Design Rule Spaces: A New Model for Representing and Analyzing Software Architecture

Yuanfang Cai, Lu Xiao, Rick Kazman, Ran Mo, and Qiong Feng

Abstract—In this paper, we propose an architecture model called *Design Rule Space (DRSpace)*. We model the architecture of a software system as multiple overlapping *DRSpaces*, reflecting the fact that any complex software system must contain multiple aspects, features, patterns, etc. We show that this model provides new ways to analyze software quality. In particular, we introduce an Architecture Root detection algorithm that captures *DRSpaces* containing large numbers of a project's bug-prone files, which are called *Architecture Roots (ArchRoots)*. After investigating *ArchRoots* calculated from 15 open source projects, the following observations become clear: from 35 to 91 percent of a project's most bug-prone files can be captured by just 5 *ArchRoots*, meaning that bug-prone files are likely to be architecturally connected. Furthermore, these *ArchRoots* tend to live in the system for significant periods of time, serving as the major source of bug-proneness and high maintainability costs. Moreover, each *ArchRoot* reveals multiple architectural flaws that propagate bugs among files and this will incur high maintenance costs over time. The implication of our study is that the quality, in terms of bug-proneness, of a large, complex software project cannot be fundamentally improved without first fixing its architectural flaws.

Index Terms—Software architecture, reverse-engineering, defect prediction, technical debt, code smells, bug localization

1 Introduction

To better understand, analyze, and maintain software architecture, especially the architecture as implemented in source code, we present an architectural model called *Design Rule Space* (DRSpace)—based on Baldwin and Clark's *design rule* theory [1]. According to their theory, *design rules* are the key architectural decisions that decouple the rest of the system into independent modules that can be implemented, revised, or even replaced without influencing other parts of the system. Independent modules can generate significant value in the form of options. We use DRSpaces to model a group of files composed of design rules and modules, modeling one aspect of a complex software architecture. Intuitively, a nontrivial software system must contain multiple design spaces: each feature implemented, each pattern applied, and each concern addressed will have their own design space. Accordingly, we propose that software architecture can and should be modeled as multiple, overlapping DRSpaces.

Software architecture describes software elements and their relationships in a system, which may come in different granularities and contain various forms of relationships [2]. For example, the elements can be source files and methods, and the relationships could be the dynamic behavior or static connections. In this paper, we specifically focus on a view of *software architecture* that describes the relations among source files. Architectural decisions, once implemented in source code, can become hard to understand, hard to modify, and can suffer from gradual decay due to continuous evolution. Recent research [3] has shown that architectural choices are the number one source of technical debt in source code. Analyzing the architecture, as realized in source code, is the focus of our research.

Researchers have proposed various ways to recover high-level software architecture views from source code based on different criteria [4], [5], [6], [7], [8], [9], [10]. For example, Praditwong et al. [4] proposed a search-based approach to cluster software objects into modules based on high cohesion and low coupling. Corazza et al. [5] leverage lexical information, e.g., in comments, identifier and method names, to cluster software elements into clusters. Naseem [7] cluster software entities based on various similarity measures. These approaches are based on different rationales and can produce dramatically different high level models from the same source code. Instead of choosing which one is correct, we observe that they all make sense from their own perspectives since a complex software system can always be viewed and analyzed from different perspectives. However, it must be noted that these techniques produce restrictive clusterings: that is, a software unit can only belong to a single cluster. In reality, however, it is natural for one file to take multiple roles and participate in

- *Y. Cai, R. Mo, and Q. Feng are with the Department of Computer Science, Drexel University, Philadelphia, PA 19104. E-mail: yfcai@cs.drexel.edu, {rm859, qf28}@drexel.edu.*
- *L. Xiao is with the School of Systems and Enterprises, Stevens Institute of Technology, Hoboken, NJ 07030. E-mail: lxiao6@stevens.edu.*
- *R. Kazman is with the Department of Information Technology Management, University of Hawaii, Honolulu, HI 96822. E-mail: kazman@hawaii.edu.*

Manuscript received 8 Feb. 2017; revised 2 Dec. 2017; accepted 21 Dec. 2017. Date of publication 23 Jan. 2018; date of current version 24 July 2019. (Corresponding author: Lu Xiao.) Recommended for acceptance by A. Egyed. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TSE.2018.2797899

multiple relationships. Our DRSpace model is proposed based on these observations.

In this paper, we report on our investigation of using DRSpaces to reveal the architectural impact of bug-prone files. In the field of defect prediction, numerous studies have shown that historical bug information can be a reliable source for predicting the locations of future bugs [11], [12], [13], [14], [15], [16], [17]. That is, bug-prone files in the past tend to remain bug-prone in the future. The implication of this correlation, however, is that bugs are seldom successfully eradicated from files, otherwise, history would not be a reliable predictor of future bugs. In a case study of an industrial software project [18], we showed that architectural flaws were the *fundamental causes* for bug-proneness. These architectural flaws propagated bugs among source files and hence made bugs difficult to eradicate. For example, when a developer changes an interface to fix a bug, they are also likely to revise the concrete classes that implement the interface. Their work is shaped by architectural connections.

The relationship between software architecture and bug-proneness has not, however, been adequately investigated to date. There has been little research illuminating how bug-prone files are architecturally connected, and the nature of the flaws introduced by these architectural connections. And there is little understanding of how flawed architectural relations contribute to the overall bug-proneness of a project over time.

In this paper, we show, using DRSpaces, that we can automatically identify architectural flaws that contribute to bug-proneness in a software project. This is achieved by automatically identifying a minimal set of *DRSpaces* that connect the top bug-prone files in the system. We call these *DRSpaces Architecture Roots* (ArchRoots). We hypothesize that the *ArchRoots* have deep-seated and enduring impacts on the bug-proneness of a project. They propagate bugs among files, making bugs hard to eradicate, and consequently causing maintenance costs to increase over time. As we will show, developers can not systematically reduce the bug rate of a project without fixing these *ArchRoots* first.

To evaluate the usefulness of DRSpaces and ArchRoots, we have studied the following research questions:

(1) *If an architecturally influential file (design rule) is bug-prone, are the files within its design space also likely to be bug-prone, and more bug-prone than other random design spaces?* The answer to this question will help us understand the architectural impact of design rules.

(2) *To what extent are bug-prone files architecturally connected?* If a large portion of error-prone files are captured in a few ArchRoots, it implies significant impact of architecture on software quality.

(3) *Are these* ArchRoots *long-lasting and persistent during projects evolution?* The answer to this question will help us understand if the architecture impact is persistent.

(4) *Do these* ArchRoots *contain architectural flaws that may contribute to their bug-proneness?* Finally, we would like to understand what kinds of architectural problems are behind these error-prone files.

To answer these questions, we examined multiple stable releases (between 9 and 15 releases) of 15 large open source projects, covering 4 to 8 years of revision history. Despite their varying characteristics—application domain, age, and scale—our approach has consistently shown its usefulness in advancing the understanding of the relationship between software architecture—in particular the flawed architectural relationships—and bug-proneness.

The takeaway messages of our study are fourfold. First, bug-prone and broad-impact design rules should merit serious attention in bug fixing activities given their influences on their dependents. Second, when trying to fix bugs, instead of focusing on bug-prone files individually, software practitioners should focus on the structure of the most important *ArchRoots* that aggregate the majority of these bug-prone files. Third, the long-lived *ArchRoots* will keep experiencing high bug-rates and thus incurring high bug-fixing costs over time if their design flaws are not fixed. Finally, high bug-rates in a project can not be fundamentally reduced without first fixing the architectural flaws in the *ArchRoots* that cause bugs to propagate.

The rest of this paper is organized as follows. Section 2 introduces the background concepts and techniques for our approach. Section 3 introduces our *Design Rule Space (DRSpace)* model. Section 4 introduces the *Architecture Root* detection algorithm. Section 5 presents the evaluation of the usefulness of this approach on 15 open source projects. Section 6 briefly discusses industrial applications of our approach. Section 7 discusses related work. Section 8 discusses limitations and threats to validity of our approach. Section 9 concludes this paper.

2 Background

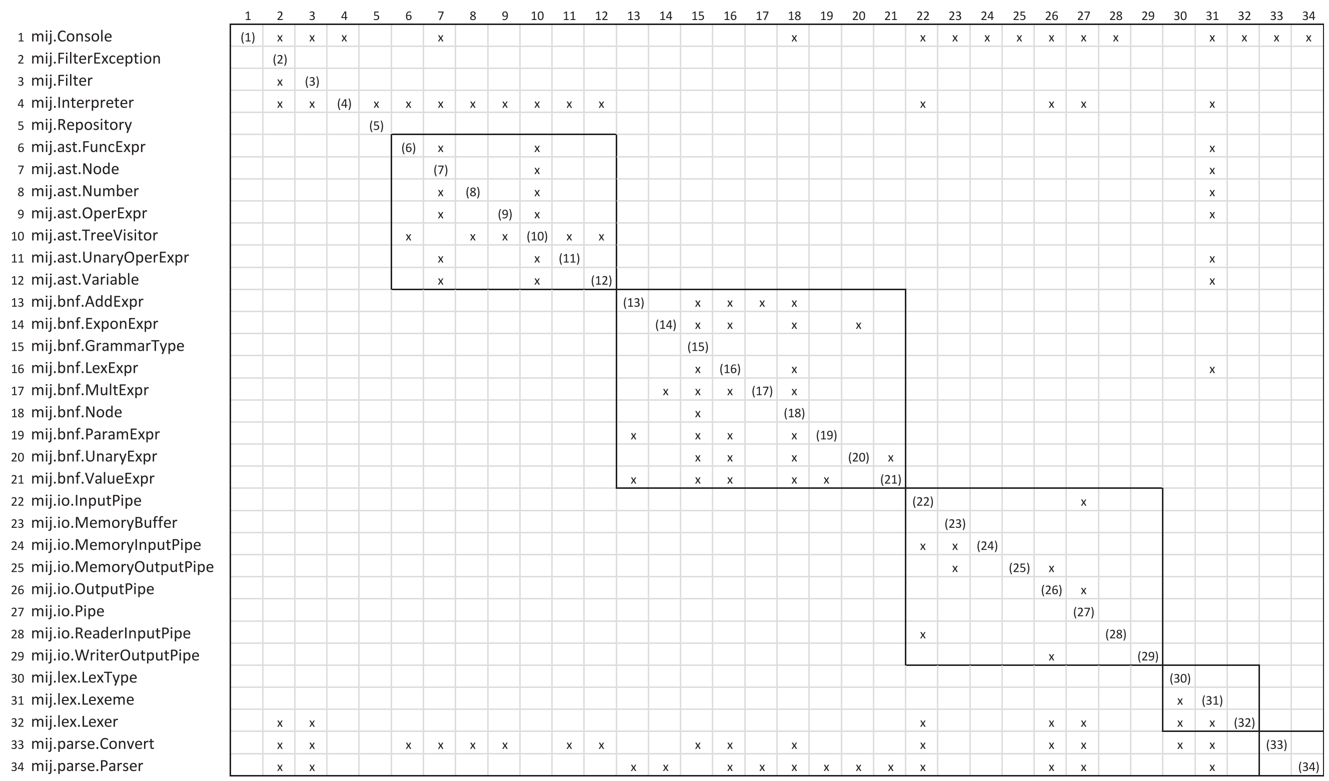
In this section, we introduce the background concepts and techniques our approach is based upon.

2.1 Design Rule Theory

Our research is based on *Design Rule Theory*, proposed by Baldwin and Clark [1]. They define Design Rules as important design decisions that decouple other parts of the system into mutually independent modules. Serving as interfaces for decoupling, design rules enable interdependence within and independence across modules. The design, implementation, and improvement of an independent module should not influence other modules, as long as the established design rules are obeyed and maintained. In modern programming languages following the object-oriented philosophy, such as Java, interface or abstract classes typically play the role of design rules. For example, in the abstract factory pattern, the abstract factory interface is the design rule, which decouples concrete factories and the clients of the factories into independent modules. As long as the abstract factory interface remains stable, the implementation of a concrete factory can be done independently from the implementation of other concrete factories, and from that of the clients.

2.2 Design Structure Matrix (DSM)

Following the work of Baldwin and Clark, we model (a part of a) software architecture using a Design Structure Matrix (DSM). In a DSM, the columns and rows represent design parameters of a system (in the same order). In this paper, we use the columns and rows to model source files and



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
1 mij.Console	(1)	x	x	x			x											x				x	x	x	x	x	x	x			x	x	x	x
2 mij.FilterException		(2)																																
3 mij.Filter		x	(3)																															
4 mij.Interpreter		x	x	(4)	x	x	x	x	x	x	x	x										x				x	x				x			
5 mij.Repository					(5)																													
6 mij.ast.FuncExpr						(6)	x			x																					x			
7 mij.ast.Node							(7)			x																					x			
8 mij.ast.Number							x	(8)		x																					x			
9 mij.ast.OperExpr							x		(9)	x																					x			
10 mij.ast.TreeVisitor						x		x	x	(10)	x	x																						
11 mij.ast.UnaryOperExpr							x			x	(11)																				x			
12 mij.ast.Variable							x			x		(12)																			x			
13 mij.bnf.AddExpr													(13)		x	x	x	x																
14 mij.bnf.ExponExpr														(14)	x	x		x		x														
15 mij.bnf.GrammarType															(15)																			
16 mij.bnf.LexExpr															x	(16)		x													x			
17 mij.bnf.MultExpr														x	x	x	(17)	x																
18 mij.bnf.Node															x			(18)																
19 mij.bnf.ParamExpr													x		x	x		x	(19)															
20 mij.bnf.UnaryExpr															x	x		x		(20)	x													
21 mij.bnf.ValueExpr													x		x	x		x	x		(21)													
22 mij.io.InputPipe																						(22)					x							
23 mij.io.MemoryBuffer																							(23)											
24 mij.io.MemoryInputPipe																						x	x	(24)										
25 mij.io.MemoryOutputPipe																							x		(25)	x								
26 mij.io.OutputPipe																										(26)	x							
27 mij.io.Pipe																											(27)							
28 mij.io.ReaderInputPipe																						x						(28)						
29 mij.io.WriterOutputPipe																										x			(29)					
30 mij.lex.LexType																														(30)				
31 mij.lex.Lexeme																														x	(31)			
32 mij.lex.Lexer		x	x																			x				x	x			x	x	(32)		
33 mij.parse.Convert		x	x			x	x	x	x		x	x			x	x		x				x				x	x			x	x		(33)	
34 mij.parse.Parser		x	x										x	x		x	x	x	x	x	x	x				x	x				x			(34)

Fig. 1. MIJ DSM in package cluster.

hence the cells of the DSM represent their relations. The off-diagonal cells represent dependencies from the file on the row to the file on the column. The diagonal cells indicate self-dependency and we annotate these cells by the row/column index of the file.

