Quantifying Architectural Debts

Lu Xiao
Drexel University
Philadelphia,United States
lx52@drexel.edu

ABSTRACT

In our prior research, we found that problematic architectural connections can propagate errors. We also found that among multiple files, the architectural connections that violate common design principles strongly correlate with the error-proneness of files. The flawed architectural connections, if not fixed properly and timely, can become debts that accumulate high interest in terms of maintenance costs over time. In this paper, we define **architectural debts** as clusters of files with problematic architectural connections among them, and their connections incur high maintenance costs over time. Our goal is to 1) precisely identify which and how many files are involved in architectural debts; 2) quantify the penalties of architectural debts in terms of maintenance costs; and 3) model the growth trend of penalties—maintenance costs—that accumulate due to architectural debts. We plan to provide a quantitative model for project managers and stakeholders as a reference in making decisions of whether, when and where to invest in refactoring.

Categories and Subject Descriptors

D.2.9 [**Software engineering**]: Management

Keywords

software quality, maintenance costs, software architecture, refactoring, architectural debt

1. PROBLEM STATEMENT AND MOTIVATION

Technical debt, proposed by Cunningham in 1992, is a metaphor used to describe the consequences of near-sighted coding activities. The definition of technical debt has been refined and expanded to describe various kinds of problems ,caused by sacrificing long-term goals for short-term benefits, in software development. In recent years, researchers gather together in technical debt workshops to discuss what is the definition of technical debt and how to manage it [8]. However, many questions remain open: how to identify technical debt? how to quantify penalties of technical debt? how to monitor the variation of technical debts over time?

Researchers in the bug prediction field found that buggy files in the past are likely to be buggy in the future [7, 4, 12]. The implication is that bugs in these files were not fixed completely. In our recent work [16], we found error-prone files are usually architecturally connected in groups instead of being isolated from each other. We observed that architectural connections among these files can propagate errors among them, making errors hard to remove and causing maintenance costs to accumulate. In our another recent work [10], we reported that architecture connections among files that violate common design principles contribute to the error-proneness of files. Based on our prior results, we consider clusters of files with problematic architecture design that incurs high maintenance costs (in terms of high change- and error-rates) over time as **architectural debts**. These debts, if not paid off in time, will influence more files through error-propagation and keep incurring high maintenance costs.

Our research goal is threefold: first, we aim to automatically and precisely identify architectural debts—which and how many files are involved in each debt; second, we will find reliable proxies to quantify the penalties—in terms of maintenance costs—of architectural debts; finally, we plan to model the evolution trend of architectural debts so that their variation can be monitored: how the impact scope of architectural debts varies and how the debt penalties accumulate in a project over time. This model can be used to evaluate the amount of maintenance costs already incurred by the debts and estimate how much more effort is required in the future if the debts are not paid off. Stakeholders can consult this model to support their decisions of whether, when and where to make a refactoring investment in a software project.

The rest of this paper is organized as follows. Section 2 discusses related work and background. Section 3 outlines our approach. Section 4 introduces the results we acquired so far. Section 5 discusses our evaluation plan. The last section concludes this paper.

2. BACKGROUND AND RELATED WORK

Our work is related to considerable prior research.

Bug Prediction. In the life cycle of a software project, considerable amount of effort goes into testing and debugging activities. To save time and labor in such activities, researchers in bug prediction field use past information from software repositories to predict the location of bugs in the fu-

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2803194

ture [4] [11] [13]. The goal of bug prediction is to effectively locate bugs in advance to prioritize testing and debugging tasks. However, the predictive power of past bug information implies that the errors are not entirely removed, otherwise, history would not be a good predictor for the future. Little work has been done to explore the architecture connections among files that contribute to the error-proneness of a project. Also, not enough work has been done to investigate how to fundamentally reduce the overall error-proneness of a project through architecture improvement.

