Detecting and Modeling Method-level Hotspots in Architecture Design Flaws

Ran Mo, Shaozhi Wei, Ting Hu, Zengyang Li
School of Computer Science & Hubei Provincial Key Laboratory of Artificial Intelligence and Smart Learning
Central China Normal University
moran@mail.ccnu.edu.cn, wsz@mails.ccnu.edu.cn, 826473959@qq.com, zengyangli@mail.ccnu.edu.cn

***Abstract*—In large-scale software systems, the majority of change-prone files are usually architecturally connected, and their architectural connections often exhibit design flaws, which propagate change-proneness among files and increase maintenance costs. Complementary to the identification and definition of the files involved architecture design flaws, the automatic detection at the method level is important for developers to understand fine-grained contributors to the architecture flaws that have incurred high maintenance costs. In this paper, we propose an approach to identify method-level hotspots. Each hotspot contains a group of evolutionarily connected methods or attributes that participate in architecture design flaws and continuously accumulate maintenance difficulties. Our investigations on six large-scale projects show that the attributes or methods captured in method-level hotspots are more change-prone than the others. The results also show that the growth trend of the maintenance effort spent on each method-level hotspot could be modeled and monitored by our approach, which sheds light in the decision of design refactoring.**

***Index Terms*—Software Architecture, Software Maintenance, Fine-grained Code Change**

I. Introduction

Software architecture has significant impacts on the maintenance of a software project. Numerous studies have been proposed to investigate the software quality by examining software architecture. For example, prior studies of Xiao et al. [18] and Kazman et al. [13] have shown that the majority of change-prone files are architecturally connected, and design flaws in these connections could propagate change-proneness among files. Consequently, it is difficult to eliminate files' change-proneness without resolving the architecture design flaws among them. The automatic detection of architecture design problems has also been studied. Xiao et al. [19] presented four types of architecture debts which have significant and long-term impact maintenance costs over time. Mo et al. [14, 15] formally defined and automatically detected a suit of architecture design flaws, which have a significant impact on files' change-proneness.

However, all of these studies are conducted at the file level, none of them investigated method-level participants of architecture design flaws. After identifying the files involved in architecture design flaws, it is still worth for developers to explore which attributes and methods are responsible for the change-proneness of these involved files, especially, for the files with hundreds of attributes or methods. This could raise questions for developers or architects: Which attributes or methods of the flawed files need to be fixed for maintenance and refactoring? Can these '*problematic* attributes or methods be detected automatically? How do the involved attributes or methods evolve over time?

In this paper, we formally define the concept of *Method-level Hotspot*: a group of evolutionarily coupled attributes or methods that participate in architecture design flaws and continuously accumulate maintenance costs. To detect a *method-level hotspot*, 1) our approach first extracts the attributes or methods from the files involved in an instance of architecture flaws that identified by the techniques in [14]; 2) based on these extracted attributes or methods, and the fine-grained history output by *ChangeDistiller* in [3], our approach will automatically identify a *method-level hotspot* in a project. Given a *method-level hotspot*, our approach will automatically model its evolution trend in terms of maintenance effort spent on its involved attributes and methods. We use three typical regression models, linear, exponential and logarithmic models, to model each hotspot' evolution to monitor whether the hostspot has been accumulating maintenance costs steadily, dramatically or slowly.

We have validated the effectiveness of our approach by using six large-scale projects. According to the evaluation results, we have found that: 1) attributes or methods involved in method-level hotspots are more change-prone than the other attributes or methods; 2) most of the detected method-level hotspots could be modeled by one of the three typical regression models, the modeling results are useful for developers to understand the evolution trend for each hotspot.

The rest of this paper is organized as follows: Section II presents the background concepts. Section III describes the definition of the method-level hotspots. Section IV presents our evaluation methods and results. Section V discusses. Section VI shows related work and Section VII concludes.

II. Background

We now introduce the basic concepts and techniques behind our work.

