2018 44th Euromicro Conference on Software Engineering and Advanced Applications

A Case Study of the Effects of Architecture Debt on Software Evolution Effort

Will Snipes
ABB Corporate Research
will.snipes@us.abb.com

Sunil L Karlekar
ABB GISPL
sunil.l.karlekar@in.abb.com

Ran Mo
ABB Corporate Research
rm859@drexel.edu

***Abstract*—In large-scale software systems, the majority of defective files are architecturally connected, and the architecture connections usually exhibit design flaws, which are associated with higher change-proneness among files and higher maintenance costs. As software evolves with bug fixes, new features, or improvements, unresolved architecture design flaws can contribute to maintenance difficulties. The impact on effort due to architecture design flaws has been difficult to quantify and justify. In this paper, we conducted a case study where we identified flawed architecture relations and quantified their effects on maintenance activities. Using data from this project's source code and revision history, we identified file groups where files are architecturally connected and participated in flawed architecture designs, quantified the maintenance activities in the detected files, and assessed the penalty related to these files.**

***Index Terms*—Software Architecture; Software Maintenance; Technical Debt**

I. Introduction

Software architecture is critical in the entire life cycle of a software project. Despite many studies in architecture design and analysis over the past decades, it still remains difficult for architects to justify the architecture designs, quantify the maintenance effort caused by poor architecture designs and assess the penalty from these. Identifying technical debt from architecture problems is still informal and experience-based. Code smells [5] have been proposed to detect signs of flaws and technical debt, and this process is supported by many open source and commercial tools [1], [9], [13]. However, many code smells do not directly indicate architectural problems, making it difficult for architects to use code smells to precisely locate and quantify the effects of the architecture problems. In addition, architecture research [16] has shown that, in large-scale software systems, bug-prone files usually are architecturally connected, and the connections exhibit design flaws which propagate errors among involved files.

In this paper, we present a case study on a large-scale project, which has evolved for 9 months through a series of change requests since the last release. In this case study, we used the definition and techniques in [8], [16] to identify file groups where the files are architecturally connected and flawed by architecture issues. Considering these flawed file groups as architecture debt, we analyze whether the files have been continuously changed as the software evolves, quantify the maintenance effort spent on the files during the evolution of the software, and assess the additional maintenance effort of these files.

Our results show that: 1) almost all files captured by architecture roots are involved in architecture issues, and these files continuously incur changes as software evolves with new change requests; 2) the identified file groups consumed 54% maintenance effort spent on the project.

The remainder of this paper is organized as follows: Section II discusses related work, Section III discusses research questions, Section IV discusses the procedure we followed in this analysis, Section V provides results of the analysis for the research questions, Section VI discusses the threats to validity, and Section VII highlights conclusions.

II. Related Work

Technical debt is defined as making technical compromises that are expedient in the short-term, but that create a technical context that increases complexity and cost in the long-term [2]. In our work, we evaluate the technical debt due to architectural dependencies in an industrial software program and evaluate effects on the long-term effort to work around that debt when developing new features. This body of related work is discussed in two parts, the first focuses on the support in literature for our approach to technical debt analysis. The second is on previous studies that considered effort in their technical debt evaluation.

A. *Background on Technical Debt Analysis*

Sullivan et al. use a Design Structure Matrix (DSM) to model information hiding in software designs [11]. Following this work, Cai et al. demonstrate how to form a DSM from the pair-wise dependence relation in a design [3]. They developed a tool to automate the process of calculating the DSM.

Wong et al. present the Design Rule Hierarchy (DRH) as a software dependency structure where the decisions within the top layer form the basis for a stable system, while the decisions within subsequent layers are based on assumptions that the top layer supports [15]. Independent modules are formed from the clusters of design decisions within a layer.

Xiao et al. define a DRSpace (Design Rule Space) as a set of architecturally connected files in a DRH [16]. Through the use of a DSM, they analyze the architecture as a set of overlapping DRSpaces. They further describe architecture roots as architecturally connected files that are associated with high frequency of changes in the configuration management history.

978-1-5386-7383-6/18/$31.00 ©2018 IEEE
DOI 10.1109/SEAA.2018.00071

In this work we apply tools built by these researchers and study the effects of architecture roots on the effort required to evolve software.

B. Studies of Technical Debt Effort

Kazman et al. apply the Titan tool to evaluate technical debt in the architecture of industrial software [7]. They compare the results from Titan with those from SonarQube finding that Titan has better precision and recall of change-prone and bug-prone files. An estimation of the impact of refactoring is performed to determine the additional lines of code and effort required for additional changes and bug fixes due to the technical debt.

Guo et al. study the impact of decisions to take on technical debt on the effort required to migrate to a new version of a third-party dependency [6]. They posit that the actual cost to migrate to the new version could have been reduced by changing decisions made to support an older protocol and to decouple the protocol and application layers of the system.