Fig. 1 depicts the DSM of a simple Java program, reversed-engineered from its source code. In this DSM, the left-most column shows the indexed list of source files. The top-most row of the DSM indicates the index of same set of files with the same order. The "x" mark on cell $c(r6, c7)$ indicates that the file in row 6 (*mij.ast.FuncExpr*) depends on the file in column 7 (*mij.ast.Node*). We use blocks along the diagonals to show clusters of files. For example, in Fig. 1, files 6 to 12 are grouped together according to their directory (package) structure.

2.3 Design Rule Hierarchy (DRH)

According to Design Rule Theory, the key concepts of a software architecture are *design rules* and *independent modules*. To manifest the differing architectural importance of different source files, Wong et al. [19], [20] proposed the *Design Rule Hierarchy (DRH)* algorithm to cluster source files so that their architectural roles can be made explicit. A DSM, clustered using the DRH algorithm, has three key features. First, the design rules and modules are arranged in a hierarchical structure, with design rules on the upper layers and the modules decoupled by the design rules on lower layers. Second, the modules in lower layers depend on the modules in higher layers, but not vice versa. Third, modules in the same level are mutually independent from each other.

Fig. 2 shows a DSM with three layers, formed by the "extend" and "implement" relations among this project's files. The first layer (rows 1 to 4) contains four files that are the super classes or interfaces—the design rules. The second layer (rows 5 to 6) contains two files that "extend" *mij.io.Pipe* in the first layer. The third layer (rows 7 to 27) contains 21 files that extend/implement files in the first two layers. Layer 3 is, furthermore, internally decoupled into six mutually independent modules, where each module follows design rules in the top two layers.

Cai et al. [21] improved the original DRH algorithm to better capture the modular structure of a software architecture. The new algorithm made two improvements to the original DRH algorithm. First, it recognizes *control programs*, another important architecture role of source files in a project. A control program depends on many other classes, but other classes usually don't depend on it. A main program is an example of a control program. In Fig. 3, "mij.console" is the main program, residing at the bottom of the DSM, and it depends on 15 other files. The second improvement is that ArchDRH recursively computes the design rule hierarchy within each module of each layer. For example, in Fig. 3, the module from rows 11 to 17 contains two inner layers.

2.4 Evolutionary Coupling

In many cases, architectural relations among files cannot be detected from their structural relations. Gall et al. [22] showed that logical relations among files can often be reflected by how they were changed together as recorded in the revision history. Wong et al. [23] used the term *modularity violation* to refer to the phenomenon where a set of files *should*, according to their modular structure, evolve independently but they were actually changed together frequently as recorded in the project's revision history. Their tool, Clio, could detect *modularity violations* in a software project by analyzing its source code and revision history concurrently. In their experiments on three open source projects, Clio identified large numbers of *modularity*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1 mij.bnf.Node	(1)																										
2 mij.io.Pipe		(2)																									
3 mij.ast.Node			(3)																								
4 mij.Filter				(4)																							
5 mij.io.InputPipe		Ext			(5)																						
6 mij.io.OutputPipe		Ext				(6)																					
7 mij.io.WriterOutputPipe						Impl	(7)																				
8 mij.io.MemoryOutputPipe						Impl		(8)																			
9 mij.io.ReaderInputPipe					Impl				(9)																		
10 mij.io.MemoryInputPipe					Impl					(10)																	
11 mij.ast.TreeVisitor											(11)																
12 mij.Interpreter				Impl							Impl	(12)															
13 mij.parse.Parser				Impl									(13)														
14 mij.lex.Lexer				Impl										(14)													
15 mij.parse.Convert				Impl											(15)												
16 mij.ast.Variable			Ext													(16)											
17 mij.ast.Number			Ext														(17)										
18 mij.ast.OperExpr			Ext															(18)									
19 mij.ast.FuncExpr			Ext																(19)								
20 mij.ast.UnaryOperExpr			Ext																	(20)							
21 mij.bnf.ValueExpr	Ext																				(21)						
22 mij.bnf.AddExpr	Ext																					(22)					
23 mij.bnf.ParamExpr	Ext																						(23)				
24 mij.bnf.UnaryExpr	Ext																							(24)			
25 mij.bnf.LexExpr	Ext																								(25)		
26 mij.bnf.ExponExpr	Ext																									(26)	
27 mij.bnf.MultExpr	Ext																										(27)

Note: “Ext” stands for “Extend” and “Impl” stands for “Implement”

Fig. 2. MIJ inherit DRSpace.

violations, which were verified to cause significant maintenance consequences, including errors, modularity decay, and expensive refactoring.

We conducted a case study on an industrial project [18]. In this study, we identified and verified many cases of *modularity violations* using Clio, and found that *modularity violations* usually suggest “shared secrets” (undocumented assumptions) among files that require better encapsulation. For example, we identified a set of files that share an assumption of how a unit of time is represented, without explicitly encapsulating this assumption. Whenever the time unit used in one of the files changed, the others had to be changed as well.

This prior research showed that co-change history contains important information that can reveal architectural relations among files. As a result, we extend Baldwin and Clark’s DSM model to include co-change information

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
1 mij.io.Pipe	(1)																																
2 mij.FilterException		(2)																															
3 mij.io.MemoryBuffer			(3)																														
4 mij.io.OutputPipe				(4)																													
5 mij.io.InputPipe					(5)																												
6 mij.lex.LexType						(6)																											
7 mij.lex.Lexeme						Dp	(7)																										
8 mij.bnf.Node								(8)																									
9 mij.bnf.LexExpr							Dp		(9)	Dp																							
10 mij.bnf.GrammarType										(10)																							
11 mij.ast.Node							Dp				(11)																						
12 mij.ast.TreeVisitor												(12)																					
13 mij.ast.UnaryOperExpr							Dp				Dp	Dp	(13)																				
14 mij.ast.Variable							Dp				Dp	Dp		(14)																			
15 mij.ast.FuncExpr							Dp				Dp	Dp			(15)																		
16 mij.ast.Number							Dp				Dp	Dp				(16)																	
17 mij.ast.OperExpr							Dp				Dp	Dp					(17)																
18 mij.Filter		Dp																(18)															
19 mij.lex.Lexer	Dp	Dp		Dp	Dp	Dp	Dp											Dp	(19)														
20 mij.parse.Convert	Dp	Dp		Dp	Dp	Dp	Dp	Dp	Dp	Dp	Dp		Dp	Dp	Dp	Dp	Dp	Dp		(20)													
21 mij.io.MemoryOutputPipe			Dp																		(21)												
22 mij.io.MemoryInputPipe			Dp																			(22)											
23 mij.bnf.UnaryExpr								Dp		Dp													(23)										
24 mij.bnf.AddExpr								Dp		Dp														(24)									
25 mij.bnf.ValueExpr								Dp		Dp															(25)								
26 mij.bnf.MultExpr								Dp		Dp																(26)							
27 mij.bnf.ExponExpr								Dp		Dp																	(27)						
28 mij.bnf.ParamExpr								Dp		Dp																		(28)					
29 mij.parse.Parser	Dp	Dp		Dp	Dp		Dp	Dp	Dp									Dp					Dp	Dp	Dp	Dp	Dp	Dp	(29)				
30 mij.io.ReaderInputPipe																														(30)			
31 mij.Repository																															(31)		
32 mij.Interpreter	Dp	Dp		Dp	Dp		Dp				Dp	Dp	Dp	Dp	Dp	Dp	Dp														Dp	(32)	
33 mij.Console	Dp		Dp	Dp	Dp		Dp	Dp			Dp							Dp	Dp	Dp	Dp	Dp							Dp	Dp		Dp	(33)

Note: “Dp” stands for all other types of static references other than “Extend” or “Implement”

Fig. 3. MIJ depend DRSpace.

between files. If a cell, $c(rm, cn)$, of a DSM contains a number, this number indicates the number of times file m and file n changed together in the given time period, as we will see in later sections of the paper.

Based on this prior work, we model *DRSpaces* using DSMs. Each DRSpace reveals architecturally important design rules and modules, and models structural and evolutionary dependencies simultaneously, as defined next.

3 Design Rule Space Model

In this section, we will introduce the definition of our architecture model—*Design Rule Space (DRSpace)*.

3.1 DRSpace Definition

A *DRSpace* is composed of a subset of a system's files along with the architectural connections among these files. For example, a software project may contain multiple design patterns. Each design pattern can be represented as a unique *DRSpace*, composed of files participating in the pattern and their relations conforming to rules of the pattern. Similarly, a feature or a component can also be modeled using DRSpaces. A *DRSpace* has the following key characteristics:

(1) A *DRSpace* always contains one or more *leading files*, that is, the files that all other files in the space directly or indirectly depend on. We use the term *leading files* instead of *design rules* because the latter usually refer to architecturally important decisions for the whole system, while the former may or may not contain architecturally important decisions, but must have significant impact on other files within the same design space.

(2) A *DRSpace* captures various relations among files. The relations can be either structural dependencies or evolutionary couplings among files.

In this paper, we examine the following three major structural dependency types: "extend", "implement", and "depend". "extend" and "implement" are basic polymorphic concepts in object-oriented programming languages. "extend" refers to a child class inheriting from a parent class, and "implement" refers to a concrete class implementing the methods of an interface. All other references, such as a function calls, between files are uniformly represented as "depend".

A *DRSpace* may also contain evolutionary couplings among files as a special form of architecture connection. The evolutionary couplings are extracted from the revision history of a project. If two or more files are changed in a single commit, we consider these files to be "evolutionarily coupled". The number of times a pair of files have changed together over the project's revision history indicates the weight of their evolutionary coupling. For example, if two files changed together in 10 commits, the weight of the evolutionary coupling is 10. Here we use the definition of co-change presented in prior research [22], and will discuss its limitation in Section 8.

(3) Each *DRSpace* is clustered using the DRH algorithm to automatically manifest the *leading files* and *modules* in a DRSpace based on one or more selected types of the relations. We call these selected relation types as *primary relations*, and call all other relations *secondary relations*.

The *leading files* and *modules* are arranged in a hierarchy which we visualize using a DSM. For example, in Fig. 2, the DRSpace is clustered based on "implement" and "extend"—the selected primary relations. All the secondary relations are omitted in this example so that the polymorphic structure within the system can be analyzed (which we will discuss in detail later). The *leading files*, which are the key interfaces and parent classes, are on the top level (row 1 to row 4), while the dependent modules are on the lower levels (row 5 to row 27) of the DSM.

3.2 DRSpace Illustration

Now we use a small Java program, called MIJ, as a running example to illustrate the concept of *DRSpace*. The MIJ program supports simple math calculations, such as addition, subtraction, multiplication, and division. The design of MIJ applied multiple design patterns, including *Interpreter*, *Visitor*, and *Pipe and Filter*. The *Interpreter* pattern defines interpreter and lexer components to parse operations. The *Visitor* pattern traverses and enacts different operations on an abstract syntax tree (AST). The *Pipe and Filter* pattern facilitates communication between different components in the system.

Fig. 1 shows a DSM of the MIJ program, reverse-engineered from the source code. This DSM models general dependencies among files using "x". The DSM is clustered to show the file directory structure, as supported by many existing reverse engineering tools, such as Lattix.[1] Each inner rectangle in the matrix represents a package in MIJ—for example, files 6 to 12 are grouped in a rectangle because they are all from package *mij.ast*. This DSM gives no insight into the modular structure of each implemented design pattern, which files participated in which patterns, or the potential architectural problems in the system.

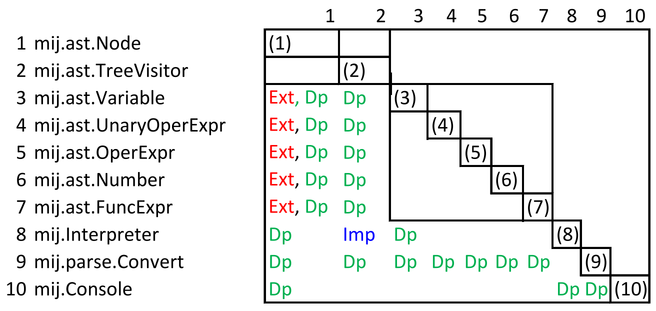
By contrast, we now illustrate the features and benefits of using *DRSpaces* to model the MIJ system.

1. View software architecture as multiple overlapping DRSpaces. The complexity of software architecture cannot be properly expressed using just a single view. Even for a small project such as MIJ, there exist multiple patterns and different features that can and should be viewed separately. Depending on different purposes, different *DRSpaces* can be extracted from a project. Here are a few examples.

Each set of dependency types can form a separate *DRSpace*. For example, Fig. 2 shows a *DRSpace* formed by dependency types "Ext" (Extend) and "Impl" (Implement). This *DRSpace* depicts the polymorphic structure of MIJ. It shows the structure of the inheritance tree in MIJ, which is not visible in Fig. 1. Groups of files extending or implementing different interfaces or base classes are clustered into different modules. For example, file *mij.ast.Variable* (row 16) to file *mij.ast.UnaryOperExpr* (row 20) form the *ast* (Abstract Syntax Tree) module, because they all extend the base class *mij.ast.Node*.

As another example, Fig. 3 shows the modular structure formed by the "Dp" (dependency) relation between files. In a "Dp" *DRSpace*, we can distinguish control modules and functional modules in this project. In Fig. 3, file *mij.Console* is

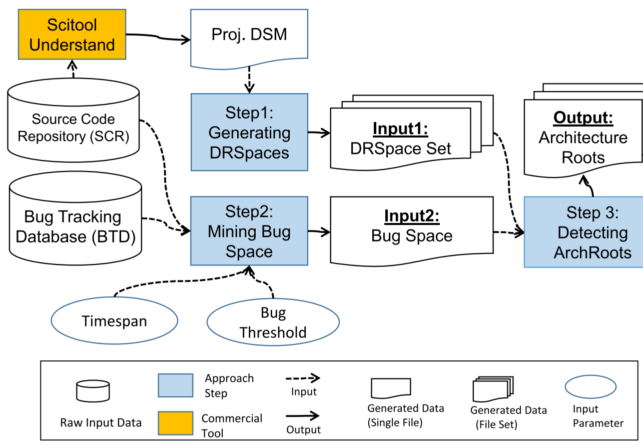
1. http://lattix.com/



	1	2	3	4	5	6	7	8	9	10
1 mij.ast.Node	(1)									
2 mij.ast.TreeVisitor		(2)								
3 mij.ast.Variable	Ext, Dp	Dp	(3)							
4 mij.ast.UnaryOperExpr	Ext, Dp	Dp		(4)						
5 mij.ast.OperExpr	Ext, Dp	Dp			(5)					
6 mij.ast.Number	Ext, Dp	Dp				(6)				
7 mij.ast.FuncExpr	Ext, Dp	Dp					(7)			
8 mij.Interpreter	Dp	Imp	Dp					(8)		
9 mij.parse.Convert	Dp	Dp	Dp	Dp	Dp	Dp	Dp		(9)	
10 mij.Console	Dp							Dp	Dp	(10)

Note: "Ext" stands for "Extend" and "Impl" stands for "Implement". "Dp" stands for all other types of static references.