DOI reference number: 10.18293/SEKE2020-051

A. Architecture Design Flaws

In [14], the authors defined and validated a suite of hotspot patterns, the recurring architectural design flaws in software systems. They presented that files involved in the detected design flaws are really change-prone. Using the techniques in [14], we identify four file-level architecture design flaws: 1) *Unstable Interface* – a highly influential file have a large number of dependents and changes frequently with many of its dependents in the revision history; 2) *Modularity Violation* – the structurally independent files frequently change together as recorded in the project's revision history; 3) *Unhealthy Inheritance* – a super class depends on its sub classes or a client class depends on both a super-class and its sub classes; 4) *Clique* – a group of files forming a strongly connected component are tightly coupled with cyclic dependency. In general, each instance of the architecture design flaws capture a group of files, and these detected files have caused high maintenance costs. From each group of flawed files, we could extract the attributes and methods that participate in each architecture design flaw.

B. Method-level Changes

Revision history records maintenance activities of a project. By examining the change history of attribute and methods, we could get the data of evolutionary relations among attributes or methods and investigate how they evolved. Following the techniques in [3], we extract fine-grained changes from a project's revision. Furthermore, we categorize these fine-grained changes into eight types of *method-level change operations* on attributes or methods:

- *ATTRIBUTE_ADRT_CHANGE*: Adding, Deleting, Renaming an attribute or changing the type of an attribute.
- *ATTRIBUTE_MODIFIER_CHANGE*: Changing the modifier of an attribute, such as changing the accessibility of an attribute or finalizing an attribute, etc.
- *METHOD_ADR_CHANGE*: Adding, Deleting or Renaming a method.
- *METHOD_MODIFIER_CHANGE*: Change the modifier of a method, such as changing the accessibility of a method or finalizing a method, etc.
- *METHOD_PARAMETER_CHANGE*: Adding, Deleting, Rename parameters, or changing the type or ordering of parameters.
- *METHOD_BODY_CHANGE*: Change the body of a method.
- *METHOD_RETURN_CHANGE*: Adding, Deleting or Changing the type of a method return.
- *DOCUMENT_CHANGE*: Adding, Deleting or Updating the documentation of an attribute or a method.

Using the suite of *method-level change operations*, we capture how an attribute or method was changed, and by how many times.

III. Identification and Modeling

To investigate method-level hotspots, our approach proceeds as follows: 1) identifying *method-level hotspots*; 2) modeling the evolution of identified method-level hotspots; 3) visualizing the identified method-level hotspots.

A. Method-level Hotspot Definition

Using the concepts and technique in [14], a set of file groups flawed by architecture flaws will be detected from a project. Although the attributes or methods in each file group participate in an architecture flaw, not all of them contribute to incurring maintenance costs into the group. For each file group (an instance of architecture flaw [14]), we aim to identifying their attributes or methods that propagate changes among the group and cause maintenance costs. Therefore, we define a *Method-level Hotspot* to be a group of *evolutionarily coupled* attributes or methods that participate in architecture design flaws and continuously accumulate maintenance difficulties. The rationale is, first, if an attribute or method could always be changed independently, then we consider this attribute or method is not coupled with others in the group, changes to it won't influence the others; second, a hotspot should continuously cause maintenance costs in the system. We don't need to worry about the methods or attributes that are inactive with respect to changes, because they won't cause maintenance costs in the future.

Based on the definition, we formally calculate a method-level hotspot as a sequence of tuples :

$$M - hotspot = (\langle mSet_1, mSetCost_1\rangle, \langle mSet_2, mSetCost_2\rangle \\ , ..., \langle mSet_m, mSetCost_m\rangle) \tag{1}$$

where m is the number of history periods the hotspot has been evolved through.