In another case study, Nord et al. define a formula for estimating the impact of technical debt in architecture upon a potential new feature development [10]. The formula considers change propagation, number of dependencies, and rework estimates for existing components that must be refactored. They illustrate the use of change propagation and rework cost in a case study showing that making an investment in architecture components that support the long-term evolution of the product is more cost-effective.

Curtis et al. describe a model for estimating technical debt principal in terms of cost [4]. The model has three estimation methods that determine the cost based on parameters for the effort to fix technical debt issues identified through static analysis of source code. They evaluate the model against a set of real projects and determine the range of estimates provided by each of the three methods.

III. Research Questions

The objective of this case study is to understand if the detected architecture debt have a continuous impact on software evolution. That is, as software evolves with new change requests, determine whether files associated with poor architecture design are still change-prone and have high maintenance costs. Towards this objective, we examine the following research questions:

RQ1: Are the files involved in an architecture root still change-prone during software evolution? In other words, we examined if the detected files still incur changes during software evolution and if they are more change-prone compared to the other files.

RQ2: Whether and to what extent the file groups identified in architecture roots are associated with higher maintenance effort? The answer to this question will reveal the correlation of the identified files with the maintenance effort for these files.

IV. Study Procedure

In this paper, we conduct a case study on an industrial software system developed at ABB. *Project_EIOW* is a product implemented in C# that is used for configuring signals in a distributed control system. To proceed with our case study, we first collect the following data:

1) A file dependency report which shows dependencies between all the project's source files, output by Understand[1]

2) The project's revision history, which is extracted from its TFS[2] repository

Second, given the above inputs, we use the tools from ArchDia[3] built in [16] to identify architecture roots within the source code of *Project_EIOW*. [16] has presented that the majority of maintenance effort is spent on files involved in the architecture roots exhibiting architecture flaws. Each detected architecture root is represented as a *Design Structure Matrix (DSM)* that is output into an excel spreadsheet. From each spreadsheet, we observe the involved files and how they are architecturally connected.

Third, we calculate the change frequency of each file, which indicates how many times a file changed in the revision history. From July 2017 to March 2018, *Project_EIOW* evolved by implementing a series of change requests. By mining this period of history, we track the maintenance activity of each file and calculate the change frequency of each file.

Fourth, to answer the second research question, we measure the real maintenance effort spent on each file by tracking its work items and consulting the developers who made changes to this file. After obtaining this kind of data, we could assess the impact of architecture debt on software evolution.

V. Analysis

In this section, we describe how we identify architecture debt and how we analyze the effect of architecture debt on software evolution.

A. Architecture Root Identification

In this case study, we identify two architecture roots using the definitions and techniques defined in [16]. These two roots contain 28 distinct files. The DSM in Figure 1 presents one of the architecture roots with fake file names. Files in this root are structured into 3 layers: L1: (rc1-rc2), L2: (rc3), L3: (rc4-rc20). Using L3 as an example, this layer contains 17 files which are grouped into 6 *structurally independent* modules: M1: (rc4-rc7), M2: (rc8-rc12), M3: (rc13-14), etc.

The cell in row 3, column 1, Cell(r3,c1), is marked as "dp" without a number, which means *path1.file3_cs* structurally depends on *path1.file1_cs*, but there are no co-changes in the change history. Considering another example, Cell(r3, c2) is marked as "dp,20", which means *path1.file3_cs* structurally depends on *path1.file2_cs*. The number shows the evolutionary coupling between two files, and indicates how many times

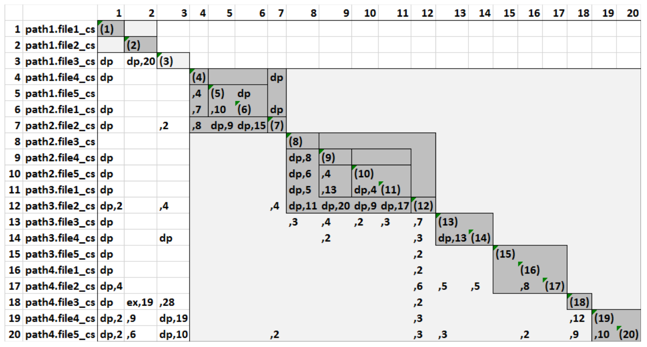
[1]https://scitools.com/

[2]https://www.visualstudio.com/tfs/

[3]https://www.archdia.net/

two files have changed together in revision history. Here, it means *path1.file3_cs* and *path1.file2_cs* have changed together 20 times as recorded in the revision history.

