Detecting and Preventing the Architectural Roots of Bugs

Lu Xiao
Drexel University
Philadelphia, PA, USA
lx52@drexel.edu

ABSTRACT

Numerous techniques have been proposed to locate buggy files in a code base, but the problem of fixing one bug unexpectedly affecting other files is persistent and prevailing. Our recent study revealed that buggy files are usually architecturally connected by architecture issues such as unstable interfaces and modularity violations. We aim to detect and prevent these architecture issues that are the root causes of defects. Our contributions include (1) a new architecture model,*Design Rule Space (DRSpace)*, that can express structural relations, quality, and evolutionary information simultaneously; (2) a method of automatically extracting defect-prone architecture roots by combining static architecture analysis with software revision history data mining. The preliminary application of our approach to dozens of open source and industry projects has demonstrated its significant potential to inform developers about how software defects should be discovered, examined, and handled.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design

Keywords

Software Architecture, Software Quality, Architecture Recovery

1. PROBLEM AND MOTIVATION

Research has shown that bug fixing history is one of the best predictors of future bug location [5]. The implication is that it is hard to completely remove bugs from files with defects. The problem of fixing one bug unexpectedly affecting other files is persistent and prevailing. Our recent research [3] has revealed that, in large-scale software systems, defective files seldom exist alone. On the contrary, defective files are usually architecturally connected, and their architectural structures exhibit significant issues that are, we hypothesize, the root causes of their bugginess. The implication is that just examining and fixing individual defective files, as is done today, will not remove the root causes of bugginess or avoid the inevitable ripple effects of causing more bugs when fixing one.

Our motivation is to detect and prevent these architecture issues that are the root causes of defects (which we call *architecture roots*). This is, we claim, the key to reducing the overall bugginess of a project. We are proposing new models, methods, and supporting tools that should fundamentally change the way software defects are discovered, examined, and handled. In short: 1) defective files should be treated as architecturally connected groups rather than individual files; 2) to reduce the recurrence of bugs in an architecture, the architectural roots of the bugginess must be addressed.

2. BACKGROUND AND RELATED WORK

Our work bridges the gap between software architecture and bug prediction, based on the conceptual framework of design rule theory [1] and design rule hierarchy [9].

Numerous approaches have been proposed to predict the location of bugs, by examining complexity metrics [6], or mining software history [5]. Ostrand et al. [7] built their prediction model based on a combination of static and history analysis. The goal of most bug prediction research is to predict the location of bugs, but all existing work ignores the architecture relations among these buggy files. In the field of software architecture, including software architecture recovery techniques such as Bunch [4] and LDA [2], the focus is to increase the understandability of a system, but there is little work investigating the impact of architectural decisions on software quality.

To bridge the gap between software architecture and quality, we leverage *Design Rule theory* proposed by Baldwin and Clark [1]. A *Design Rule* is an architecturally important design decision that dominates and decouples other parts of the system into independent modules. Wong et al. [9] proposed a *design rule hierarchy* algorithm used to identify *design rules* and *independent modules* in software systems. Next we present our approach based on these concepts.

3. APPROACH AND UNIQUENESS

Our approach features a new architecture model,*Design Rule Space (DRSpace)*, that can express software structure relations, quality, and evolutionary information simultaneously, paired with an automatic *architecture root* detection technique that locates defect-prone architecture issues by combining static architecture analysis with software revision history data.

FSE'14, November 16–21, 2014, Hong Kong, China
ACM 978-1-4503-3056-5/14/11
http://dx.doi.org/10.1145/2635868.2661679

Design Rule Space (DRSpace) Modeling. We view a software architecture as composed of multiple DRSpaces, each modeling a different aspect of the architecture. Each DRSpace consists of one or more design rules and independent modules that are decoupled by, and depend on, the design rules. For example, in a system applying a strategy pattern, the strategy interface is an important design rule that decouples concrete strategies with clients. The DRSpace is visualized and manipulated using a Design Structure Matrix (DSM) [1] representation.

Our model is unique in the following aspects. First, a DRSpace can be formed using different types of dependencies, including static dependencies, such as "extend", "implement" and "call", and also evolutionary coupling, showing the number of times two files change together in a project's revision history. Second, with our supporting tool, Titan, the user can flexibly examine different perspectives of the architecture, by first selecting dependency relations of interest, and then applying design rule hierarchy algorithm.

For example, if a system applies multiple design patterns, each pattern forms its own design space that can be examined separately using a DRSpace. Such a *design pattern view* can be generated by selecting the key interfaces of a design pattern, extracting the modules from the software system that depend on this interface, and clustering them into a design pattern DRSpace. If the user only chooses evolutionary coupling, then an *evolutionary view* can be generated to reflect the groups of files that have frequently changed together. Similarly, if the user only select "extend" and "implement", a *polymorphism view* can be generated to show the modular structure formed by base and sub-classes.