Fig. 4. MIJ visitor pattern DRSpace.



Scitool Understand
Proj. DSM
Source Code Repository (SCR)
Step1: Generating DRSpaces
Input1: DRSpace Set
Output: Architecture Roots
Bug Tracking Database (BTD)
Step2: Mining Bug Space
Input2: Bug Space
Step 3: Detecting ArchRoots
Timespan
Bug Threshold
Raw Input Data
Approach Step
Commercial Tool
Input
Output
Generated Data (Single File)
Generated Data (File Set)
Input Parameter

Fig. 5. ArchRoot detection approach framework.

the control module, while file *mij.bnf.UnaryExpr* (row 23) to file *mij.Parser* (row 29) form a *bnf* expression parser module.

Each pattern can also form a *DRSpace*. In MIJ, three design patterns are used, but they are hard to distinguish from the DSM in Fig. 1. Using *DRSpaces*, each pattern can be represented separately. For example, the *DRSpace* in Fig. 4 shows the modular structure of the *Visitor* pattern. File *mij.ast.Node* and file *mij.ast.TreeVisitor* are the *design rules* of this pattern, and they decouple the other files in this pattern into modules.

Furthermore, any file (or files) can be used as a *leading file* to split out a *DRSpace* that consists of the *leading files* and files that directly or indirectly depend on the *leading files*. For instance, the visitor pattern shown in Fig. 4 is calculated using the key interfaces of the pattern, *mij.ast.Node* and *mij.ast.TreeVisitor*, as the leading files. All other files in the system that directly or indirectly depend on the leading files are extracted to form the *DRSpace*. If there are *n* files in a project, we can split out *n* separate *DRSpaces* using each file as the *leading file*.

2. Distinguish files of different architectural importance. In each *DRSpace*, we use the DRH algorithm to automatically capture the *leading files* and *modules* in a hierarchy based on the selected primary relations. The *leading files* of a DRSpace are clustered in the top layers, while the independent modules, decoupled by the *leading files*, are arranged in the lower layers. For instance, in Fig. 2, the polymorphic *DRSpace* is clustered using "implement" and "extend" as the primary relations. The base classes and interfaces are identified as the *leading files*, including *bnf.Node*, *io.Pipe*, *ast.Node*, and *Filter*, because they are arranged in the first layer (rows 1 to 4) of the DSM. It is easy to see that these are the key files with architectural importance. Similarly, the two files in the top layer of the DSM shown in Fig. 4, *ast.Node* and *ast.TreeVisitor*, are the key interfaces of the visitor pattern. Using DRSpaces, it becomes clear which files take architecturally important roles.

Groups of concrete classes that extend or implement the same base class or interface are decoupled into independent modules in the lower layers, such as file 5 to file 6, and file 7 to file 27 in Fig. 2. These blocks along the diagonal display meaningful modules, including IO pipe module (rc5-6), OutputPipe Module (rc7-8), InputPipe module (rc9-10), ast node module (rc16-20), and bnf node module (rc21-27). Similarly, the "Dp" *DRSpace* in Fig. 3 is clustered using "depend" as the primary relation, while the *Visitor* Pattern *DRSpace* in Fig. 4 is clustered based on "implement", "extend", and "depend" as the primary relations. Each DSM displays a different, but meaningful modular structure.

3. Express structural dependencies and evolutionary couplings among files simultaneously. We model evolutionary couplings among files, extracted from a project's revision history, as another form of architecture connection. In a *DRSpace*, we can express structural dependencies and evolutionary couplings among files simultaneously. Fig. 7 shows such a *DRSpace* from an open source project—Apache Cassandra. In this DSM, the number in each cell indicates the number of times two files changed together as recorded Cassandra's revision history. For example, cell [r1, c2] has "Dp, 34", indicating that *SSTabledp* (row 1) structurally depends on *SSTablereaderdp* (column 2), and they changed together 34 times in the revision history.

Evolutionary couplings and structural dependencies, expressed simultaneously, can help to reveal architectural problems in a *DRSpace*. For example, cell [r11, c3] has "39", indicating the file in row 11 (*ColumnFamilyStoredp*) and the file in column 3 (*SSTableWriterdp*) changed together 39 times in the revision history, but there is no structural dependency between them. This reveals a *modularity violation* [23]. In this *DRSpace*, all the *modularity violations* are highlighted with shaded backgrounds. As we described earlier, *Modularity violations* suggest shared, but implicit assumptions among files. The history couplings can also help to reveal other architectural flaws, as we will discuss further in Section 5.

4 Architecture Root Detection

We envision numerous possible applications of DRSpace modeling, such as recovering design patterns, analyzing how features are implemented, and detecting which parts of the architecture are degrading. We hypothesize that a DRSpace that aggregates and/or propagates large number of bugs due to its flawed architectural structure, could be the root cause of bug-proneness, and we thus call such a DRSpace an *Architecture Root (ArchRoot)*, a specific view of software architecture. In the following, we introduce the approach to detect ArchRoots that automatically capture the most bug-prone *DRSpaces* in a software project.

4.1 Approach Overview

The framework of the *ArchRoot* detection approach is illustrated in Fig. 5. The raw input to this framework include 1) the source code repository (SCR), which contains both the code base and the revision history, and 2) the bug tracking database (BTD). As a preprocessing step, we use the commercial code analysis tool, SciTools Understand,[2] to reverse-engineer the source code of a project to construct the *Proj. DSM*. The DSM captures the structural dependencies between each pair of source files in a project.

The overall flow of our approach is consisted of three steps as shown in Fig. 5. The first step uses the *Proj. DSM* as input to automatically compute a comprehensive set of *DRSpaces*. The second step mines a *Bug Space* from the revision history in *SCR* and the *BTD*. The third step takes both the set of *DRSpaces* and the *Bug Space* as input to greedily detect the *Architecture Roots (ArchRoots)*. In the following, we introduce the details of each step.

4.2 Terms and Concepts

Before we introduce each step of our approach, we first introduce terms and concepts used in this approach.

(1) Bug Space: a list of bug-prone files, ranked by their degree of bug-proneness in descending order. The extraction of a bug space will be introduced in Section 4.3.2. *(2) Design Space Bugginess (dsb)*: refers to the percentage of files in a DRSpace that are also in a bug space. It describes how bug-prone a *DRSpace* is: the more bug-prone files a *DRSpace* contains, the higher its *dsb* value. The *dsb* of a DRSpace is calculated as

$$dsb = \frac{|DRSpace \cap BugSpace|}{|DRSpace|}, \tag{1}$$

where $|DRSpace|$ denotes the number of files in the $DRSpace$; and $|DRSpace \cap BugSpace|$ denotes the number of files in the intersection between $DRSpace$ and $BugSpace$, i.e., the number of bug-prone files in $DRSpace$. The *dsb* is mathematically equivalent to the precision in the context of information retrieval [24]. Precision is the percentage of retrieved documents that are relevant. However, the purpose of this work is not to locate/retrieve bugs. We choose to use the term *dsb* to describe the bug-proneness of a *DRSpace*. This helps to differentiate the purpose of our work from bug prediction or bug localization.

The *dsb* alone is not sufficient in describing the bug-proneness level of a *DRSpace*. For example, it is not valid to claim that a *DRSpace* containing 2 files with *dsb* 100 percent is more bug-prone than a *DRSpace* containing 50 files with *dsb* 50 percent just because 100 percent $>$ 50 percent. Therefore, we further define the second term below.

(3) Bug Space Coverage (bsc): It is the percentage of files in a bug space that are covered by a DRSpace. It describes how comprehensively a *DRSpace* connects the bug-prone files of a project. For example, if a *DRSpace* contains all the bug-prone files in a project, the *bsc* is 100 percent. The *bsc* of a DRSpace is calculated as below:

$$bsc = \frac{|BugSpace \cap DRSpace|}{|BugSpace|}. \tag{2}$$

The denominator is the number of bug-prone files, denoted as $|BugSpace|$, instead of the number of files in the $DRSpace$ as in Equation (1). Similarly, the *bsc* is mathematically equivalent to the recall in the context of information retrieval [24]. Recall is defined as the percentage of the relevant documents being retrieved. The goal of this work is to investigate how bug-prone files in a software system are architecturally connected, instead of locating bug-prone files. Thus, we choose to use *bsc* to differentiate from the context of bug prediction or bug localization.

The *dsb* and *bsc* together describe the bug-proneness level of a *DRSpace*: a *DRSpace* with both high *dsb* and *bsc* is truly bug-prone. For example, a *DRSpace* with a *dsb* of 50 percent and a *bsc* of 30 percent with respect to $BugSpace_5$ means that 50 percent of the files in it are bug-prone, changed for bug-fixing 5 times or more, and this *DRSpace* covers 30 percent of all such bug-prone files in the project. The higher both values are, the more bug-prone a *DRSpace* is, and hence deserving of scrutiny.

(4) Cumulative Bug Space Coverage of a DRSpace set (c_bsc): A project may contain hundreds of bug-prone files and thus may overlap with more than one *DRSpace*. The *c_bsc* of a set of *DRSpaces*, $S = \{DRS_1, DRS_2, \ldots, DRS_n\}$, refers to the percentage of files in a bug space that are cumulatively covered by all the *DRSpaces* in S. The *c_bsc* of S can be calculated as below:

$$c\_bsc = \frac{|(DRS_1 \cup DRS_2 \ldots \cup DRS_n) \cap BugSpace|}{|BugSpace|}. \tag{3}$$

The numerator is the non-duplicated number of bug-prone files contained in the set of DRSpaces, S. The denominator is, again, the number of bug-prone files.

The *c_bsc* of $S = \{S_1, S_2, \ldots, S_n\}$, together with the number of *DRSpaces* in S, denoted by $|S|$, can be used to measure how closely the bug-prone files in a bug space are architecturally connected. The smaller $|S|$ is and the larger the *c_bsc* of S is, the more closely bug-prone files in the $BugSpace$ are architecturally connected.[3]

(5) LOC Normalized c_bsc of a Set of DRSpaces (loc_c_bsc): Ostrand et al. [12] have found that files with larger LOC (Lines of Code) are more likely to be bug-prone. As a result, a set of *DRSpaces* containing files with more LOC are likely to have a higher *c_bsc*, compared to a set of *DRSpaces* containing files with fewer LOC. It is therefore biased to conclude that the former set of *DRSpaces* are more bug-prone than the latter set, because the former contains more LOC. To avoid the interference of LOC, we define *loc_c_bsc*, which is normalized by the LOC of each file. The *loc_c_bsc* of a set of *DRSpaces*, $S = \{S_1, S_2, \ldots, S_n\}$, can be calculated by the following steps:

(1) For each file, f, in a project, we compute

$$f_{weight} = \frac{f_{Bug\_Freq}}{f_{LOC}}. \tag{4}$$

f_{Bug_Freq} is the number of bug revisions on f.
f_{LOC} is the LOC in f.
f_{weight} represents the bug-proneness of f normalized by its LOC.

(2) For a bug space, $BugSpace$, we compute the sum of f_{weight} on all the files in it as below. It describes the total "mass" of bug-proneness in $BugSpace$.

2. https://scitools.com/

3. When the *DRSpace* set S only contains one *DRSpace*, *c_bsc* becomes *bsc*.

$$W_{BugSpace} = \sum_{\forall\, f \in BugSpace} f_{weight}. \quad (5)$$

(3) We compute the sum of f_{weight} on files from the intersection between $S = \{DRS_1, DRS_2, \ldots, DRS_n\}$ and the *BugSpace*. It describes the bug-proneness "mass" contained in the DRSpace set, S.

$$W_{S \cap BugSpace} = \sum_{\forall\, f \in (S_1 \cup S_2 \ldots \cup S_n) \cap BugSpace} f_{weight}. \quad (6)$$

(4) The ***loc_c_bsc*** of $S = \{S_1, S_2, \ldots, S_n\}$ with regards to $BugSpace$ is computed using Equation (6) divided by Equation (5)

$$loc\_c\_bsc = \frac{W_{S \cap BugSpace}}{W_{BugSpace}}. \quad (7)$$

If *loc_c_bsc* is significantly lower than *c_bsc*, this indicates that S just contains a set of very large files in the project. Otherwise, if *loc_c_bsc* $\approx$ *c_bsc*, it indicates that S contains normal size files.

4.3 Three Steps in Approach

This section introduces the details of each step in the approach.

4.3.1 *Step1: Generating* DRSpaces

Before we can identify the *ArchRoots*—the *DRSpaces* aggregating bug-prone files, we first need to represent the architecture of a software system by a comprehensive set of *DRSpaces* to capture all the source files and their dependencies. According to the *DRSpace* definition: a *DRSpace* always contains one or more *leading files*. To generate a comprehensive set of *DRSpaces*, one option is to pick any random set of files as the *leading files* to form a *DRSpace*. Given a system with *n* source files, there are $\sum_{x=1}^{n} \binom{n}{x}$ number of combinations of *leading files* to form *DRSpaces*. The problem is that this cannot be calculated in polynomial time.

Thus we simplify the selection of *leading file* by choosing each source file in a system as a *leading file* to form a *DRSpace*. All the files that structurally depend on the *leading file*, directly or indirectly, are included in its *DRSpace*. If there are *n* files in a project, we will thus generate *n DRSpaces*. The calculation of the *n DRSpaces* done in polynomial time and captures all the source files and their dependencies. Another advantage is that this also allows us to explore the impact of each file as a *leading file* and its influence on the overall bug-proneness of a project, requiring no prior knowledge of a software system. For example, if a *DRSpace* only contains one file, the *leading file* itself, it means that it is unlikely to propagate changes to other files. In comparison, if a *DRSpace* aggregates many files, it means that the leading file has high impact. For example, when an interface is used as the *leading file*, all the concrete classes that implement it, and all client classes that use it, are included in its *DRSpace*. Changes to this interface may propagate to other files in its *DRSpace*.