B. Method-level Hotspot Identification

Given an instance of architecture flaw, let F to be the set of involved files, and M to be the universal set of all attributes and method within these files. To identify the group of evolutionarily coupled attribute or methods in *M*, we checked the co-changes between the attributes or methods in M by mining a particular period of revision history. If an attribute or method hasn't changed together with any other attributes or methods in M during the period of history, we consider it is evolutionarily independent to the others in M. Assume $mSet_k$ is the maximal group of evolutionarily coupled attributes or methods in M, then $mSet_k$ contains a subset of attributes and methods in M, and these attributes or methods should satisfy:

$$mSet_k : \forall m_i \in mSet_k, \exists m_j \in mSet_k | cochange(m_i, m_j) \tag{2}$$

where $i \neq j$, $i, j = [1, 2, 3, ..., n]$, n is the number of attributes or methods in the $mSet_k$. k means the k^{th} period of history. $cochange(m_i, m_j)$ means m_i and m_j have been changed together in the same commits. The co-changes between m_i and m_j are calculated from the given $period_k$ of revision history.

Based on each M of an architecture flaw instance, we detected the *method-level hotspot* consisting of a sequence of $mSet$ and $mSetCost$ by using different history periods. In this paper, we back-forwardly decreased the history period by a 6-month history interval. For example, if we have detected an architecture flaw instance from a version of project A, which was released in 2016-07 and its history began in 2015-09. Let the universal set of attributes and methods involved in this instance be M_a, to construct the hostpot, we calculated its $mSet$ sequence by using four history periods: 2015-03 - 2016-07, 2015-03 - 2016-01, and 2015-03 - 2015-07. The number of $mSet$ in the hotspot may be less than 3, because attributes and methods in M may not change together at the early history. Besides, we also calculated the maintenance costs spent on the attributes or methods in each $mSet$ to be $mSetCost$ by using the same history period.

We consider a *method-level hotspot* that has been accumulating maintenance costs, if two conditions are satisfied: 1) the hotspot has been involved for an enough long time. In this paper, since all of the projects have a long history, we required that a hotspot should have evolved for 3 years, and we sampled six months as the history interval; 2) attributes or methods involved in the hotspot should continuously incur maintenance costs. Let $\langle mSet_1, mSetCost_1\rangle$, and $\langle mSet_n, mSetCost_n\rangle$ be the first and last elements in a method hostopt, $mSetCost_n$ should be larger than $mSetCost_1$, where $mSetCost_i$ means the total number of changes made on the attributes or methods in $mSet_i$ by using corresponding history period.

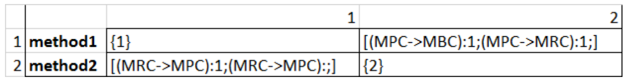
C. Method-level Hotspots Modeling

Given a *method-level hotspot*, our approach automatically models its evolution in terms of maintenance costs. In this way, we could present the variation trend of these method-level hotspots in terms of maintenance costs. Our approach uses the sequence of mSetCost as an input, and searches the best regression model for it. Since the selected history periods are cumulative, we applies three typical regression models in our approach, each type of regression models presents a different maintenance evolution of method-level hotspots:

- *Linear model*, which describes the method-level hotspot accumulates maintenance costs steadily over time.
- *Exponential model*, which describes the method-level hotspot accumulates maintenance costs dramatically over time.
- *Logarithmic model*, which describes the method-level hotspot accumulates maintenance costs slowly over time.

D. Method-level Hotspots Visualization

For each *method-level hotspot*, we proposed a *Method-level DSM* (M-DSM) to model each of its $mSet$. A M-DSM is extended from the DSM, which is proposed by [1]. A DSM is a square matrix whose rows and columns are labels with the same elements in the same order. A cell in a DSM present the structure or co-change relations between an element in row and an element in column. Elements in the original DSM could be files, classes or packages. We used the M-DSM to present the attributes or methods and their relations. Instead of presenting the co-change numbers in a cell, the cells in a M-DSM show *method-level co-change pairs*. We define a *Method-level Co-change Pair* to be a pair of change operations introduced in Section II. Each *co-change pair* shows how two attributes or methods changed together and the number of its occurrence. Considering a M-DSM contains 2 method, $method1$ and $method2$, 1) these two methods were both involved in a commit, where $method1$'s parameter and $method2$'s body were changed; 2) these two methods were involved in another commit, where $method1$'s parameter and $method2$'s return type were changed. Then, the method-level DSM will be shown as in Figure 1.



