the PDFBox project showed a different tendency from the others, and the percentage of such production methods was 3.87%.

B. Discussions

We conducted a case study using ten open-source software development projects to observe how the proposed metric $Tconf$ works to detect problematic production Java methods from the viewpoint of the co-evolution relationship between production and test methods. As a result, we found that the $Tconf$ values of about 98% of production methods are approximately equal to 1, i.e., those production methods have co-evolved with one or more test methods. Thus, we can say that

[2]The set of experimental data obtained in this study is available at our supplemental data site https://bit.ly/3Ekurdc.

TABLE II
PERCENTAGES OF PRODUCTION METHODS CLASSIFIED BY $Tconf$ VALUES

Project Name	$Tconf < 0.5$	$0.5 \leq Tconf \leq 0.999$	$Tconf \simeq 1$
Curator	0.43%	0.64%	98.93%
Fineract	0.04%	0.04%	99.92%
Flume	0.27%	0.59%	99.14%
Maven	0.76%	1.94%	97.30%
Parquet	0.13%	0.04%	99.82%
PDFBox	3.87%	7.65%	88.49%
Ranger	0.46%	1.92%	97.62%
RocketMQ	0.29%	0.12%	99.59%
Shiro	0.03%	0.23%	99.74%
Zookeeper	0.37%	0.13%	99.49%
Average	0.67%	1.33%	98.00%

TABLE III
NUMBER AND PERCENTAGE OF PRODUCTION METHODS WITH $Tconf < 0.5$ BEFORE AND AFTER FILTERING THE SIMPLE ONES OUT

	# of Methods (Percentage)	
Project Name	Before Filtering	After Filtering
Curator	12 (0.43%)	0 (0.00%)
Fineract	7 (0.04%)	0 (0.00%)
Flume	7 (0.27%)	0 (0.00%)
Maven	44 (0.76%)	1 (0.02%)
Parquet	6 (0.13%)	0 (0.00%)
PDFBox	403 (3.87%)	118 (1.14%)
Ranger	55 (0.46%)	6 (0.05%)
RocketMQ	20 (0.29%)	3 (0.04%)
Shiro	1 (0.03%)	0 (0.00%)
Zookeeper	17 (0.37%)	0 (0.00%)

the developers have successfully developed and maintained most Java production methods with the test methods in the studied projects.

Nonetheless, we also found some production methods with low $Tconf$ values. For example, $Tconf$ value of method "PDPropBuildDataDict#public_String_getVersion" in the PDFBox (Fig. 1) was zero. We explored the method-level repository and noticed it appeared in the commit history only once. That is, the method has not been changed after its creation. Although that method was a simple "getter" method, as shown in Fig. 1, it would be better to prepare the corresponding test method. However, some developers might not make unit test cases for such simple methods in practice because they look less likely to be buggy.

We take the existence of simple production methods like Fig. 1 into account and filter out the production methods satisfying both of the following two conditions: (i) it has appeared in the commit history only once, and (ii) it has the lowest cyclomatic complexity [12], i.e., it contains no branch statement (e.g., `if`, `for`, `while`); The lowest cyclomatic complexity (=1) indicates that the method has a single execution path and is the most straightforward code under test. Table III shows the number and percentage changes of the problematic production methods whose $Tconf$ values are less than 0.5 before and after filtering the simple methods out.

As shown in Table III, all production Java methods with $Tconf < 0.5$ disappeared in six out of ten projects through the above filtering, and a few stayed in three out of the remaining four projects. Although all projects showed a significant decrease in such methods, only the PDFBox project still has many (=118; 1.14%). We found a Java production method "addPath(PDAppearanceContentStream,GeneralPath)" of class "PDTextAppearanceHandler" in package "org.apache.pdfbox.pdmodel.interactive.annotation.handlers," whose LOC = 44 and cyclomatic complexity = 7. While this method has appeared in the commit history twice, it has not been co-evolved with any other production or test methods, i.e., its $Tconf$ value is zero. Since the method is not simple in terms of LOC and cyclomatic complexity, it is ideal to co-evolve with one or more corresponding test methods during the development. Nevertheless, it can be a potential problem that the method also has *no* (*indirect*) *logical* coupling to any test methods ($Tconf = 0$). Although developers might test such a method in another way, we consider it is worth it to alert the presence of such a production method and make the developers aware of the risk of missing the method's test to prevent unexpected regressions.

```
public String getVersion()
{
   return dictionary.getString("REx");
}
```

Fig. 1. Source code of PDPropBuildDataDict#public_String_getVersion.

Finally, we discuss the differences among projects from the viewpoints of the production-test method ratio and the number of developers. Table IV compares projects using two metrics: PTR (Production-Test method Ratio),

$$\text{PTR} = \frac{\text{number of production methods}}{\text{number of test methods}},$$

and NDEV (Number of DEVelopers). The higher PTR value indicates fewer test methods for the production ones in the project. As the PDFBox project has the second-highest PTR value, following the Fineract project (see Table IV), these two projects prepare fewer test methods for the number of their production methods than the other projects. However, NDEV values differ substantially between these two projects: the NDEV value of the Fineract project is 164, and that of the PDFBox project is only 6. Hence, the testing of production methods may depend on a small group of developers, and there would be a higher risk of overlooking regressions during maintenance activities.