Thus, in this step, the algorithm called $DRSpaceGenerator$ calculates the *n DRSpaces* of a given project. The pseudo-code of the algorithm is shown in Fig. 1. The input is a DSM, $ProjDSM$, representing all source files and the structural dependencies among files in a project. The output is a comprehensive set of *DRSpaces*, denoted by $DRSpaceSet$. In line 1, $DRSpaceSet$ is initialized to be an empty set. In line 2, we use the DRH clustering algorithm [21], [25] to generate $DrhClusters$, where the files in $ProjDSM$ are clustered into modules according to their structural dependencies. Each $Module$ in $DrhClusters$ contains a group of highly connected files and their connections. In the *for* loop from line 3 to line 13, each file in $ProjDSM$ is used as a *leading file*, to split out a separate *DRSpace*, which contains the *leading file* and modules from $DrhClusters$ that depend on the leading file. As long as one file in $Module$ depends on the leading file, the whole $Module$ is considered to be dependent on the leading file as well, since files in $Module$ are highly connected to each other. By the end of each iteration of the loop, the newly generated *DRSpace* is added to the output $DRSpaceSet$ (line 12). After the *for* loop, in line 14, $DRSpaceSet$ is returned.

4.3.2 *Step 2: Mining Bug Space*

The purpose of the *ArchRoot* detection approach is to investigate the architecture connections and the architectural flaws among bug-prone files in a software system. The first step generates a comprehensive set of *DRSpaces* as the first input to the *ArchRoot* detection algorithm (which will be introduced in step 3). In this step, we focus on calculating the second input, the set of bug-prone files in a project.

This step mines the project repository, including the source code repository and bug tracking database to extract bug-prone files in a project (as shown in Fig. 5). We can link a revision in the project's revision history to a bug report recorded by the issue-tracking database by matching key words in the revision comments left by the developers. For example, it is a convention in Apache (and most other) open source projects that a developer should include a bug ID, from the issue-tracking system, in the commit message when committing a change to fix a bug. Thus, to generate the bug space, we count the number of times each file is involved in bug-fixing changes. We then rank the files by their number of bug-fixing changes, in descending order.

To control the bug-prone level and timeliness of an input bug space to the *ArchRoot* detection, we use two parameters when generating a bug space. The first is the *bug threshold*, which controls the bug-proneness level of a bug space. The second is the *timespan* of the bug space. We will discuss why and how to control each parameter below.

1) Bug threshold. Intuitively, the number of times a file is involved in changes to fix bugs indicates the bug-proneness of this file. For example, if a file has never been involved in bug fixes, it implies that this file is potentially bug-free (the file may of course contain undiscovered bugs). In comparison, if a file is frequently revised (e.g., 90 percent more often than other files) to fix bugs, we know that this file is bug-prone and thus deserves greater attention. However, a file with 5 bug fixes may or may not be considered as bug-prone, depending on the project, the analyst, and the purpose of analysis. Therefore, the perception of bug-proneness is relative and subjective. But it is common sense that a file with 10 bug fixes is more bug-prone than a file with 2 bug fixes.

When controlling the threshold, one option is to control the bug threshold by an absolute frequency value. For example, an analyst may just be interested in the architectural connections among bug-prone files with more than 20 bug fixes. By extracting a bug space with a specified threshold n (which we denote $BugSpace_n$) we get the list of files with at least n bug-fixing changes in a project. For example, a bug space with threshold two, denoted as $BugSpace_2$, is the set of files involved in at least two bug-fixing changes.

Another option is that we can control the bug threshold in the format of a percentile. For example, the set of files that are the top 10 percent most frequently revised to fix bugs are referred to as the top 10 percent bug-prone space, $BugSpace_{10\%}$. Using this percentile scale, we can select bug-prone files at different levels, such as the top 20, and the top 30 percent bug-prone files. As we will discuss later, in the evaluation of this paper, we choose to focus on the top 10, top 20, and top 30 percent bug-prone spaces. These choices allow us to focus on the *most* bug-prone files in a project. Of course, one might also choose to analyze 5 or 55 percent bug spaces. There is no "just-right" choice of threshold. However, as the threshold goes to the 100 percent bug-prone space (i.e., frequency threshold of 1), the files even with only 1 bug fix are included and they may not be truly "bug-prone". That is why this parameter can be chosen based on the interest of the analyst.

2) *Timespan.* In addition, the bug-proneness of a file is a dynamic attribute. For example, a file may be bug-free when it is first introduced in version X. With the evolution of the system, developers start to identify and fix bugs in it starting at version Y. The relative bug-proneness levels of different files may also fluctuate with time. Therefore, when calculating bug-prone files, we also need to confine the time frame we are considering.

A software project typically has multiple releases throughout its lifetime. Different sets of files are revised to fix bugs in different releases. Therefore the bug space of the project for each release dynamically evolves over time. In this study, we consider the bug space of each release as a time window that expands over the previous release. For example, a project, initiated at time-stamp t_0, has n stable releases at time-stamp t_1, t_2, to t_n. The set of files revised to fix bugs from t_0 to t_x, where $0 < x <= n$, forms the bug space of $release_x$. The bug space for a later release $release_{x+1}$ is formed by files revised to fix bugs from t_0 to t_{x+1}.

The above two parameters together define a bug space. For example, Bug_2 between t_0 and t_2 contains the list of files changed for bug-fixing at least twice between t_0 and t_2. By using bug spaces of different timespans and thresholds as inputs, the *ArchRoot* detection algorithm generates *ArchRoots* with different foci. For example, by increasing the threshold of the input bug space, the *ArchRoots* will capture the architectural connections among files with higher bug-proneness levels. By choosing a bug space within recent releases, the detected *ArchRoots* will focus on the architecture connections among files that have been bug-prone recently.

4.3.3 *Step 3: Detecting* ArchRoots

The above two steps allow us to generate the two inputs: 1) a comprehensive set of *DRSpaces* capturing all the files and their inter-dependencies; and 2) a bug space defined by a specific bug-prone threshold and within a certain timespan based on the focus of analysis.

In the third step of the *ArchRoot* detection approach, we analyze the intersections between the *DRSpaces* and a bug space to identify a minimal list of *DRSpaces* that maximally aggregate the bug-prone files in the input bug space. The identified *DRSpaces* are the *ArchRoots* that, as we will show, reveal how bug-prone files are architecturally connected with each other.

The detection of *ArchRoots* is based upon the terms and concepts discussed in Section 4.2. For each *DRSpace* from input 1, we calculate its bsc (bug space coverage) with respect to input 2, the bug space. This helps to show how bug-prone a *DRSpace* is: how many bug-prone files are aggregated in it. The *DRSpace* with the highest bsc is selected to add to the final output: *ArchRoots*. The reason is that we want to focus on *DRSpaces* that maximally connect the bug-prone files. The dsb is also calculated to show what is the percentage of bug-prone files in the *DRSpace*, but it is not considered for the selection. This is because a *DRSpace* containing only 1 file could have maximal, i.e., 100 percent, dsb, but such a *DRSpace* is not helpful to reveal how bug-prone files are architecturally connected. Files contained in the selected *DRSpace* are excluded from the original bug space. The selection from the remaining (unselected) *DRSpaces* continues until all the bug-prone files are included in the final output *ArchRoots*. While the *DRSpaces* are selected, the c_bsc and loc_c_bsc of the detected *ArchRoots* are updated to show the total percentage of bug-prone files aggregated, and the selection process ends when c_bsc reaches 100 percent.

The *ArchRoot* detection problem is actually the classic Knapsack problem, which is NP-complete. The solution described above follows a greedy process such that in each step a local optimum is selected, which may not lead to a global optimum. The top few *ArchRoots* have approximately maximal coverage of the bug-prone files. Although it is not algorithmically optimal, as we will show in the evaluation, it is very helpful to understand how bug-prone files are architecturally connected.

The pseudo-code of the *ArchRoot* detection is shown in Algorithm 2. The algorithm outputs a list of *DRSpaces*, denoted by $ArchRoots$, which architecturally connects the bug-prone files in $BugSpace$. In line 1, the output, $ArchRoots$, is initialized to be empty. In line 2, the input $BugSpace$ is copied to $RemainingBugSpace$, which contains the remaining files in the original $BugSpace$ that are to be covered by the output $ArchRoots$. In the while loop from line 3 to line 12, the *DRSpaces* from $DRSpaceSet$ are added to the tail of $ArchRoots$ one by one until $RemainingBugSpace$ is empty (meaning all the bug-prone files in $BugSpace$ are covered by $ArchRoots$). In each iteration of the while loop, a *DRSpace* from $DRSpaceSet$ with a maximal *bsc* with respect to $RemainingBugSpace$ is selected as $MaxBscSpace$ (line 4) and the $MaxBscSpace$ is removed from the $DRSpaceSet$ (line 5). The goal of selecting the $MaxBscSpace$ is to greedily find a minimal number of *DRSpaces* that maximally cover the input $BugSpace$. Then, in lines 6 and 7, the dsb and bsc of $MaxBscSpace$ are calculated. In line 8, $MaxBscSpace$ is add to the end of $ArchRoots$. In line 9, all the files in $MaxBscSpace$ are removed from the $RemainingBugSpace$ (meaning these bug-prone files have been covered by current $ArchRoots$).

Finally, in lines 10 and 11, the *c_bsc* and the *loc_c_bsc* of current *ArchRoots* with respect to the original *BugSpace* are calculated. When all the files in *BugSpace* are covered by *ArchRoots*, in line 13, the result *ArchRoots* is returned.

In the following section, we will investigate how the identified *ArchRoots* can help advance our understanding of the relationships between software architecture and bug-proneness.

Algorithm 1. DRSpaceGenerator ($ProjDSM$)

```
1: $DRSpaceSet \leftarrow \emptyset$    ▷ Initialize the return value to be an empty set
2: $DrhClusters \leftarrow DrhAlgorithm(ProjDSM)$    ▷ Cluster the input ProjDSM into Design Rule Hierarchy [21], [25]
3: for each $file$ in $ProjDSM$ do
4:     $DRSpace \leftarrow \emptyset$    ▷ Create a new empty $DRSpace$
5:     $DRSpace.LeadingFile \leftarrow file$    ▷ Set the leading file of $DRSpace$ to be the current $file$ in the loop
6:     for each $Module$ in $DrhClusters$ do
7:         if $Module$ depend on $LeadingFile$ in $ProjDSM$ then
8:             $DRSpace.addVertices(Module.vertices())$ ▷ Add files in $Module$ into $DRSpace$.
9:             $DRSpace.addEdges(Module.edges())$ ▷ Add connections among files in $Module$ into $DRSpace$.
10:        end if ▷ A $Module$ in $DrhClusters$ is a cluster of highly connected files.
11:    end for
12:    $DRSpaceSet$.add($DRSpace$)    ▷ Add the newly constructed $DRSpace$ into the return set $DRSpaceSet$.
13: end for
14: return $DRSpaceSet$    ▷ Return the comprehensive set of DRSpaces.
```

5 Evaluation

In this section, we present our evaluation procedure and results using 15 open source projects. We will use the *dsb* and *bsc* measures (equivalent to the notions of *precision* and *recall* in information retrieval) to describe how the top bug-prone files in a software system are architecturally connected. Ultimately, we aim to demonstrate how DRSpace modeling and ArchRoot analysis can reveal the significant impact of architecture on software quality, through the answers to our four research questions.

5.1 Subjects and Data Selection

We chose 15 Apache open source projects as our evaluation subjects. Those projects differ in size, application domain, length of history, and other characteristics.

They are: Avro[4] – a data serialization system; Camel[5] – an integration framework based on known Enterprise Integration Patterns; Cassandra[6] – a distributed database management system; CXF[7] – a fully featured Web services framework using APIs like JAX-WS and JAX-RS; Derby[8] – a relational database implemented entirely in Java; Hadoop[9] – a framework for reliable, scalable, distributed computing; HBase[10] – the Hadoop database, a distributed, scalable, big data store; Mahout[11] – a scalable machine learning application; MINA[12] – a network application framework which helps users develop high performance and high scalability network applications easily; OpenJPA[13] – an implementation of the Java Persistence API specification; PDFBox[14] – a Java library for working with PDF documents; Pig[15] – a platform for creating MapReduce programs used with Hadoop; Tika[16]– a content analysis toolkit; Wicket[17] – a lightweight component-based web application framework; ZooKeeper[18] – a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.