		1	2
1	**method1**	{1}	[(MPC->MBC):1;(MPC->MRC):1;]
2	**method2**	[(MRC->MPC):1;(MRC->MPC):;]	{2}

Fig. 1: Example of M-DSM showing co-change pairs
MRC: METHOD_RETURN_CHANGE;
MPC: METHOD_PARAMETER_CHANGE;
MBC: METHOD_BODY_CHANGE

IV. Evaluation

In this section, we report our evaluation subjects, methods and results.

A. Research Question

To evaluate the effectiveness of our approach, we investigate the following research questions:

RQ1: *Are the attributes or methods involved in method-level hotspot really change-prone?*
A positive answer to this question would demonstrate that the detected method-level hotspot really capture the change-prone attributes and methods in the project, which deserve more actions for refactoring.

RQ2: *Could we model the evolution of detected method-level hotspots?* To answer this question, we are attempting to investigate whether the evolution of method-level hotpots could be effectively monitored. A positive answer to this will enable us to understand how the detected method-level hotspts evolve over time.

B. Subjects

Six Apache open-source projects have been chosen as the subjects, which differ in size, domain and other project characteristics: Camel is an integration framework; Cassandra is a distributed NoSQL database management system; CXF is a services framework; Hadoop is a tool for distributed Big Data processor; OpenJPA is a Java persistence project; PDFBox is a library for manipulating PDF documents.

We list the basic facts for each studied project in Table I. The column “#Members” presents the total number of attributes and methods in a project. The column “#Commits” presents the number of revisions over the time period from the begin to the selected release date for each project. All projects' revision histories are extracted from GitHub[1]. The column “#History Length” shows the number of months from the begin to the selected release date.

[1]https://github.com/

For each project, we first obtained its source code and chose its stable version as our research subject. Then we used Understand[2] to generate a file dependency report. Given the revision history and the file dependency file, we used the toolset in Mo et al.'s work [14] to detected architecture flaws. We used *ChangeDistiller* [3] to extract the method-level revision history. Given all the flaws and method-level history as inputs, our tool automatically detects all *method-level hotspots* and fits them into regression models.

TABLE I: Researched Projects

	Release	#Members	#Commits	#History Length
Camel	2.15.5	84,734	24,933	113 months
Cassandra	2.1.13	41,767	19,333	89 months
CXF	3.0.9	54,210	11,160	96 months
Hadoop	2.6.3	70,054	12,506	86 months
OpenJPA	2.4.1	25,896	4,729	123 months
PDFBox	1.8.10	19,236	4,337	102 months

C. Evaluation Results

To quantify the maintenance effort, we use a typical history measures: *Change Frequency (CF)*, the number of times an attribute or method has been changed in commits with a given period of revision history.

RQ1: *are the attributes or methods methods captured in method-level hotspots notably change-prone?*

For each project, we calculated the average change frequency (avg_CF) values for all the attributes or methods involved in *method-level hotspots* and the attributes or methods not involved in any *method-level hotspot*. Table II reports the comparison results between two sets of average values. "avg_CF" means the average CF for attributes or methods involved in method-level hotspots, "avg_nCF" means the average CF for the other attributes or methods. "Inc." means by how much the average value has increased by comparing the attributes or methods in hotspot to the attributes or methods not in hotspots. We calculated the Inc. of change frequency as:

$$Inc. = (avg\_CF - avg\_nCF)/avg\_nCF \times 100\% \quad (3)$$

From Table II, we can observe that all the avg_CF values are larger than the avg_nCF. The greatest increase happens in the *CXF* project. 147% means that, in this project, the attributes or methods involved in method-level hotspots were changed twice more often than the attributes or methods which are not involved in any hotspot. The smallest increase is still as high as 74% in the *Cassandra* project.