TABLE IV
A COMPARISON OF PROJECTS USING PTR AND NDEV

Project Name	PTR	NDEV
Curator	2.6	108
Fineract	8.3	164
Flume	1.0	55
Maven	3.5	151
Parquet	2.2	166
PDFBox	7.8	6
Ranger	5.9	88
RocketMQ	3.6	320
Shiro	5.0	56
Zookeeper	1.5	163

Notice that a low $Tconf$ value does not directly indicate the poor quality of the production method. Nonetheless, the proposed metric $Tconf$ can detect the methods that have not been successfully co-evolved with test methods. By alerting such potentially problematic production methods to the developers, $Tconf$ can contribute somewhat to the successful software evolution. We plan to validate the usefulness of our metric by asking the developers for feedback in the future.

C. Threats to Validity

1) Construct Validity: We proposed the metric $Tconf$ to measure how a Java production method has properly co-evolved with one or more test methods. Our measurement is based on the co-change events (commits) of Java methods observed in the code repository. Thus, the fundamental factor is the number of commits in which both the production and the test method changed. However, there might be cases where the test code's commits occurred *after* the production code's commits [13]. Such a *delayed* logical coupling is a threat to the construct validity in this study, and we need further investigation regarding it in the future.

2) Internal Validity: Because we often see the commits involving changes in methods' names and signatures, it is crucial to accurately track the renaming commits in the method-level repository. In other words, the accuracy of the rename-commit tacking is a threat to the internal validity of this study. To mitigate the threat, we utilized the state-of-the-art tool (FinerGit) that outperforms the conventional one (Historage [14]) regarding rename tracking accuracy. However, if there was a renaming commit with many code changes, we might mistakenly treat it as a pair of method addition and deletion. Furthermore, some changes in production methods might be non-functional changes such as refactoring or comment changes. As such changes do not require test code changes, our metric may make some false-positive alerts. We need to employ a richer code-change analysis to control such false-positive cases, and it is our significant future work.

3) External Validity: Our case study examined ten Apache top-level projects. Although they are popular and well-managed projects, they cannot become the representative ones reflecting all Java software developments. Different projects driven by other organizations or individuals may have different styles of managing their test methods. Especially, it would be an impactful factor if they adopt the test-driven development or not. Moreover, our study did not investigate projects with development languages other than Java. The above things can be threats to the external validity of this study.

V. Conclusion and Future Work

Whenever a production method is changed, the corresponding test methods are ideal to be updated or created to test the modified code at the same commit. Such co-evolution relationships ensure successful software maintenance and evolution. Thus, we focused on the co-evolution (logical coupling) between production and test methods and utilized FinerGit to track method-level code change history in this paper. Then, we proposed $Tconf$ as a metric for evaluating how a Java production method has properly co-evolved with test methods.

Through a case study of $Tconf$ using ten open-source Java software development projects, we showed most production methods (about 98% on average) had been co-evolved with test methods successfully, but some methods were not; We could detect the potential problematic production methods that have not been co-evolved with any test method using the proposed metric $Tconf$. Although it does not directly imply those production methods are poor quality, it is worth it to make the developers aware of the potential risk that they might not test those methods adequately. The proposed metric $Tconf$ is helpful to alert such production methods.

Our future work includes (1) a validation study of $Tconf$ by getting feedback from the developers, (2) an empirical study of the relationship between $Tconf$ value and the overlooked faults, and (3) further analyses focusing on the code change details and the *delayed* logical couplings where the test method's commit occurs after the production one's commit.

Acknowledgment

This work was supported by JSPS KAKENHI #20H04184, #21K11831, and #21K11833.

References

[1] I. Sommerville, *Software Engineering*, 10th ed. Essex: Pearson Education, 2016.

[2] C. Jones, *Applied Software Measurement: Global Analysis of Productivity and Quality*, 3rd ed. New York: McGraw-Hill, 2008.

[3] S. Elbaum, D. Gable, and G. Rothermel, "The impact of software evolution on code coverage information," in *Proc. IEEE Int'l Conf. Softw. Maintenance*, Nov. 2001, pp. 170–179.

[4] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen, "Mining software repositories to study co-evolution of production amp; test code," in *Proc. 1st Int'l Conf. Softw. Testing, V. & V.*, Apr. 2008, pp. 220–229.

[5] C. Marsavina, D. Romano, and A. Zaidman, "Studying fine-grained co-evolution patterns of production and test code," in *Proc. 14th IEEE Int'l Working Conf. Source Code Analysis & Manipulation*, 12 2014, pp. 195–204.

[6] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proc. Int'l Conf. Softw. Maintenance*, Nov. 1998, pp. 190–198.

[7] T. Zimmermann, P. Weibgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proc. 26th Int'l Conf. Softw. Eng.*, Jul. 2004, pp. 563–572.

[8] Y. Higo, S. Hayashi, and S. Kusumoto, "On tracking java methods with git mechanisms," *J. Syst. Softw.*, vol. 165, pp. 110 571:1–13, Jul. 2020.

[9] Z. Lubsen, A. Zaidman, and M. Pinzger, "Using association rules to study the co-evolution of production amp; test code," in *Proc. 6th IEEE Int'l Working Conf. Mining Softw. Repo.*, May 2009, pp. 151–154.

[10] L. Vidács and M. Pinzger, "Co-evolution analysis of production and test code by learning association rules of changes," in *Proc. 2018 IEEE Workshop Machine Learning Tech. for Softw. Q. Eval.*, Mar. 2018, pp. 31–36.

[11] B. Fluri, M. Wursch, M. PInzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 725–743, Nov. 2007.

[12] T. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[13] A. Zaidman, B. V. Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empr. Softw. Eng.*, vol. 16, no. 3, pp. 325–364, Jun. 2011.

[14] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *Proc. 34th Int'l Conf. Softw. Eng.*, May 2012, pp. 200–210.