Algorithm 2. $ArchitectureRootDetector$ (DRSpaceSet, BugSpace)

```
1: ArchRoots $\leftarrow \emptyset$;    ▷ Initialize the return value $ArchRoots$ to be an empty list.
2: RemainingBugSpace $\leftarrow$ BugSpace;    ▷ Assign $BugSpace$ to $RemainingBugSpace$, which contains bug-prone files to be covered.
3: while RemainingBugSpace $\neq \emptyset$ do
4:     MaxBscSpace $\leftarrow$ $Select\_Max\_Bsc\_Space$(DRSpaceSet, RemainingBugSpace)    ▷ Select the $DRSpace$ from input $DRSpaceSet$ with max $bsc$ for the $RemainingBugSpace$.
5:     DRSpaceSet.$Remove$(MaxBscSpace);    ▷ Remove the $MaxBscSpace$ from the $DRSpaceSet$
6:     $Calculate\_dsb$(MaxBscSpace)    ▷ Calculate the $dsb$ of the $MaxBscSpace$
7:     $Calculate\_bsc$(MaxBscSpace)    ▷ Calculate the $bsc$ of the $MaxBscSpace$
8:     ArchRoots.$Add\_to\_Tail$(MaxBscSpace)    ▷ Add $MaxBscSpace$ to the tail of the return list $ArchRoots$
9:     RemainingBugSpace.$Remove\_All$(MaxBscSpace.$get\_Files$())    ▷ Remove all covered bug-prone files from $RemainingBugSpace$
10:    $Calculate\_c\_bsc$(MaxBscSpace, BugSpace) ▷ Calculate the c_bsc of the current $ArchRoots$
11:    $Calculate\_loc\_c\_bsc$(MaxBscSpace, BugSpace) ▷ Calculate the loc_c_bsc of the current $ArchRoots$
12: end while    ▷ The $while$ loop stops when there is no bug-prone files remained in $RemainingBugSpace$ to be covered.
13: return ArchRoots ▷ Return the identified list of $ArchRoots$
```

Summary information for these projects is given in Table 1. The first column shows the project names. The second column shows the length of each project's history covered by our study, denoted by a start time-stamp to an end time-stamp, and the number of months within the history (in parentheses). The third and fourth columns show the number of releases and the latest release we selected in each project. The column "#Commits" is the number of revisions made to each project

4. https://avro.apache.org/
5. http://camel.apache.org/
6. http://cassandra.apache.org/
7. http://cxf.apache.org/
8. https://db.apache.org/derby/
9. https://hadoop.apache.org/
10. http://hbase.apache.org/
11. https://mahout.apache.org/
12. https://mina.apache.org/
13. http://openjpa.apache.org/
14. https://pdfbox.apache.org/
15. https://pig.apache.org/
16. https://tika.apache.org/
17. https://wicket.apache.org/
18. https://zookeeper.apache.org/

TABLE 1
Summary of Evaluation Projects

Subject	Length of history (#Months)	#Releases	Latest Release#	#Commits	#Developers	#Issues	#Files
Avro	8/2009 to 1/2014 (53)	12	1.7.6	1,115	17	734	156 to 426
Camel	7/2008 to 7/2014 (72)	12	2.12.4	14,563	106	2,790	1,838 to 9,866
Cassandra	9/2009 to 11/2014 (62)	10	2.1.2	14,673	122	4,731	311 to 1,337
CXF	12/2007 to 5/2014 (77)	13	3.0.0	8,937	46	3,854	2,861 to 5,509
Derby	10/2007 to 8/2014 (83)	13	10.11.1.1	4,275	23	2,726	2,388 to 2,776
Hadoop	8/2009 to 8/2014 (60)	9	2.5.0	8,253	75	5,443	1,307 to 5,488
HBase	12/2009 to 5/2014 (53)	9	0.98.2	6,718	37	6,280	560 to 2,055
Mahout	10/2008 to 2/2014 (64)	9	0.9	3,113	22	658	455 to 1,262
MINA	10/2005 to 10/2009 (49)	8	2.0.5	1,760	19	467	219 to 550
OpenJPA	2/2007 to 4/2013 (74)	11	2.2.2	6,098	25	1,572	1,266 to 4,314
PDFBox	8/2009 to 9/2014 (62)	12	1.8.7	2,005	16	1,857	447 to 791
Pig	3/2008 to 1/2012 (47)	10	0.9.2	1,668	19	2,579	302 to 1,195
Tika	6/2008 to 1/2015 (80)	15	1.7	2,412	17	714	131 to 550
Wicket	6/2007 to 1/2015 (92)	15	6.19.0	8,309	65	3,557	1,879 to 3,081
ZooKeeper	4/2008 to 11/2012 (55)	10	3.4.5	1,012	10	1,154	151 to 382

during the selected time range. The column "#Developers" is the number of developers who were submitting changes to each project. Both "#Commits" and "#Developers" are extracted from the version control systems of the projects: either from SubVersion[19] or Git.[20] The column "#Issues" is the number of bug reports in each project, which is extracted from the JIRA[21] bug-tracking database of each project. The last column shows the size range of each project, measured by the number of files in the first release and the last release.

As discussed earlier, with input bug spaces of different time-spans and bug thresholds, the *ArchRoot* detection algorithm generates *ArchRoots* of different foci. In this section we will explain the rationale of our experiment setting: how we selected the time-spans and bug thresholds of the input bug spaces for the evaluation.

Release Timespan. For each project, we studied multiple releases, as shown in the third column of Table 1, to investigate the variance of *ArchRoots* over time. As introduced earlier, for each release x in a project, we consider files revised to fixed bugs between t_0 (fixed starting point) and t_x (release time) as the bug space of the release. The stable releases of each project are carefully selected such that any two consecutive releases have an interval of 4 to 6 months. The timestamp t_0 is thus controlled to be 6 months before the first selected release in each project.

Bug-proneness Level. For each project, we study two levels of bug spaces, denoted in percentile scales: $Bug_{30\%}$ and $Bug_{10\%}$, to focus on the most bug-prone files in a project. As mentioned earlier, the perception of bug-prone is relative and even subjective sometimes. This selection is a general choice to focus on the *most* bug-prone files in a project. Based on specific interests of analysis, one can also choose to analyze 35 or 55 percent bug spaces. There is no "the-just-right" choice of threshold. Project insiders may have better knowledge to fine tune the bug-prone level of the input bug space to address their concerns. In evaluating the bug-proneness level of a *DRSpace*, we investigate the *bsc* and *dsc* with respect to these two levels of bug spaces.

19. http://subversion.apache.org/
20. http://git-scm.com/
21. https://issues.apache.org/jira/secure/Dashboard.jspa

For each release of each project, we extracted the two bug spaces ($Bug_{30\%}$ and $Bug_{10\%}$). *ArchRoots* identified from the two bug spaces in different releases focus on the architectural connections among bug-prone files of different time-spans and bug-proneness levels.

5.2 Research Questions and Finding Summary

To evaluate the effectiveness of our *DRSpace* model and *ArchRoot* detection algorithm, we investigated four research questions, addressing the significance and persistence of ArchRoots, quantitatively and qualitatively. We summarize the major findings for these questions as follows.

RQ1: If a leading file is bug-prone, are the files within its DRSpace also likely to be bug-prone, that is, more bug-prone than other spaces? A positive finding implies that a bug-prone leading file has significant impact on the bug-proneness of the files architecturally connected to it.

Finding: We investigated the *bsc* and *dsb* of the *DRSpaces* led by the top 30 percent bug-prone files in each project, and found that on average from 10 to 41 percent of files in these *DRSpaces* are also ranked within top 30 percent most bug-prone. In particular, *DRSpaces* led by buggy leading files are on average *2 to 3.9 times* more bug-prone than two baselines: 1) *DRSpaces* led by bug-free leading files and 2) randomly generated spaces with equal sizes. Thus it appears that if a design rule/leading file is bug-prone, files architecturally aggregated into its DRSpace are more likely to be bug-prone than other spaces. Hence, high-impact bug-prone leading files should be given higher priority in quality assurance, bug fixing, or refactoring activities.

RQ2: To what extent are bug-prone files architecturally connected? This question investigates whether most bug-prone files can be captured in just a few ArchRoots. As we discussed earlier, the number of *ArchRoots* and the percentage of bug-prone files covered by the roots, together, reflect the strength of the architecture connections among those bug-prone files.

Finding: We investigated the *c_bsc* and the *loc_c_bsc* of the *ArchRoots* with respect to $Bug_{30\%}$ and $Bug_{10\%}$. We found that a significant portion of bug-prone files are indeed captured by just a few—usually five—DRSpaces, which only account for a relative small portion of files in a project.

TABLE 2
DRSpaces Led by Buggy Leading Files versus Two Baselines

Project (Release#)	$Bug_{30\%}$										$Bug_{10\%}$									
	Bug Space Coverage (bsc)					Design Space Bugginess (dsb)					Bug Space Coverage (bsc)					Design Space Bugginess (dsb)				
	B-DRS	BSL1	Rt.	BSL2	Rt.	B-DRS	BSL1	Rt.	BSL2	Rt.	B-DRS	BSL1	Rt.	BSL2	Rt.	B-DRS	BSL1	Rt.	BSL2	Rt.
Avro	22%	17%	1.3	15.2%	1.4	15.3%	15.2%	1	11.1%	1.4	30.0%	24.2%	1.2	14.4%	2.1	7.1%	7.2%	1	3.4%	2.1
Camel	7%	1%	10.4	3.7%	1.8	22.6%	2.4%	9.5	5.3%	4.3	8.9%	0.8%	11.5	4.0%	2.3	11.3%	1.0%	11.6	1.9%	5.9
Cassandra	17%	10%	1.6	7.9%	2.2	38.8%	34.1%	1.1	17.2%	2.2	19.2%	11.4%	1.7	8.1%	2.4	26.2%	21.9%	1.2	10.2%	2.6
CXF	5%	1%	4.1	1.9%	2.5	23.4%	6.8%	3.4	7.9%	2.9	7.3%	1.6%	4.7	1.8%	4	12.6%	3.0%	4.2	2.5%	5
Derby	11%	4%	2.6	3.7%	3.1	38.0%	19.7%	1.9	11.2%	3.4	17.1%	6.4%	2.7	3.5%	4.9	20.0%	9.7%	2.1	3.2%	6.2
Hadoop	8%	1%	9.4	4.0%	2	10.0%	2.4%	4.2	3.7%	2.7	12.6%	0.9%	13.3	4.1%	3.1	4.0%	0.7%	6.1	1.0%	3.9
HBase	9%	3%	3.5	5.4%	1.7	33.7%	12.5%	2.7	17.6%	1.9	12.9%	2.8%	4.5	5.7%	2.2	20.2%	6.1%	3.3	7.5%	2.7
Mahout	9%	2%	4.7	6.0%	1.5	24.6%	8.5%	2.9	16.9%	1.5	14.2%	2.2%	6.3	8.4%	1.7	12.4%	2.6%	4.7	6.4%	2
MINA	33%	35%	1	21.0%	1.6	19.4%	16.7%	1.2	9.1%	2.1	41.2%	39.2%	1.1	21.6%	1.9	16.5%	12.7%	1.3	5.5%	3
OpenJPA	5%	2%	2.7	2.3%	2.3	41.6%	24.1%	1.7	17.3%	2.4	9.6%	2.8%	3.4	2.3%	4.2	31.7%	16.5%	1.9	7.2%	4.4
PDFBox	23%	20%	1.1	14.1%	1.6	30.6%	29.5%	1	18.1%	1.7	25.0%	28.6%	0.9	14.6%	1.7	12.0%	16.9%	0.7	6.4%	1.9
Pig	16%	10%	1.6	7.8%	2.1	29.6%	17.8%	1.7	13.5%	2.2	20.4%	13.0%	1.6	7.9%	2.6	11.8%	7.4%	1.6	4.5%	2.7
Tika	35%	28%	1.3	12.6%	2.8	21.4%	19.3%	1.1	7.8%	2.7	44.2%	34.0%	1.3	11.7%	3.8	11.8%	10.0%	1.2	2.5%	4.7
Wicket	8%	3%	2.9	4.6%	1.8	26.0%	13.8%	1.9	11.5%	2.2	10.9%	3.5%	3.1	5.1%	2.2	12.0%	6.1%	2	4.4%	2.8
ZooKeeper	33%	19%	1.8	15.0%	2.2	29.2%	17.8%	1.6	10.0%	2.9	39.2%	25.3%	1.6	14.8%	2.6	16.3%	11.0%	1.5	4.7%	3.5
All Projs Avg	16%	10%	3.3	8%	2.0	27%	16%	2.5	12%	2.4	21%	13%	3.9	9%	2.8	15%	9%	3.0	5%	3.6
t-test p-value		7.2E-06		1.5E-5			1.0E-05		3.3E-08			5.4E-06		3.4E-05			1.8E-4		1.6E-06	

Note:
B-DRS *stands for DRSpaces led by a bug-prone leading file.*
BSL1 *stands for baseline 1—DRSpaces led by a bug-free leading file.*
BSL2 *stands for baseline 2—random generated spaces with equal sizes to "B-DRS".*
Rt. *stands for the ratio between the* bsc *or* dsb *of "BSL1" or "BLS2" divided by that of "B-DRS"*

Specifically, from 35 to 91 percent of the $Bug_{30\%}$ files in the 15 projects are captured by the top five *ArchRoots*, which contain 7 to 61 percent of files in the projects. These bug-prone files are *1.5 to 1.9 times* as likely to be architecturally connected to each other compared to average files in a project. The implication is that when software practitioners try to fix bugs, instead of treating bug-prone files individually, they should focus on just a few *ArchRoots* and be aware of the architecture connections that may propagate bugs among multiple files. The developers should pay attention to ArchRoots with the highest concentration of bug-prone files, as they may be able to remove or prevent multiple bugs simultaneously by refactoring.

RQ3: Are these ArchRoots *long-lasting and persistent during a project's evolution?* This question investigates if ArchRoots can survive multiple releases of a project. A positive finding to this question implies that the architectural impact of *ArchRoots* is persistent over time. In other words, the over-all bug-proneness of a project can not be fundamentally reduced while long-lived *ArchRoots* persist.

Finding: Considering *ArchRoots* identified in different releases with the same *leading files* as snapshots of the same *ArchRoot*, we have observed 1 to 5 long-lived ArchRoots in each of the 15 studied projects. These *ArchRoots* survived at least 40 percent of the total number of releases in a project. These long-lived ArchRoots are usually led by a top ranked bug-prone leading file and they are 3 times more bug-prone than average, similarly-sized groups of files. The data shows that these long-lived *ArchRoots* should be the top priority for developers to fix (or reconstruct) if they want to reduce the over-all bugginess of their project in the long run.

RQ4: Do these ArchRoots *contain architectural flaws that may contribute to their bug-proneness?* Mo et al. [26] reported that there is a strong positive correlation between project bug-proneness and architectural flaws: files involved in greater numbers of architecture flaws are more bug-prone than average files. Here we plan to analyze the architectural flaws in *ArchRoots*, and to understand how these flaws contribute to the over-all bug-proneness of software projects.

Finding: We have observed recurring architectural flaws—such as cyclic dependencies, unhealthy inheritance, unstable interfaces, and modularity violations [26]–in every project we have studied. Our studies [26], [27] revealed that these flaws are the *root causes* of high bug rates because they can propagate changes among files, making bugs hard to eradicate.

5.3 Detailed Data Analysis

We now elaborate on the experiment data and detailed answers to these research questions.

RQ1: If a leading file is bug-prone, are the files within its DRSpace also likely to be bug-prone, that is, more bug-prone than other design spaces?

To answer this question, we investigated the *bsc* and *dsb* of the DRSpaces led by the top 30 percent bug-prone files (referred to as BL-DRSpaces in the following context) in each project. We only considered spaces with more than 30 files, otherwise the leading files are unlikely to be architecturally important if its impact scope is too small. We evaluated whether BL-DRSpaces have high *bsc* and *dsb* with respects to $Bug_{30\%}$ and $Bug_{10\%}$ respectively. In addition, to examine whether BL-DRSpaces are particularly bug-prone, we compare their bsc and *dsb* with that of two baselines: 1) the DRSpaces led by bug-free leading files and 2) random generated spaces with equal numbers of files, which are referred to as *baseline 1* and *baseline 2* respectively in the following.