To rigorously validate this observation, we employ the *Wilcoxon signed-rank test*, a non-parametric statistical hypothesis test for comparing two related samples, to test whether the population of avg_CF is significantly larger than the population of avg_nCF over the six projects. We defined the hypotheses as follows:

[2]https://scitools.com/

TABLE II: Comparison between avg_CF and avg_nCF

	avg_CF	avg_nCF	Inc.
Camel	1.94	0.85	129%
Cassandra	2.92	1.68	74%
CXF	2.18	0.88	147%
Hadoop	1.98	1.05	89%
OpenJPA	1.85	0.78	138%
PDFBox	2.29	1.17	95%

Null Hypothesis: H0, the population of avg_CF is not significantly larger than the population of avg_nCF.

Alternative Hypothesis: H1, population of avg_CF is significantly larger than the population of avg_nCF.

The p-value of this test is less than 0.05, thus H1 is accepted. The results indicate that there exists statistically significant differences between avg_CF and avg_nCF over all 6 projects. Therefore, we can claim that that attributes or methods captured in method-level hotspots will be more change-prone, and hence cause higher maintenance costs in a project.

RQ2: *Could we model the evolution of detected method-level hotspots?*

A *method-level hotspot* captures a group of evolutionarily coupled attributes or methods that accumulates maintenance costs. We investigate this problem to answer whether we can monitor the growth trend of the maintenance effort spent on each *method-level hotspot* over time.

In this paper, we fit a hotspot's growth trend of maintenance costs to one of the three models: linear, exponential and logarithmic regression models, which indicating the attributes or methods in a method-level hotspot accumulates maintenance costs steadily, extremely fast or slowly respectively. For each method-level hotspot, the regression model with highest R^2 will be selected to be the best fit for it. Besides, the P-value of each fitting model should be less than 0.05, which guarantees that the derived model is significant.

Following the guidelines in the work of [10, 12], where the authors described R^2 = 0.75, 0.5 and 0.25 as substantial, moderate and weak models, respectively, we summarized the fitting results in Table III. Column "#Hotspot" means the total number of *method-level hotspots* detected from a project. Columns "Lin", "Exp" and "Log" present the number of hotspots that fit into linear, exponential or logarithmic models respectively. The following "Pt." columns show the ratios to the total number of detected hotspots. In the column of "$0.5 \leq R^2 < 0.75$", "Num" means the number percentage of hotspots fitted into a regression model with a R^2 from 0.5 to 0.75. The last two columns shows the number and percentage of hotspots which couldn't or weakly fit into one of the three regression models.

Using "Camel" as an example, we can see that there are 907 method-level hotspots were detected from this project, and 69% of these hotspots can be substantially modeled by the regression models ($R^2 \geq 0.75$). 48% of all detected hotspots could fit into a linear model, means these hotspots accumulate maintenance costs steadily. 8% and 13% of all detected hotspots could fit into the exponential and logarithmic models respectively. Only 69 hotspots, 8% of all detected hotspots, couldn't fit into a substantial or moderate regression model.

The last row of Table III presents that, considering all the detected method-level hotspots over all projects together, 67% of them could be fitted into a substantial regression model. 52% of all hotspots could be modeled by linear models. For both exponential and logarithmic models, there are 7% of all hotspots follow a substantial fitting. Only 8% of all the detected hotspots can not or weakly fit into a model. In summary, our approach could model the growth trend of maintenance costs for each hotspot..

Figure 2 shows the example of a linear fitting. We can observe that the selected release date of this project is "2015-11", and attributes or methods in this hotspot started to be co-changed before 2010-07, but after 2010-01, since the history interval is 6 months. The maintenance costs trend of this hostspot is fitted into a linear model, which has a $R^2 = 0.98$, with a formula as: $y = 64.2x + 88.6$.