Architecture Views. Due to the absence of up-to-date and accurate documentation to record the architecture of software systems, researchers proposed different approaches to reverse-engineering the architecture of a project from its source code. Mancoridis [9] proposed the Bunch clustering algorithm, Tzerpos [15] designed the ACDC clustering algorithm and Bavota [3] proposed the LDA view. Different views were used to help developers understand the architecture of a software system and to support the analysis of software architecture. However, there remains a gap between software architecture and quality (in terms of error-proneness). That is, the impact of architecture design on the quality of a software project has not been fully explored.

Technical Debt. Technical debt (TD) describes the consequences of near-sighted coding activities since Ward Cunningham coined the metaphor back in 1992. People refine and expand its definition to describe various kinds of problems, caused by sacrificing long-term goals for short-term benefits, in software development [8]. Researchers suggest various heuristics to approximate TD using code anomalies such as code clones, long methods, and god classes. The problem is that code anomalies do not necessarily lead to high maintenance costs. Zazworka et al., [17] compared four TD identification techniques—modularity violations, grime build up, code smells, and automatic static analysis (ASA)—in terms of the correlation between detected TD and maintenance effort. They found only a subset of TD detected by each technique over-lapped with highly error-/change-prone files. In this paper, we define the concept of architectural debts. It refer to clusters of files with high maintenance costs that are incurred by architecture design flaws among them. We also attempt to automatically and precisely identify the scope (in terms of which and how many files are involved), quantify the penalties, and model the evolution trend of such debts.

Architecture Root Detection. In our recent work [16], we proposed a novel architecture model—Design Rule Space (DRSpace)—to bridge the gap between software architecture and maintenance effort. The DRSpace model is especial in three aspects: 1) it models software architecture as multiple overlapping design spaces; 2) it distinguishes different roles of files in a project as design rules and modules based on the design rule theory proposed by Baldwin and Clark [2]; and 3) it presents structural and evolutionary dependencies among files simultaneously. Using the new architecture model—DRSpace, we designed an architecture root detection algorithm that calculates a minimal set of DRSpaces that aggregate the error-prone files. We claimed that the minimal set of DRSpaces are the *architecture roots* of error-proneness. In evaluating a single version of three open source projects: Hadoop, JBoss and Eclipse JDT, we found that the majority of the error-prone files in these projects are architecturally connected in only a few architecture roots. In addition to that, we observed various architecture issues, such as circular dependencies, unstable interfaces, unhealthy inheritance and modularity dependencies [10] in the roots. These architectural issues usually co-locate with files of high maintenance costs, suggesting a strong correlation between architecture issues and high maintenance costs in a software project.

3. APPROACH AND SOLUTION

In this section, we define the concept of ***architectural debts*** and outline our approach of identifying architectural debts, quantifying their penalties, and modeling their evolution/growth trends.

3.1 Architectural Debt Definition

We define ***architectural debts*** as clusters of files with architectural design problems that incur high maintenance costs over time. A cluster of files that satisfies the following conditions is an architectural debt.

First, the cluster of files contains problematic architectural connections among them. In prior work, we observed that architecture design flaws propagate errors among files and cause high change- and error-rates of multiple files. **Second**, the cluster of files are error-prone over time. Not all files with architecture issues are error-prone and will lead to maintenance difficulties. Only files that persistently incur high maintenance costs **over time** are true debts. **Third**, files in the cluster couple with each other in revision history. The coupling among files indicates changing one file is likely to trigger changes to other files in the cluster. Files that evolutionarily coupled with each other incur high maintenance costs as a group rather than as individuals. Each cluster of file is a unit of architecture debt that merits special attention or even refactoring.

3.2 Identifying Architecture Debts

The architecture root detection algorithm proposed in our prior work [16] detects cluster of error-prone files that are architecturally connected to each other. We observed various architecture issues in these roots that co-located with high maintenance files. However, the detected architecture roots were not necessarily architectural debts. First, not all files in an architecture root were high-maintenance. Second, the architecture roots were detected in a single version during the revision history of a project; the long-term impact of architecture roots were not analyzed.