TABLE 3
Top Five *ArchRoots* c_bsc versus Project Baseline

Project	$Bug_{30\%}$			$Bug_{10\%}$			Baseline: Top 5 DRS			
	#BF	c_bsc	loc_c_bsc	#BF	c_bsc	loc_c_bsc	#TF	coverage	$R_{30\%}$	$R_{10\%}$
Avro	46	89%	89%	16	94%*(2)	95%*(2)	426	57%	1.6	1.7
Camel	496	38%	31%	194	54%	41%	9,866	9%	4.1	5.9
Cassandra	347	58%	82%	161	67%	81%	1,337	42%	1.4	1.6
CXF	492	43%	35%	254	56%	57%	5,509	15%	2.9	3.8
Derby	318	60%	49%	130	74%	57%	2,776	24%	2.5	3.1
Hadoop	209	41%	44%	73	67%	63%	5,488	17%	2.4	4
HBase	521	40%	30%	160	64%	64%	2,055	30%	1.3	2.1
Mahout	312	43%	50%	149	56%	68%	1,262	47%	0.9	1.2
MINA	59	91%*(4)	94%*(4)	18	94%*(2)	99%*(2)	279	74%	1.2	1.3
OpenJPA	770	35%	18%	210	79%	56%	4,314	18%	2	4.5
PDFBox	270	69%	62%	101	68%	91%	791	62%	1.1	1.1
Pig	196	66%	85%	91	68%	91%	1,195	51%	1.3	1.3
Tika	81	81%	86%	31	87%	88%	550	55%	1.5	1.6
Wicket	411	43%	45%	194	57%	60%	3,081	24%	1.8	2.4
ZooKeeper	51	86%	62%	21	95%*(3)	95%*(3)	382	59%	1.4	1.6
All Proj. Avg	305	59%	57%	120	72%	74%	2,621	39%	1.5	1.9
t-test p-value		2.9E-06			1.1E-06					

Note:
*(n) *means that the c_bsc by the top n ($n < 5$) ArchRoots is already maximal.*
Baseline *refers to the top 5 DRSpaces covering a maximal number of average files in a project.*
#TF *stands for the total number of files in a project.*
#BF *stands for the number of files in a respective bug space.*
c_bsc *stands for the cumulative bug space coverage by a set of* DRSpaces.
loc_c_bsc *stands for the LOC normalized c_bsc.*
coverage *stands for the cumulative coverage of general files in a project by the top 5 DRSpaces.*
$R_{30\%}$ *is the c_bsc of top 5 ArchRoots for $Bug_{30\%}$ divided by the c_bsc of the baseline.*
$R_{10\%}$ *is the c_bsc of top 5 ArchRoots for $Bug_{10\%}$ divided by the c_bsc of the baseline.*

The evaluation results are summarized in Table 2. Columns 2-7 shows the *average bsc* and *dsb* with respect to $Bug_{30\%}$ of all the BL-DRSpaces (columns 2, 5) in each project and the comparison with the two baselines (columns 3, 4, 6, and 7). Similarly, columns 8-13 shows the same information with respect to $Bug_{10\%}$.

The data show that the files within BL-DRSpaces are also bug-prone: The average *dsb* of BL-DRSpaces with respect to $Bug_{30\%}$ and $Bug_{10\%}$ indicates that from 10 to 41.6 percent (column 5) of the files in BL-DRSpaces also rank within top 30 percent most bug-prone, and from 4 to 31.7 percent (column 11) of files in BL-DRSpaces are ranked among top 10 percent most bug-prone.

We compare these *dsb* values with that of the two baselines (i.e., *DRSpaces* led by bug-free leading files and random generated spaces of equal sizes). The comparison indicates that the BL-DRSpaces are *2.4 to 3.6 times* more bug-prone than the two baseline spaces. The p-value of the paired t-test ($p < 0.05$, shown in the last row of Table 2) indicates that the differences between the *dsb* of BL-DRSpaces with the two baselines are *statistically significant*.

These observations are consistent with our intuitive understanding that files led by bug-prone leading files are also likely to be bug-prone—and are statistically more bug-prone than the two baseline spaces—since bugs in architecturally important design rules are likely to propagate.

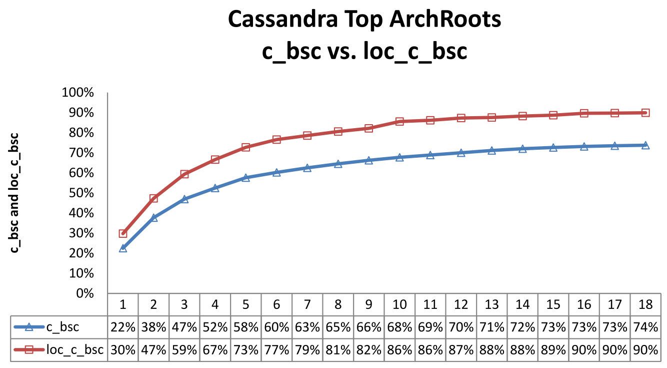
We also analyzed what percentage of the total number of bug-prone files in a project (i.e., *bsc*) are covered in BL-DRSpaces. Columns 2 and 8 show that on average from 5 to 35 percent of all the top 30 percent bug-prone files and from 8.9 to 44.2 percent of all the top 10 percent most bug-prone files in a project are concentrated in just one BL-DRSpace.

In addition, the *bsc* of BL-DRSpaces are *2 to 3.9 times* higher than the two baselines, which is significant (paired t-test p-value < 0.05). This result suggests that bug-prone files in BL-DRSpaces are strongly connected groups rather than isolated individuals.

We now can confidently answer RQ1: If a design rule is bug-prone, files architecturally aggregated in its DRSpace*are also likely to be bug-prone—and are statistically significant more bug-prone than the two baseline spaces.* The take-away message from RQ1 is twofold. On the one hand, bug-prone and high-impact design rules should be given higher priority in quality assurance activities because of their significant impacts on large numbers of files in their *DRSpaces*. On the other hand, instead of treating bug-prone files in isolation, developers should consider such files as groups related by architectural connections: design rules.

RQ2: To what extent are bug-prone files architecturally connected? There could be hundreds of files involved in frequent bug-fixing activities in a large project. As shown in columns 2 and 5 of Table 3, there are 46 (AVRO) to 770 (OpenJPA) files in $Bug_{30\%}$ and 16 to 254 (CXF) files in $Bug_{10\%}$ for each project.

By definition, each DRSpace is a group of architecturally connected files. Therefore if a few *DRSpaces*, collectively, aggregate a large portion of a project's bug-prone files, this indicates that these bug-prone files are architecturally connected. The fewer the number of *ArchRoots* and the larger percentage of bug-prone files covered by these ArchRoots,



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
c_bsc	22%	38%	47%	52%	58%	60%	63%	65%	66%	68%	69%	70%	71%	72%	73%	73%	73%	74%
loc_c_bsc	30%	47%	59%	67%	73%	77%	79%	81%	82%	86%	86%	87%	88%	88%	89%	90%	90%	90%

Fig. 6. The c_bsc of the top 18 ArchRoots in cassandra-$Bug_{30\%}$.

the more strongly these bug-prone files are architecturally connected, and these ArchRoots thus deserve more attention.

Our study results show that a significant portion of bug-prone files are indeed captured by just a few—usually five—DRSpaces. We now use the $Bug_{30\%}$ *ArchRoots* in Cassandra as a detailed illustrating example. Fig. 6 visualizes the c_bsc and loc_c_bsc of the top 18 ArchRoots for $Bug_{30\%}$ as of release 2.1.2. The x-axis represents the top x ArchRoots, while the y-axis represents the c_bsc and loc_c_bsc by the top x ArchRoots. The two lines for c_bsc and loc_c_bsc increase sharply within the top five *ArchRoots* and slowly flatten out. For example, as the number of ArchRoots went from 5 to 18, the c_bsc only increased from 58 to 74 percent and loc_c_bsc only increased from 73 to 90 percent. This indicates that the top five ArchRoots capture the most bug-prone files. We have made similar observations in virtually every project we have studied, open source or commercial: the top five ArchRoots typically cover a majority of the project's bug-prone files.

Table 3 shows the c_bsc and loc_c_bsc of the top five ArchRoots for $Bug_{30\%}$ and $Bug_{10\%}$ in the 15 projects. Starting from the most bug-prone files, the top five $Bug_{10\%}$ ArchRoots aggregate from 54 to 95 percent (column 6) of the all files in $Bug_{10\%}$. In particular, the top 10 percent most bug-prone files in AVRO are aggregated in just two *ArchRoots*. Similar for MINA and ZooKeeper, their top 10 percent most bug-prone files are aggregated in two and three *ArchRoots*. In CXF, which has 254 $Bug_{10\%}$ files, the top five $Bug_{10\%}$ *ArchRoots* aggregate 56 percent of these files. In other words, each of the top five *ArchRoots* in CXF on average contains 28.4 top 10 percent bug-prone files.

Now we examined the top five $Bug_{30\%}$ ArchRoots. As shown in Table 3, from 35 to 91 percent (column 5) of the files in $Bug_{30\%}$ are concentrated in the top five ArchRoots. In MINA, the top four ArchRoots cover 91 percent of the bug-prone files (marked as $94\%^{*}(4)$ in Table 3).

These data provide strong evidence that the majority of bug-prone files are architecturally connected in just a few ArchRoots. In addition, the loc_c_bsc and c_bsc are consistent with each other, indicating that the top five $Bug_{10\%}$ and $Bug_{30\%}$ *ArchRoots* are not simply groups of very large files. Hence, these ArchRoots should merit special attention in quality assurance activities due to their significant coverage of bug-prone files.

To strengthen the significance of our findings, we further conducted a baseline experiment to examine the extent to which the general files in a project are architecturally connected. If, in general, files in a software system are architecturally connected in a few *DRSpaces*, it would not be surprising to see that bug-prone files are architecturally connected in a few *ArchRoots*. For example, if 50 percent of the entire file set are connected in just 5 *DRSpace*, it is not surprising to see that 50 percent of bug-prone files are connected in just 5 *ArchRoots*, because bug-prone files are just part of the entire file set of a system. In comparison, if 80 percent of bug-prone files are aggregated in just 5 *ArchRoots*; while only 50 percent of the entire file set aggregated in these top 5 *DRSpaces*, it implies that the architectural connections among bug-prone files are more significant compared to the entire file set. To do this, we replace the second input, the *Bug Space*, to the *ArchRoot* detection algorithm by the entire file set of a project. The output of the detection algorithm thus becomes a minimal set of *DRSpaces* that maximally cover the entire file set. This helps to understand the extent to which general files in a project are architecturally connected to each other. The top few ArchRoots won't be as significant if general files in a project are—compared to bug-prone files—equally or more closely connected to each other.

Therefore, we compare the *c_bsc* for bug prone files by the top 5 *ArchRoots* with the *c_bsc* for the entire file set by the top 5 *DRSpaces* from the baseline experiment. We calculate the ratio of the former *c_bsc* divided by the latter *c_bsc* to show how much more likely bug-prone files are architecturally connected compared to general files. The comparison results are shown in columns 8-11 in Table 3. As summarized in row "All Proj. Avg.", on average 39 percent of the files in a project are architecturally connected in the top 5 DRSpaces. But, on average 72 percent of the $Bug_{10\%}$ files and 59% $Bug_{30\%}$ files are connected in the top 5 ArchRoots. *Thus, bug-prone files are, on average, 1.5 to 1.9 times (as shown as $R_{30\%} = 1.5$ and $R_{10\%} = 1.9$ in the Table) more likely to be architecturally connected to each other than average files in a project*. Again, this comparison is statistically significant (paired t-test p-value < 0.05).

However, this property of the top 5 ArchRoots still won't be as meaningful **if** the high c_bsc of the bug-prone files is simply because they contain a comparable, or even higher, portion of the total number of files in a project. In Table 4 we compare the size of the top 5 ArchRoots (in terms of the number and percentage of files in each project that they contain) with their c_bsc to $Bug_{30\%}$ and $Bug_{10\%}$. As summarized in row "All Proj. Avg.", in the 15 projects, on average the top 5 ArchRoots for $Bug_{30\%}$ and $Bug_{10\%}$ contain 26 and 33 percent of all the files in a project, while covering 59 and 72 percent of the top 30 and top 10 percent bug-prone files respectively. The paired t-test suggests that the difference between the percentage of files and the c_bsc is statistically significant (as shown in the last row, p-value < 0.05). *Hence, the top 5 ArchRoots contain a relatively small portion of files, but cover a relative large portion of the bug-prone files.*

In summary, we can answer RQ2 with confidence: the top five ArchRoots, which contain on average 33 percent of files in a project capture a significant portion (on average 59 percent) of the bug-prone files. And the bug-prone files in these ArchRoots are more likely (on average 1.5 to 1.9 times) to be architecturally connected to each other compared to general files in a project. The implication is that the top five $Bug_{10\%}$ and $Bug_{30\%}$ *ArchRoots* merit special attention, because they aggregate the most bug-prone files. The architectural connections among these top ranked bug-prone files are, we argue, the root causes of bug-proneness.

TABLE 4
Top Five *ArchRoots* c_bsc versus Size

Project	$Bug_{30\%}$			$Bug_{10\%}$		
	#F	%F	c_bsc	#F	%F	c_bsc
Avro	160	38%	89%	82	19%	94%
Camel	688	7%	86%	537	5%	95%
Cassandra	529	40%	41%	482	36%	67%
CXF	638	12%	43%	664	12%	57%
Derby	600	22%	35%	499	18%	79%
Hadoop	597	11%	81%	564	10%	87%
HBase	504	25%	91%	434	21%	94%
Mahout	593	47%	43%	453	36%	56%
MINA	170	61%	38%	169	61%	54%
OpenJPA	549	13%	66%	549	13%	68%
PDFBox	406	51%	60%	279	35%	74%
Pig	528	44%	40%	332	28%	64%
Tika	262	48%	43%	227	41%	56%
Wicket	623	20%	69%	612	20%	68%
ZooKeeper	197	52%	58%	160	42%	67%
All Projs. Avg	470	33%	59%	403	26%	72%
t-test p-value		0.0097			1.1E-05	

Note:
#F *stands for the number of files contained in the top 5 ArchRoots.*
%F *stands for the percentage of all the files in the top 5 ArchRoots.*
c_bsc *stands for the cumulative bug space coverage by the top 5 ArchRoots.*

RQ3: Are ArchRoots *long-lasting and persistent during project evolution?* We consider *ArchRoots* identified in different releases of a project with the same *leading file* as different snapshots of the same root, and use the number of snapshots as the *age* of an *ArchRoot*, to measure how long-lasting an *ArchRoot* is.