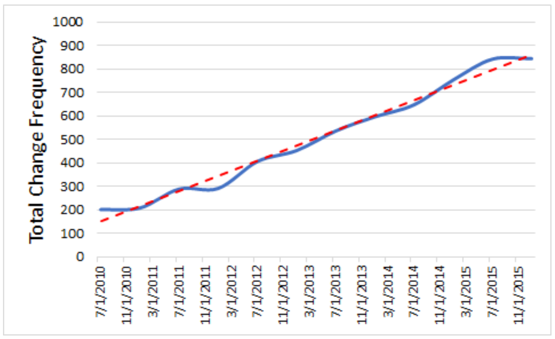


Fig. 2: Hotspot fitted into a linear model
$R^2 = 0.98$; *Formula:* $y = 64.2x + 88.6$

D. Results Summary

Based on the evaluation results, we can positively answer our research questions as follows:

RQ1: If an attribute or method is involved in a *method-level hotspot*, it is more likely to have higher change-proneness.

RQ2: The majority of detected method-level hotspots could be modeled by one of the three regression models, the modeling results will help developers to understand the evolution trend for each hotspot.

V. Discussion

A. Threats to validity

First, we can not guarantee that change frequency is the best proxies for maintenance effort. In our future work, we intend to use more proxy measures for our analysis, for example, the bug frequency, the time frame for each issue, the budget spent, etc. Thus we could further demonstrate the effectiveness of our approach and tool.

Second, we only applied our research on six Apache open source projects, hence we can not claim that our results are generalizable across all software projects. However, we chose projects with different sizes and domains to partially address this issue. A larger study employing more projects and more metric types would improve the validity of our conclusions.

Third, the detection of our *method-level hotspot* needs to mine the project's revision history. We need to examine the co-changes between attributes or methods. The availability and accuracy of the method-level history information heavily depends on the project's protocols.

Finally, since the history periods are accumulative, we only used three typical regression models to model the growth trend of maintenance costs. We can not grantee the completeness of our applied models. But our tool and our approach are scalable, which enables easy additions for new regression models.

B. Future Work

We are planning to apply our approach on more projects to further demonstrate the effectiveness of our approach and detection tool. We also plan to investigate whether the *method-level DSM* could help to examine underlying problems in a project and help to explore the refactoring opportunities for each *method-level hotspot*, such as splitting files, combining methods, etc.

VI. Related work

In this section we compare our approach with the following research areas.

Defect Prediction and Localization: There are numerous studies [2, 5, 8, 9, 11] aimed at predicting and locating error-prone/change-prone files by using file metrics, file change history, or both. For example, Jones et al. [11] obtained the ranking information of each statement and used the information to assist fault location. Nagappan et al. [16] studied different complexity metrics and demonstrated that a combination of these metrics are useful predictors for defects and successful for defect prediction. Cataldo et al. [2] investigated the density of change coupling and showed that it correlated with failure proneness. Ostrand et al. [17] demonstrated that a combination of files metrics and file change history can be used to effectively predict defects.

However, all these studies treat the error-/change-prone files individually but don't consider the architectural connections among these files. Consequently, even if the predicted files were modified, the root causes of high-maintenance costs would still exist there, because architecture design flaws haven't been eliminated. Our study focuses on the attributes or methods participating in architecture design flaws.

Code and Architecture Quality: Gamma et al. [6] introduced commonly occurring software design problems. They presented design patterns as proven solutions to these recurring problems. Fowler [4] introduced the concept of a "bad smell" to identify code problems and provide refactoring references.

TABLE III: Distribution of Architecture flaws' Regression Models

		$R^2 \geq 0.75$								$0.5 \leq R^2 < 0.75$		Other	
Project	#Hotspot	Lin	Pt.	Exp	Pt.	Log	Pt.	Total	Pt.	Num	Pt.	Num	Pt.
Camel	907	437	48%	77	8%	116	13%	630	69%	208	23%	69	8%
Cassandra	737	477	65%	15	2%	16	2%	508	69%	209	28%	20	3%
CXF	712	344	48%	69	10%	43	6%	456	64%	167	23%	89	13%
Hadoop	924	668	72%	63	7%	40	4%	771	83%	121	13%	32	3%
OpenJPA	544	129	24%	15	3%	12	2%	156	29%	289	53%	99	18%
PDFBox	246	76	31%	62	25%	53	22%	191	78%	54	22%	1	0%
Total	4,070	2,131	52%	301	7%	280	7%	2712	67%	1048	26%	310	8%

Garcia [7] investigated and presented some bad smells from architectural perspectives. These methods presented the concepts and principles for the design solutions or problems, but still leave much of the effort to developers, and depend on the skill of the architecture analysts.