A DRSpace formed by multiple structure dependencies reveals a *hybrid view*, which can be used to identify architecture issues. Figure 1 demonstrates a polymorphism view of Hadoop, with the *FileSystem* class chosen as the base class. After the *dependency* relation is selected, an architecture issue is revealed: the parent class, `FileSystem`, depends on its child class, `DistributedFileSystem`. After adding the evolutionary dependency relation, shown as the numbers in each cell, we can see that this issue is associated with 26 co-changes (i.e., the two classes changed together 26 times). Modularity violations [8] among files, highlighted in red, implies shared secrets even though the classes don't directly depend on each other. The analysis of this DRSpace suggests that, to reduce maintenance costs, the developers need to better design the polymorphism structure led by `FileSystem`.

	1	2	3	4	5	6	7	8
1 org.apache.hadoop.fs.FileSystem	(1)			Depend,				
2 org.apache.hadoop.fs.RawLocalFileSystem	Extend,	(2)						
3 org.apache.hadoop.fs.InMemoryFileSystem$RawInMemoryFileSystem	Extend,7	,8	(3)					
4 org.apache.hadoop.dfs.DistributedFileSystem	Extend,26	,7		(4)				
5 org.apache.hadoop.fs.kfs.KosmosFileSystem	Extend,				(5)			
6 org.apache.hadoop.dfs.HttpFileSystem	Extend,					(6)		
7 org.apache.hadoop.fs.FilterFileSystem	Extend,						(7)	
8 org.apache.hadoop.fs.s3.S3FileSystem	Extend,8	,6						(8)

Figure 1: DRSpace from Hadoop

Architecture Root Detection. Considering all the files with multiple bug fixes as a *BugSpace*, we created a novel architecture root detection algorithm that extracts the set of DRSpaces that covers a specified portion of defective files from the whole DRSpace of the system. Our approach works as follows: The first component takes dependency information extracted from source code and transforms it into a DSM with just structural dependencies. The second component takes the revision history of a system as input and generates a DSM with just evolutionary coupling. The third component produces a bug space based on the bug fixing history. The fourth component takes the output of the first three components as input and computes a group of DRSpaces that connect the most error-prone files as architecturally related groups. This approach is unique in that it is the first to automatically bridge the gap between software architecture and quality (as measured by bugginess).

4. RESULTS AND CONTRIBUTIONS

We have developed a toolset called Titan to support the approach introduced above. Titan supports the automatic generation of architecture roots and the interactive visualization of DRSpaces to aid in various architecture analyses.

In our prior study [3] of three open source projects, we made the following observations: 1) a project's buggy files are architecturally connected and only a few DRSpaces can cover the majority of buggy files: in all three systems we studied, 78% to 89% of most error-prone files are captured by just 5 DRSpaces. We call these DRSpaces *architecture roots*. 2) The DRSpaces led by error-prone design rules tend to be also error-prone. 3) Error-prone DRSpaces, that is, architecture roots, usually contain architecture issues, such as inappropriate inheritance, unstable interfaces, modularity violations, etc. that contribute to the error-proneness.

In our most recent study of ten open source projects and one industry project, similar observations were made: the majority of the buggy files were architecturally connected by only a few DRSpaces. Besides, we also found that most of the projects have the same set of architecture roots over different snapshots during the life cycle of the project. More importantly, as the size of bug space grows over time, the number of architecture roots required to cover it remain constant, showing that the impact of architecture roots are consistent and significant. These results strongly suggest that when developers are fixing bugs, they should 1) treat buggy files as architecturally connected groups instead of single files; 2) examine the architecture issues within each architecture root and fix these, rather than attempting to fix a bug in isolation.

5. SUMMARY AND FUTURE WORK

In summary, our research contributes a new architecture model and an architecture root detection technique, bridging the gap between architecture and quality by clustering hundreds of buggy files into just a few DRSpaces. Our Titan tool provides interactive visualization of DRSpaces, so that developers can examine the architecture issues behind the buggy files, and get insights in how to fix them. Buggy files are usually connected to each other by architectural design flaws that contribute to the error-proneness. For this reason the developer should treat buggy files as architecturally connected groups and fix the architecture issues behind the buggy files to avoid recurring bugs. Our future work is to further automate the architecture issue detection within DRSpaces and to automatically provide refactoring suggestions. We also plan to leverage *Design Rule Spaces* to automate the detection of design patterns and anti-patterns.

6. REFERENCES

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

[2] M. Gethers and D. Poshyvanyk. Using relational topic models to capture coupling amoung classes in object-oriented software systems. In *Proc. 26th IEEE International Conference on Software Maintenance*, pages 1–10, Sept. 2010.

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

[4] 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.

[5] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proc. 27th International Conference on Software Engineering*, pages 284–292, May 2005.

[6] 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.

[7] 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.

[8] 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.

[9] 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.