We investigated the evolution of the identified *ArchRoots* for $Bug_{30\%}$ and $Bug_{10\%}$ in all 15 projects, and observed multiple *long-lived ArchRoots* in each of these projects. The ages of these *ArchRoots* are, on average, 40 percent of the number of releases in a project. For example, if there are nine releases in a project, these *long-lived ArchRoots* have survived for at least four releases.

In Table 5, we list 47 *long-lived ArchRoots* for $Bug_{30\%}$ in the 15 projects. To illustrate the characteristics of these long-lived *ArchRoots*, Table 5 shows the leading file and the number of files contained in each root (column 2), the age and maximum possible age (total number of releases in a project) of each root (column 3), the bug-proneness ranking of the leading file (column 4), the bug space coverage (columns 5 to 7), and the design space bugginess (columns 8 to 10) of each root. We can make the following observations from Table 5.

The long-lived ArchRoots *are usually led by a bug-prone leading file.* For example, 31 of the 47 long-lived *ArchRoots* are led by files in $Bug_{30\%}$. And 15 long-lived *ArchRoots* are led by files in $Bug_{10\%}$. This informs our understanding of RQ1: if a design rule is bug-prone, it *persistently* aggregates bug-prone files throughout the life cycle of a project. Therefore, the development team should give higher priority to these design rules, to reduce long term bug rates.

The long-lived ArchRoots *cover a higher portion (on average 3 times) of the* $Bug_{30\%}$ *files compared to a random equal-sized group of files.* Column 5, "Avg.", shows that, the *long-lived ArchRoots* cover from 4 to 82 percent of the top 30 percent most bug-prone files in a project. In particular, there are 14 long-lived ArchRoots with bsc greater than 30 percent. In other words, each root architecturally aggregates more than 1/3 of the $Bug_{30\%}$ files in the project. The standard deviation on column 6 indicates that the c_bsc of these roots remains stable across different releases during the lifetime of a project. Column 7, "Rate", shows the relative bsc of a *long-lived ArchRoot* as compared to that of a random group of files of equal size. With only two exceptions—HBaseConfiguration (from HBase) and COSObjectable (from PDFBox)—all other roots cover higher portions (on average 3 times) of bug-prone files compared to random groups of files. The difference between the long-lived roots and random groups has been confirmed to be statistically significant (p-value $<$ 0.05). The implication is that long-lived *ArchRoots* cover more bug-prone files than average groups of files.

Similarly, the long-lived ArchRoots *contain a higher percentage (on average 3 times) of files from* $Bug_{30\%}$ *as compared to a random group of files.* Column 8, "Avg.", shows that, 8 to 73 percent of the files in each long-lived *ArchRoot* are from $Bug_{30\%}$. In 36 long-lived *ArchRoots*, the dsb reaches 20 percent, meaning at least one in every five files in each root is from $Bug_{30\%}$. Of note, there are five roots with a dsb of at least 50 percent: half the files in these roots are from $Bug_{30\%}$. The dsb *Rate* on column 10 shows that the long-lived *ArchRoots* contain a higher percentage (on average 3 times) of files from $Bug_{30\%}$, as compared to a random group of files of equal size. This means that long-lived *ArchRoots* contain a higher percentage of bug-prone files than average groups of files. Again, the difference between the long-lived roots and random groups has been confirmed to be statistically significant (p-value $<$ 0.05).

In summary, we can answer RQ3. For each project, there exist several (1 to 5) long-lived ArchRoots *that aggregate the top ranked bug-prone files throughout the life of a project. These long-lived* ArchRoots *are usually led by a top ranked bug-prone leading file. And, to no surprise, they are more bug-prone than average groups of files: they cover a higher portion and contain a higher percentage (on average 3 times), of the* $Bug_{30\%}$ *files in a project.* The implication is, once again, that long-lived *ArchRoots* have a persistent impact on the bug-proneness of the projects. Higher bug rates can "grow" out of these roots over time. In particular, roots with bug-prone design rules are likely to keep aggregating bug-prone files over time.

RQ4: Do these ArchRoots *contain architectural flaws that may contribute to their bug-proneness?*

To intuitively understand the root causes that contribute to the bug-proneness of the *ArchRoots*, we qualitatively investigated the (flawed) architectural connections among files in *ArchRoots*. We observed that *ArchRoots* usually contain multiple architectural flaws, such as cyclic dependencies, unhealthy inheritance, unstable interfaces, and modularity violations. These flaws have been verified to have strongly positive correlations with increased bug rates and churn in software projects [26]. The identification of these flaws is automated using a pattern matching approach developed in our prior work [26].

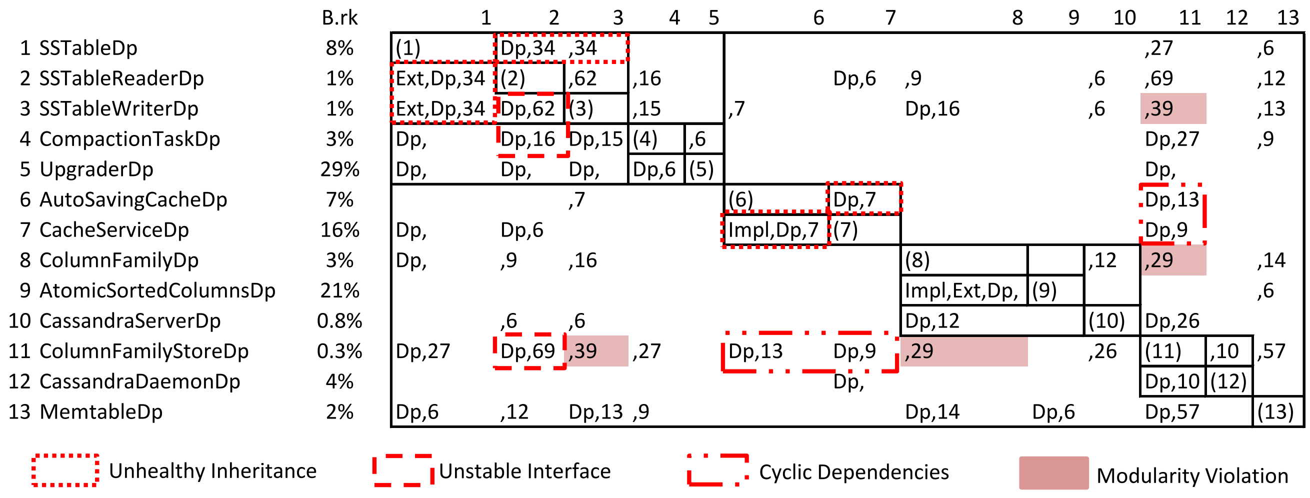
We will use a root identified in Cassandra as an example to show how the *DR*Space modeling, by visualizing structural dependencies and evolutionary coupling simultaneously, can help gain intuitive understanding of how architectural flaws contribute to the high bug rates. In Fig. 7, we highlighted different types of flaws observed in

TABLE 5
Long-Lived ArchRoots for $Bug_{30\%}$

Project	Leading File of Root (#Files in Root)	Root Age (Max Age)	Bug Rank of Leading File	Bug Space Coverage Avg.	Bug Space Coverage Std.	Bug Space Coverage Rate	Design Space Bugginess Avg.	Design Space Bugginess Std.	Design Space Bugginess Rate
Avro	Schema (139)	10 (12)	5%	82%	13%	1.65	16%	4%	1.48
	Protocol (55)	6 (12)	18%	38%	2%	1.88	15%	3%	1.17
	Decoder (44)	6 (12)	-	34%	2%	2.23	22%	3%	1.82
Camel	RouteDefinition (107)	8 (12)	7%	6%	1%	4.56	24%	4%	5.10
	ExchangeHelper (155)	7 (12)	6%	11%	2%	3.72	29%	5%	6.26
	IOHelper (117)	6 (12)	14%	9%	2%	4.30	34%	2%	6.78
	ServiceHelper (159)	6 (12)	27%	12%	5%	4.66	33%	3%	6.61
Cassandra	DatabaseDescriptor (135)	9 (10)	2%	46%	14%	2.17	39%	10%	2.36
	CFMetaData (132)	7 (10)	2%	34%	3%	2.04	47%	6%	2.41
	FBUtilities (155)	7 (10)	7%	46%	9%	1.90	38%	10%	2.11
CXF	NoJSR250Annotations (139)	7 (13)	-	8%	1%	2.63	23%	5%	3.08
	ClassLoaderUtils (129)	6 (13)	-	15%	3%	5.29	43%	2%	5.67
	AbstractPropertiesHolder (176)	9 (13)	-	14%	3%	3.47	32%	5%	5.22
Derby	TableDescriptor (185)	10 (13)	18%	12%	2%	1.73	23%	7%	3.24
	Property (131)	7 (13)	21%	14%	2%	2.60	33%	5%	3.97
	SqlException (119)	13 (13)	13%	12%	2%	2.37	17%	6%	2.62
	Monitor (193)	6 (13)	7%	14%	2%	1.88	18%	12%	2.76
Hadoop	WritableComparable (146)	5 (9)	52%	18%	10%	3.40	8%	2%	2.11
	KerberosName (15)	4 (9)	-	10%	1%	6.19	65%	1%	6.58
	ReflectionUtils (144)	6 (9)	50%	11%	2%	1.31	11%	2%	2.58
	FsPermission (152)	6 (9)	27%	18%	11%	1.67	10%	2%	2.91
HBase	HConstants (164)	4 (9)	16%	57%	23%	1.94	24%	9%	2.30
	ServerName (164)	4 (9)	12%	25%	8%	1.69	43%	4%	1.83
	Filter (86)	4 (9)	18%	9%	2%	1.51	50%	3%	1.98
	HBaseConfiguration (163)	4 (9)	11%	9%	2%	0.69	30%	4%	1.31
Mahout	HadoopUtil (103)	5 (9)	5%	15%	4%	1.76	36%	5%	2.59
	AbstractDistribution (16)	4 (9)	3%	15%	2%	9.92	68%	14%	3.95
	Matrix (142)	6 (9)	14%	16%	1%	1.38	22%	3%	1.72
MINA	IoServiceConfig (57)	4 (8)	-	47%	28%	2.10	25%	7%	2.06
OpenJPA	J2DoPrivHelper (135)	11 (11)	7%	23%	27%	3.61	73%	4%	5.04
	JDBCStore (196)	6 (11)	-	6%	0%	1.19	42%	1%	2.34
	JavaTypes (187)	9 (11)	14%	7%	5%	1.10	50%	7%	3.10
	FetchConfiguration (101)	5 (11)	21%	5%	2%	1.40	49%	7%	3.37
	Value (113)	6 (11)	43%	4%	0%	1.30	40%	1%	2.15
PDFBox	PDDocument (168)	7 (12)	4%	26%	3%	1.07	28%	7%	1.55
	COSArray (163)	7 (12)	6%	64%	5%	1.87	22%	10%	2.14
	COSObjectable (121)	7 (12)	-	19%	4%	0.96	24%	6%	1.32
Pig	PigContext (171)	5 (10)	21%	34%	4%	2.04	27%	3%	2.06
Tika	TikaException (153)	8 (15)	-	59%	13%	1.36	18%	6%	1.46
	MediaType (136)	10 (15)	31%	55%	9%	1.56	22%	5%	1.81
	ContentHandlerDecorator (28)	7 (15)	-	26%	33%	1.36	18%	5%	1.26
Wicket	FormComponent (158)	8 (15)	2%	15%	5%	2.60	24%	6%	2.60
	Session (177)	8 (15)	7%	22%	5%	2.55	18%	3%	2.87
	Strings (172)	7 (15)	11%	41%	6%	4.94	23%	7%	5.56
ZooKeeper	QuorumPeer (47)	9 (10)	7%	29%	3%	1.40	27%	7%	1.80
	KeeperException (75)	5 (10)	42%	29%	12%	1.09	23%	3%	1.59
	ZooDefs (90)	7 (10)	-	42%	19%	1.11	15%	4%	1.21
	t-test p-value				1.05E-10			1.28E-12	

Note: In column "Bug Rank of Leading File", "-" means that the leading file is bug free.
Avg. *stands for the average bsc/dsb of a long-lived root over multiple releases.*
Std. *stands for standard deviation of the bsc/dsb of a long-lived root over multiple releases.*
Rate *is the bsc/dsb of a long-lived root divided by the bsc/dsb of a random space with equal size.*

this space. First, there exists unhealthy inheritance between the parent class *SSTabledp* (row 1) and its child class *SSTableReaderdp* (row 2). *SSTableReaderdp* and *SSTableWriterdp* extend and depend on class *SSTabledp* (cells $c(r2, c1)$ and $c(r3, c1)$). But there is an inverted dependency from *SSTabledp* to its child class *SSTableReaderdp* (cell $c(r1 : c2)$). According to the dependency inversion principle [28], an interface or abstract class should not depend on concrete



		B.rk	1	2	3	4	5	6	7	8	9	10	11	12	13
1	SSTableDp	8%	(1)	Dp,34	,34								,27		,6
2	SSTableReaderDp	1%	Ext,Dp,34	(2)	,62	,16			Dp,6	,9		,6	,69		,12
3	SSTableWriterDp	1%	Ext,Dp,34	Dp,62	(3)	,15		,7		Dp,16		,6	,39		,13
4	CompactionTaskDp	3%	Dp,	Dp,16	Dp,15	(4)	,6						Dp,27		,9
5	UpgraderDp	29%	Dp,	Dp,	Dp,	Dp,6	(5)						Dp,		
6	AutoSavingCacheDp	7%			,7			(6)	Dp,7				Dp,13		
7	CacheServiceDp	16%	Dp,	Dp,6				Impl,Dp,7	(7)				Dp,9		
8	ColumnFamilyDp	3%	Dp,	,9	,16					(8)		,12	,29		,14
9	AtomicSortedColumnsDp	21%								Impl,Ext,Dp,	(9)				,6
10	CassandraServerDp	0.8%		,6	,6					Dp,12		(10)	Dp,26		
11	ColumnFamilyStoreDp	0.3%	Dp,27	Dp,69	,39	,27		Dp,13	Dp,9	,29		,26	(11)	,10	,57
12	CassandraDaemonDp	4%							Dp,				Dp,10	(12)	
13	MemtableDp	2%	Dp,6	,12	Dp,13	,9				Dp,14	Dp,6		Dp,57		(13)

Note:
1. Column “B.rk” shows the bug-proneness ranking of the files in this *ArchRoot*.
2. “Ext” stands for “Extend” and “Impl” stands for “Implement”. “Dp” stands for all other types of static references.