Automatic detection of architecture problems has been widely studied. Mo et al.'s [14, 15] work formally defined a suite of architecture hotspot patterns—recurring architecture smells—in a project. Xiao et al.'s [18] work helps to detect architecture roots, file groups where the constituent files are architecturally connected and cause high maintenance costs. However, all of this work focuses on file-level problems. Even if we found the flawed connections among files, it still worth for developers or architects to examine the details in the flawed files. In particular, for large-scale and long-term projects, file sizes increase as software evolves. What is worse, the larger and complicated a file, and the more attributes or methods it will have. In our approach, we automatically detect and model the group of attributes or methods or fields which participate in architecture flaws and accumulate maintenance costs.

VII. Conclusion

In this paper, we have formally defined the concept of *Method-level Hotspot*, a group of evolutionarily coupled attributes or methods that participate in architecture design flaws and continuously incur maintenance difficulties. Our approach could automatically detect method-level hotspots from a project and model the evolution trend of the detected method-level hotspots in terms of maintenance costs. Our approach also uses *method-level co-change pairs* to visualize the co-change details between attributes or methods in a *method-level hotspot*.

From our analysis on six large-scale open source projects, we have demonstrated that the attributes or methods involved in *method-level hotspots* have significantly higher change-proneness compared to the attributes or methods not involved in any *method-level hotspot*. We have also presented that our approach could model most of the identified method-level hotspots into one of the three typical regression models. The modeling results could help developers understand the evolution trend of each hotspot and provide guidance for the refactoring decisions.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under the grant No. 61702377, and the Fundamental Research Funds for the Central Universities under the grant No. CCNU19TD003, and IBO Technology (Shenzhen) Co., Ltd., China.

References

[1] C. Y. Baldwin and K. B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. MIT Press, 2000.

[2] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35(6):864–878, July 2009.

[3] 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*, 33(11):725–743, 2007.

[4] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, July 1999.

[5] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proc. 14th IEEE International Conference on Software Maintenance*, pages 190–197, Nov. 1998.

[6] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[7] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In *Proc. 13th European Conference on Software Maintenance and Reengineering*, pages 255–258, Mar. 2009.

[8] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[9] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 263–272, 2005.

[10] J. F. Hair, C. M. Ringle, and M. Sarstedt. Pls-sem: indeed a silver bullet. *Journal of Marketing Theory and Practice*, 19(2):139–151, 2011.

[11] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proc. 24thInternational Conference on Software Engineering*, 2002.

[12] J. Joseph F. Hair, G. T. M. Hult, C. Ringle, and M. Sarstedt. *A Primer on Partial Least Squares Structural Equation Modeling (PLS-SEM)*. Sage, Thousand Oak, 2013.

[13] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka. A case study in locating the architectural roots of technical debt. In *Proc. 37th International Conference on Software Engineering*, May 2015.

[14] R. Mo, Y. Cai, R. Kazman, and L. Xiao. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *Proc. 12thWorking IEEE/IFIP International Conference on Software Architecture*, May 2015.

[15] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng. Architecture anti-patterns: Automatically detectable violations of design principles. *IEEE Transactions on Software Engineering*, 2019.

[16] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proc. 28th International Conference on Software Engineering*, pages 452–461, 2006.

[17] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[18] L. Xiao, Y. Cai, and R. Kazman. Design rule spaces: A new form of architecture insight. In *Proc. 36rd International Conference on Software Engineering*, 2014.

[19] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng. Identifying and quantifying architectural debt. In *Proc. 38thInternational Conference on Software Engineering*, pages 488–498, 2016.