Fig. 1: The DSM of a detected architecture root



		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	path1.file1_cs	(1)																			
2	path1.file2_cs		(2)																		
3	path1.file3_cs	dp	dp,20	(3)																	
4	path1.file4_cs	dp			(4)			dp													
5	path1.file5_cs				,4	(5)	dp														
6	path2.file1_cs	dp			,7	,10	(6)	dp													
7	path2.file2_cs	dp		,2	,8	dp,9	dp,15	(7)													
8	path2.file3_cs								(8)												
9	path2.file4_cs	dp							dp,8	(9)											
10	path2.file5_cs	dp							dp,6	,4	(10)										
11	path3.file1_cs	dp							dp,5	,13	dp,4	(11)									
12	path3.file2_cs	dp,2		,4				,4	dp,11	dp,20	dp,9	dp,17	(12)								
13	path3.file3_cs	dp							,3	,4	,2	,3	,7	(13)							
14	path3.file4_cs	dp		dp						,2			,3	dp,13	(14)						
15	path3.file5_cs	dp											,2			(15)					
16	path4.file1_cs	dp											,2				(16)				
17	path4.file2_cs	dp,4											,6	,5	,5		,8	(17)			
18	path4.file3_cs	dp	ex,19	,28									,2						(18)		
19	path4.file4_cs	dp,2	,9	dp,19									,3						,12	(19)	
20	path4.file5_cs	dp,2	,6	dp,10				,2					,3	,3			,2		,9	,10	(20)

dp: depend, ex: extend

B. Architecture Issue Identification

Each architecture root is an architecture hotspot, which captures a group of files that are architecturally connected and these connections exhibit architecture issues. Mo et al. [8] presented how the architecture issues propagate changes among files. By using definitions and techniques defined in [8], we detect 5 types of architecture issues in our case study:

1) *Unstable Interface*: A file with a large number of dependents that changes frequently with many of them.

2) *Implicit Cross-module Dependency*: Files in *structurally independent* modules of the DRH that are changed together frequently. This is also called *modularity violation* [14].

3) *Unhealthy Inheritance Hierarchy*: A parent class depends on one of its subclasses, or a client of the inheritance hierarchy depends on both the parent class and its subclasses.

4) *Clique*: A clique is a group of files which are tightly coupled by dependency cycles to form a strongly connected component.

5) *Package cycle*: Usually, the package structure of a software system should form a hierarchical structure. A cycle among packages is typically considered to be harmful.

In this case study, we observed that 51 distinct files (25% of all files) involved in architecture issues, and all of these files have involved in 756 changes (57% of all changes for this projects). 26 of them involved in only one type of architecture issue, and the others involved in multiple architecture issues. More specifically, 26 files participated in Unstable Interface, 38 files participated in modularity violation, 12 files affected by Unhealthy Inheritance, 6 and 4 files have dependency cycles and package cycle respectively.

C. Evaluation

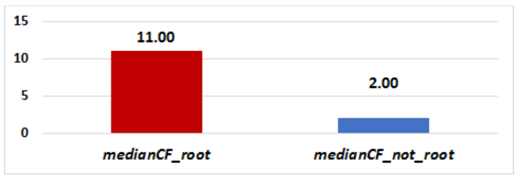
From July 2017 to March 2018, a series of change requests were implemented in *project_EIOW*. To investigate whether the detected architecture roots are correlated with higher evolution effort, we track the changes on the files and calculate the maintenance effort spent on files involved in architecture roots during this period of time.

RQ1: Are the files involved in an architecture root still change-prone during software evolution? To answer this question, we investigate the change-proneness of files during the software evolution.

We use the change frequency (CF) for each file to assess how many times a file has been changed in the revision history for bug fixes and new features. To evaluate whether files involved in an architecture root are associated with more changes in software evolution, we calculate the change frequency of the files involved in architecture roots. We also calculate the median value of change frequency for the files involved in architecture flaws as medianCF_root, and the median value for the other files as medianCF_not_root.

By mining the revision history from July 2017 to March 2018, we observe that 93% of all the files involved in architecture roots have been changed for change requests during this time period. Besides, 177 files, which are not involved in any architecture root, have also been changed. Figure 2 presents the median values of change frequency. The red cylinder presents the median CF for files involved in architecture roots. The blue cylinder presents the median values for the other files. In Figure 2, we can observe that medianCF_root is 11 and medianCF_not_root is as low as 2, which means files involved in architecture roots have more changes than files not involved in the architecture roots.

Fig. 2: Comparison: medianCF_root vs medianCF_not_root



The results suggest that, during the software evolution, files involved in architecture roots are still active for changes and they are much more change-prone than the other files.

RQ2: Whether and to what extent the file groups identified in architecture roots are associated with higher maintenance effort? To answer this question, we quantify the maintenance effort spent on each file, then analyze the effort spent on identified file groups and assess the additional effort associated with these files.