We plan to automatically and precisely identify the scope of architectural debts, in terms of which and how many files are involved, as well as how are they involved. We plan 3 steps to identify which and how many files are involved in architectural debts. First, we analyze the revision history of a project to identify cluster of files that are coupled with each other. Then, we identify problematic architectural connections among files in the clusters detected in the first step. Files involved in architectural issues are retained for further consideration. Last, we investigate the error-proneness of these clusters of files in multiple versions. Only clusters of files that are persistently error-prone in multiple versions are identified as architecture debts. To identify how files are involved in architectural debts, we plan to investigate which lines and methods in these files are touched to fix bugs in revision history, and investigate their relations. .

3.3 Quantifying Architecture Debt Penalty

After identifying architecture debts, the next step is to find reliable proxies to quantify the penalties of debts in terms of the maintenance costs they incur. The quantification evaluates the severity of architectural debts. We plan to quantify architectural debt penalties using maintenance cost measures mined from software management repositories, including revision systems and bug tracking databases. The challenges in quantifying penalties of architectural debts are from two aspects: 1) the noise in version control and bug tracking data, as discussed by other researchers [1] [5], it difficult to accurately locate bug revisions; 2) reliable proxies to measure maintenance costs are not clear. It is impossible to directly measure maintenance costs—in terms of the amount of time or money spent—due to the unavailability of such data. We plan to approximate debt penalties using measures mined from revision history, such as 1) the number of bug issues associated with architectural debts; 2) the number of bug revisions on the files involved in architectural debts; and 3) the number of code churn of the files involved in architectural debts. The goal is to evaluate the consequences of architectural debts quantitatively.

3.4 Modeling the Trend of Architecture Debt

As developers add new features and fix bugs, the architecture of a project evolves during its life cycle. We plan to explicitly model the evolution trend of architectural debts over time: how the impact scope of architectural debts enlarges and how debt penalties accumulate from version to version. The model describes the relationship between the **age** and the **scope**, as well as the **age** and the **penalties**, of architectural debts. The age of an architectural debt is the number of versions in which it is identified as a debt. The scope of an architectural debt is the number of files involved in it. The penalties of a debt are the maintenance costs, quantified using reliable proxies, incurred by files involved in the debt. This model reveals how the number of files impacted by an architectural debt increases from version to version. This model also evaluates the costs spent on an architectural debt in the past and predict how many additional penalties it will accumulate in the future. The goal is to provide quantitative reference for stakeholders and project managers in deciding if it is necessary to refactor the project to reduce the overall error-proneness fundamentally, as well as in deciding "when" and "where" to perform the refactor.

4. RESULTS ACHIEVED SO FAR

In a case study of an industry project [6], we were able to detect the architecture debts that, verified by developers, were the root causes of high maintenance costs. We used simple economic models to predict additional maintenance costs required in terms of number of bugs, lines of code in these roots. Supported by the economic model, we proposed a refactoring plan to the developers team, which was accepted and planned for implementation. This case study, however, was based on the source code of a single version of the project.

We have also collected data from 15 open source project, each with multiple stable versions selected for analysis. We are able to locate groups of architecturally connected files that are persistently change- and error-prone in these projects during multiple versions. Besides, we have built a simple regression model showing that the number of error-prone files in these groups increases over time.

5. EVALUATION PLAN

We plan to evaluate our approach using both open source and industry projects, investigating **multiple versions** of each project.

Collecting project data For each project, we will collect structural and historical data. The structural data include the reverse-engineered architecture of multiple stable versions. The historical data include the revision logs and bug tracking data.

Research Questions To evaluate the effectiveness of our approach, we plan to answer the following three research questions for each project.

1. **RQ1: Are the architectural debts identified by our approach real problems?** This question verifies if the identified architectural debts are problems that are worthy attention. If so, we also plan to find out how early our approach can detect the problem.

2. **RQ2: Are the proxies for quantifying architectural debt penalties reliable?** This research question aims at evaluating whether the quantification proxies can reliably and reasonably approximate the architectural debt penalties, as well as how reliable they are.

3. **RQ3: How effective is the evolution model of architectural debts?** This research question evaluates whether the evolution model can correctly estimate the amount of costs that have been and will be spent on a debt. We also want to know whether the model is helpful for managers in making refactoring decisions and how effective it is.