We address this question in two ways, first we perform a correlation analysis between change frequency and the maintenance effort spent on each file. We find that they are correlated using a spearman-rank correlation [12] that produced a ρ coefficient of 0.96. This result indicates that our analysis of change frequency also applies to the analysis of maintenance effort. Therefore, files that are in architecture roots are considered change-prone and consume more effort.

Second, we analyze in Table I the staff-days spent on the files in each architecture root and the other changed files. In this project, only 28 distinct files are involved in these two architecture roots, and 177 files which are not involved in any architecture root have also been changed. Considering all the studied files together, we observe that only 14% of these files are captured by root 1 and root 2, but they consumed 54% of all maintenance effort. Only 46% of all maintenance effort spent on the other changed files. The result indicates that the file groups associated in architecture roots also consumed the majority of maintenance effort spent on this project.

TABLE I: Maintenance effort spent on the project

	Staff-days	Staff-Months	Percentage of total effort
Root1	260	12	34%
Root2	150	7	20%
Rest	350	18	46%
Total	760	35	100%

Similar to the real world debt, we consider each architecture root an a debt which consists of principle and interest in terms of maintenance effort. We calculate the expected maintenance effort spent on a root to be the principle and calculate the penalty from each root as the interest, which is the difference between the actual maintenance effort and the expected maintenance effort. We use existing project median as a basis for estimating expected effort. From July 2017 to March 2018, all the 205 files have been changed 1,325 times, the *median* file in this project has 2 changes. 760 staff-days have been spent on this project, the *median* change in this project takes 3 staff-days. Our assumption is that the expected effort on each of the files involved in *architecture roots* should be similar to an *median* file, that is to be changed 2 times in the revision history, and each change would require 3 staff-days.

Based on this assumption, we can now calculate the expected effort spent on the 28 files in detected architecture roots: *3*2*28 = 168* staff-days. Table I shows the actual effort spent on architecture roots is 410 staff-days. Now, we are able to calculate the penalty to be 242 staff-days, indicating that in the 9 months' evolution with 4 resources, the files in architecture roots are related to a maintenance penalty of 242 staff-days, which is about 31% of all effort on this project.

These results demonstrate that files involved in architecture roots are associated with higher effort that was spent on these files.

VI. Discussion

In this section, we discuss the limitations of our work. The first threat is that we only studied one project, and we cannot claim that the result can be generalized to other projects in different sizes, domains or languages. The second threat is that we only studied 9-months of history because this project is a relatively new project. The third threat is our study depends on the availability of revision history with associated work items that capture the effort. Although architecture roots contain files with higher change frequency and maintenance effort, this does not extend to indicate that architecture roots cause high changes or maintenance effort.

VII. Conclusion

In this paper, we present a case study to identify and quantify the effect of architecture debt on software evolution. Through our analyses, we find that: 1) files involved in architecture roots are flawed by architecture issues and are much more change-prone than the other files; 2) during the evolution of software, files involved in architecture roots are associated with higher maintenance effort and this effort is much greater than that of the *median* file's changes. In future work, we plan to study the process of issue removal to determine which issues are the most costly to remove and generally extend this analysis to a wider range of project domains and sizes.

References

[1] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 181–190, Washington, DC, USA, 2011. IEEE Computer Society.

[2] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman. Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). *Dagstuhl Reports*, 6(4):110–138, 2016.

[3] Y. Cai and K. J. Sullivan. Modularity analysis of logical design models. In *Proc. 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 91–102, Sept. 2006.

[4] B. Curtis, J. Sappidi, and A. Szynkarski. Estimating the principal of an application's technical debt. *IEEE software*, 29(6):34–42, 2012.

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

[6] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Q. B. Da Silva, A. L. M. Santos, and C. Siebra. Tracking technical debt—An exploratory case study. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 528–531. IEEE, 2011.

[7] 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.

[8] 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.

[9] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, and L. Duchien. A domain analysis to specify design defects and generate detection algorithms. In *Proc. 11th International Conference on Fundamental Approaches to Software Engineering*, pages 276–291, Mar. 2008.

[10] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas. In Search of a Metric for Managing Architectural Technical Debt. In *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, pages 91–100. IEEE, Aug. 2012.

[11] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *Proc. 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 99–108, Sept. 2001.

[12] T.Hill and P.Lewicki. *Statistics: Methods and Applications*. StatSoft, Inc., 2007.

[13] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, May 2009.

[14] S. Wong and Y. Cai. Improving the efficiency of dependency analysis in logical models. In *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 173–184, Nov. 2009.

[15] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi. Design rule hierarchies and parallelism in software development tasks. In *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 197–208, Nov. 2009.

[16] 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.