Evaluation on open source projects The advantage of studying open source project is the easily availability of the project data, even the source code, which is usually unavailable if studying an industry project. However, it is relatively hard to verify our results using open source projects, because direct feedback from developers is usually hard to acquire. Our plan is to divide the history data into two parts: "past" and "future". Thus the "future" can be used as a retrospect verification source for the "past". For RQ1, we will check if a debt identified in the "past" is still a debt in the "future", and if the number of files involved in it increases in the "future" comparing to the "past". If the answers are both yes, this debt can be verified as a real problem. For RQ3, we will verify whether our penalty model built from "past" data can correctly "predict" the "future" costs of an architectural debt. In the meanwhile, we will reach out to the leaders of open source projects to get feedback. RQ2 is hard to evaluate without feedback from developers.

Evaluation on industry projects We will also seek corporation with industry partners. The advantage of using industry data is that we can get verification and feedback directly from the developer team. We plan to interview the developers and project managers. For RQ1, we plan to ask the developers if the identified architectural debts are real problems. If the answer is positive, we will ask if they are ware of their existence and when do they know. For RQ2, we will ask the developers whether the penalty quantification deviates from their experience in development. For RQ3, we

will ask the project manager if the quantification proxies and penalty model can help them to make refactoring decisions. Finally, if a refactor decision is made with the help of our approach, we will track the updates and evaluate the real benefit, if any, of the implemented refactoring.

6. CONCLUSION

In this paper, we defined architectural debts as clusters of files with architectural design problems that incur high maintenance costs over time. We proposed an approach to automatically and precisely identify architectural debts, quantify the penalties, and model their evolution/growth trend. Our work pushes the concept of technical debt closer to a practice from a metaphor: first, we defined a special kind of "debts" ,caused by architecture design problems, that is automatically detectable; second, our approach makes the penalties of architectural debts pragmatically measurable and quantifiable. The ultimate goal of our work is to provide scientific reference to stakeholders and project managers in making decisions of whether, when and where to make a refactoring investment in a project for long-term benefits. We plan to evaluate the usefulness of our work using both open source and industry projects.

7. ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Yuanfang Cai, for her valuable feedback on this paper.

8. REFERENCES

[1] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc. Is it a bug or an enhancement?: A text-based approach to classify change requests. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, pages 23:304–23:318, New York, NY, USA, 2008. ACM.

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

[3] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. d. Lucia. Improving software modularization via automated analysis of latent topics and dependencies. *ACM Trans. Softw. Eng. Methodol.*, 23(1):4:1–4:33, Feb. 2014.

[4] 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.

[5] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 392–401, Piscataway, NJ, USA, 2013. IEEE Press.

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

[7] S. Kim, T. Zimmermann, J. Whitehead, and A. Zeller. Predicting faults from cached history. In *Proc. 29st International Conference on Software Engineering*, pages 489–498, May 2007.

[8] P. Kruchten, R. Nord, and I. Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, Mar./Apr. 2012.

[9] S. Mancoridis, B. S. Mitchell, Y.-F. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proc. 15th IEEE International Conference on Software Maintenance*, pages 50–59, Aug. 1999.

[10] R. Mo, Y. Cai, R. Kazman, and L. Xiao. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *Proc. 15th Working IEEE/IFIP International Conference on Software Architecture*, May 2015.

[11] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22:886–894, Dec. 1996.

[12] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. In *Proc. 13thACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 86–96, July 2004.

[13] 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.

[14] R. Schwanke, L. Xiao, and Y. Cai. Measuring architecture quality by structure plus history analysis. In *Proc. 35rd International Conference on Software Engineering*, pages 891–900, May 2013.

[15] V. Tzerpos and R. C. Holt. ACDC: An algorithm for comprehension-driven clustering. In *Proc. 7th Working Conference on Reverse Engineering*, pages 258–267, Nov. 2000.

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

[17] N. Zazworka, A. Vetro, C. Izurieta, S. Wong, Y. Cai, C. Seaman, and F. Shull. Comparing four approaches for technical debt identification. *Software Quality Journal*, pages 1–24, 2013.