Fig. 7. Architectural flaws in a root in Cassandra.

classes, thus we consider the inverted dependency from the parent to its child as a flawed architectural connection. The co-change data shows that the parent class *SSTabledp* changed together with its child classes *SSTableReaderdp* and *SSTableWriterdp* 34 times in the revision history. We conjuncture that whenever one of the files in this unhealthy inheritance changes, the change will propagate to the other files in the inheritance relation, increasing the likelihood of bugs in these files. As evidenced in Column “B.rk”, *SSTabledp* ranks in the top 8 percent most bug-prone, and the two child classes rank in the top 1 percent most bug-prone among all project files.

We also observed an unstable interface in this root. There are numerous files that structurally depend on *SSTableReaderdp* (as shown by the cells on the column with label “1”). Therefore, *SSTableReaderdp* should be kept as stable as possible, otherwise, changes to it will potentially affect all the files depending on it. In fact, the history co-change data indicates that *SSTableReaderdp* changed together with three dependents: *SSWriterdp*, *CompactionTaskdp*, and *ColumnFamilyStoredp*, 62 (cell[r3,c2]), 16 (cell[r4,c2]), and 69 (cell[r11, c2]) times respectively. We posit that whenever *SSTableReaderdp* changes, it could propagate changes to files that structurally depend on it. As a result, these four files suffer from high bug rates (all rank in the top 3 percent most bug-prone) as shown in column “B.rk”.

There are also cyclic dependencies between *ColumnFamilyStoredp* (row 11) and two files: *AutoSavingCachedp* (row 6) and *CacheServicedp* (row 7). As shown in cells $c(r6, c11)$ and $c(c11, r6)$, *ColumnFamilyStoredp* and *AutoSavingCachedp* form a structural dependency cycle with each other. Similarly for *ColumnFamilyStoredp* and *CacheServicedp* (cells $c(r7, c11)$ and $c(r11, c7)$). Whenever, one file in a cycle changes, it has a strong likelihood to propagate changes to other files in the cycle, thus increasing change effort. The history co-change data indicates that *ColumnFamilyStoredp* changed together with *AutoSavingCachedp* and *CacheServicedp* 13 and 9 times respectively.

Last but not least, there are modularity violations among *ColumnFamilyStoredp* (row 11), *SSTableWriterdp* (row 3), and *ColumnFamilydp* (row 8). The concept of modularity violation was first proposed by Wong et al. [23] as the phenomenon where a set of files frequently change together in the revision history but lack any structural dependencies. In this case, *ColumnFamilyStoredp* has no structural dependencies with *SSTableWriterdp* or *ColumnFamilydp*, but it changes with them 39 (cell[r4, c11] and cell[r11, c4]) and 29 (cell[r8, c11] and cell[r11, c8]) times respectively. In a prior case study, Schwanke et al. [18] found that modularity violations usually imply shared concepts between files that would benefit from a better encapsulation design. Whenever the shared concept changes in one file, the other files have to accommodate the change. Unless the concept is encapsulated, files sharing the concept tend to change together frequently, causing bug rates to increase.

Given that this root contains numerous architectural flaws, it is not surprising to see from column “B.rk” that files in this root all rank above the top 30 percent most bug-prone. Actually, except for three files, *Upgraderdp*, *CacheServicedp*, and *AtomicSortedColumnsdp*, the files in this root rank in the top 10 percent most bug-prone.

To answer RQ4, in each ArchRoot, *we have observed recurring architectural flaws, such as cyclic dependencies, unhealthy inheritance, unstable interfaces, and modularity violations. Our analysis suggests that these flaws could be the root causes of high bug rates because they can propagate changes among files, making bugs hard to eradicate.* The take-away message from RQ4 is that it is hard for a developer to make a single file bug-free without fixing the architectural flaws that connects it to other files. Thus, when fixing bugs involving a set of files with a dependency cycle, the developers should first cut the cycle to prevent the changes from propagating. To fundamentally reduce the bug rates in the long run, the developer team should consider fixing these flaws by refactoring. Otherwise, these flaws are likely to keep incurring high bug-fixing costs over time.

5.4 Approach Complexity

The above research questions have shown the usefulness of the *ArchRoot* detection approach for investigating how bug-prone files are significantly and consistently aggregated in

TABLE 6
DRSpace Modeling and Root Detection—Execution Time

Subject	Total # Files	DRSpace Set Generation	Top 5 ArchRoot Detection
ZooKeeper-3.4.5	382	0.95 s	0.02 s
Avro-1.7.6	426	1.4 s	0.01 s
MINA-1.1.7	550	2.7 s	0.02 s
Tika-1.7	550	1.9 s	0.01 s
PDFBox-1.8.7	791	6 s	0.04 s
Pig-0.9.2	1,195	20 s	0.08 s
Mahout-0.9	1,262	22 s	0.04 s
Cassandra-2.1.2	1,337	33 s	0.09 s
HBase-0.98.2	2,055	2 m	0.15 s
Derby-10.11.1	2,776	4.6 m	0.3 s
Wicket-6.19.0	3,081	5 m	0.26 s
OpenJPA-2.2.2	4,314	13.8 m	0.28 s
Hadoop-2.5.0	5,488	42 m	0.56 s
CXF-3.0.0	5,509	32 m	0.5 s
Camel-2.12.4	9,866	3.7 h	0.7 s

architectural connected groups, and for revealing the potential architectural flaws among the bug-prone files.

For any approach to be actually useful in practice, it has to have acceptable computation costs. In this section, we will discuss the complexity and computational costs of our approach. Since the bug space mining in step 2 is straight forward it can be accomplished in O(n) time (where n is the number of files in a project). We will focus on the complexity and running time of Algorithms 1 and 2. As we will show, these algorithms are all computable in affordable time on large-scale projects. The evaluation experiments are all conducted on a Windows 7 machine, Intel(R) Core(TM) i7-6700HQ CPU @ 2.60 GHz.

Complexity of Algorithm 1: the time complexity of the DRH algorithm [21], [25] on line 2 is $O(n^3)$ for a project of n source files. And the two nested for loops between line 3 and line 13 have complexity $O(n^2)$. Therefore, the overall complexity of Algorithm 1 is $O(n^3)$. As shown in Table 6, The generation of the comprehensive *DRSpace* set for most projects finishes in a few minutes. However, as the scale of the project increases, the processing time in our experiments reached several hours (e.g., 3.7 hours for Camel-2.12.4 with 9,866 files). While the comprehensive *DRSpaces* generation for large-scale systems, such as Camel, is clearly not a real-time task, this calculation is only necessary for new releases when the architecture of the system updates dramatically.

Complexity of Algorithm 2: the complexity of this algorithm is $O(n^2)$ for a project of n source files. Large scale projects usually can be processed in a few seconds. For example, the execution time for Camel was 0.7 seconds in our experiments. In general, as is listed in Table 6, once the *DRSpaces* are generated for a given release, the detection of the top *ArchRoots* during the development process can be done in less than a second, even when the bug-proneness of source files changed over time.

Therefore, our approach is applicable to large-scale industry level projects with affordable computation times on modest computing hardware.

6 Industry Impacts

In this paper, we presented the results of our case study on open source projects. The proposed *DRSpace* modeling and *ArchRoot* detection approach have also shown great potential in systematically uncovering architectural root causes for bug-proneness and high maintenance consequences in commercial software projects.

In a recent case study on a large-scale industry project [29], we identified the six most bug-prone *ArchRoots* and reported our findings to the developer team. The architects verified that these six groups of files were truly the architectural root causes of maintenance difficulties in their project. Based on the analysis results, we suggested refactoring plans to fix the architectural root causes of bug-proneness for the long-term maintainability. These plans were justified by an economic model estimating both the costs and the benefits (the return-on-investment) of the refactoring. These refactoring proposals were accepted for implementation by the case company.

Other unpublished industry case studies involving a broader variety of software projects have also consistently shown the potential of our approach in identifying architectural root causes of bug-proneness and maintenance difficulties. We believe, based on our observations, that the identified *ArchRoots* in a software project are potential refactoring opportunities to achieve long-term maintainability.

7 Related Work

In this section, we will discuss research related to our study, including defect prediction, architecture views, and technical debt.

Defect Prediction Software defects consume a considerable amount of effort to discover, test and fix. To improve the efficiency of such effort, researchers have built prediction models to attempt to locate defects in software product releases. Machine learning based techniques, such as multilinear regression models and multivariate modules, artificial neural networks, decision trees, and rule inductions, have been used in such effort [30]. The data used as predictors falls into three categories:

1) Predicting defects using code complexity metrics. Ohlsson et al. [31] collected various complexity metrics, such as McCabe's Cyclomatic complexity, fan-in and fan-out etc., to build their prediction model. Nagappan et al. [32] investigated eighteen complexity metrics as defect predictors in a case study of five Microsoft software components. Their studies showed that different complexity metrics worked best for different projects. Lessamann et al. [33] used 39 complexity-based metrics and applied 22 classifiers to improve the defect prediction. Zheng developed three cost-sensitive algorithms to boost neural networks for software defection based on 11 code complexity metrics to mitigate the high cost of false positive. However, Fenton et al. [34] pointed out that it is problematic to use size and complexity metrics to predict defects. Despite the strong correlation between high complexity and defects, it is flawed to conclude causality from correlation. Furthermore, prediction modules based on complexity metrics do not offer any coherent explanation of the potential root causes of defects. Similarly, Challagulla et al. [30] also found that size and complexity metrics are not sufficient for accurately predicting real-time

software defects. In comparision, Menzies et al. [35] suggests that static code measures are useful and easy to use, especially when quality assurance budge is limited. Menzies et al. [36] summarized from extensive prior work that *response for class* (# methods executed by arriving messages) received the highest consensus in defect prediction; while *lack of cohesion* (pairs of methods referencing one instance variable) has the least consensus.

2) Predicting defects using software history. Moser et al. [37] found that metrics based on software history are more efficient defect predictors than code complexity metrics in Eclipse project. Graves et al. [11] proposed a weighted time damp model, which computes the fault potential of modules in a software system, based on the changes made to each module in its revision history. Ostrand et al. [12] built a negative binomial regression model using revision information from previous releases, such as file age and file change, to predict the number of faults for each file in a large industrial inventory system. Nagappan et al. [13] showed that, in a case study of a large industry software system, code churn (number of lines of code modified) could be an excellent predictor for defect density. Kim et al. [14] built a model, called FixCache, using cached bug information, to predict future bug location with high accuracy. More recently, Martinez et al. [15] learned from past bug-fixing transactions to automate bug fixing of future bugs. Musco et al. [16] managed to use past change impacts to predict future change impacts. In addition, Alencar et al. [17] systematically investigated the bug introducing changes and the future impact of such changes.

3) Predicting defects using both change history and complexity metrics. Koru and Liu [38] found that code metrics and defect history data collected at class level can be used to build machine-learning models that preidct top defect classes in practice. Ostrand et al. [39] successfully predicted 80 percent of the faults using file size and file change information for two large industrial systems. Gao et al. used a hybrid set of metrics in a search algorithm which outperforms other approaches. And when 85 percent of the selected metrics are dropped, the prediction model either improved or remained the same [40]. Jing et al. [41] recently used dozens of code and history metrics to predict defects within/cross projects, specifically addressing the interferences that projects usually contain imbalanced defect-free (majority) and defective (minor) classes. Zhang et al. [42] investigated the effectiveness of using different summation schemes to aggregate different combinations of metrics to predict defects.

Despite the significant research effort, defect prediction is still challenging in practice. First, one of the challenges comes from the noise in the history data, such as missing or misleading key words linking to bug fixes [43], and imbalance between defect data versus non-defect data [44]. Second, the challenges also comes from the generosity of prediction models [45], [46], [47]. For example, Zimmermann et al. [45] reported that prediction model built for one project cannot be applied to accurately predict defects in another project, even in a comparable problem domain and use a similar development process. Similarly, Turhan et al. [47] also reported that defect predictors learned from within-company outperform the ones learned from cross-company data. They suggest that companies without local defect data to initiate the defect prediction by collecting cross-company data while they collect local defect data. Rahman et al. [48] found that cross-project prediction could be as good as within-project prediction when applying a cost-sensitive approach, and it is much better than random prediction. Third, almost all bug prediction work relises on IR-based measurements for evaluating the effectiveness. Precision, recall, and the ROC curve. Mende and Koschke pointed out that these measurements don't consider the cost of quality assurance on different software modules and thus are not reasonable in practice [49]. Researchers have tried to overcome these challenges in different ways. D'Ambros et al. [50] contributed a benchmark for defect prediction, with a publicly available dataset and a extensive comparison of well-known bug prediction approaches. Jureczko and Madeyski [51] performed clustering on 92 versions of 38 proprietary open-source and academic projects. Projects with similar characteristics are grouped together from the defection prediction point of view. Mende and Koschke [49] suggest performance measures of prediction models should consider cost metrics for quality assurance to be realistic.

The goal of defect prediction work is to prioritize the quality assurance, testing, and debugging tasks. However, Tantithamthavorn et al. [52] recently compared the bias and variances of 12 defect prediction models and found that existing prediction models are likely to be biased and generate unstable results.

More importantly, what has not been explored as fully are the root causes that contribute to bug-proneness, and how to reduce the over-all bug-proneness more fundamentally in software projects. Leszak et al. [53] investigated the root causes of bug-proneness by manually categorizing modification requests in a bug-tracking database. They documented five types of root causes: component specification and design documentation, component implementation, system and domain knowledge, document and code reviews, and project management. However the contribution of a system's modular structure on bug-proneness has not been systematically addressed in the research literature.

We clarify that the goal of this work is *not* to predict defects in a software system. Instead, our objective to reveal the underlying architectural connections among files that connect bug-prone files that could be the root causes of bug-proneness and high bug-fixing rates. We choose to use *dsb* and *bsc* to evaluate the bug-proneness level of a *DRSpace* (defined in Section 4.2) to emphasize our unique goal, although they are mathematically equivalent to the IR-based precision and recall measurement used in bug prediction work. The most important implication of this study is that to fundamentally reduce the bug-proneness and increase the quality of a software project, the developers should pay attention to the architectural connections among bug-prone files, especially the flawed connections that violate well-accepted design principles. A potential solution is to perform refactoring to fundamentally remove the architectural flaws and improve software quality, as supported by Ratzinger et al. [54]: the number of software defects decreases, if the number of refactorings increased in the preceding time period.

Architecture Description Languages. To communicate software architecture for addressing different concerns, stakeholders require a language that can be understood and documented. The work in Architecture Description Languages (ADLs) provides tools for parsing, displaying, analyzing, or simulating architectural descriptions written in their associated languages. An ADL could be any form of formal representation to describe software architecture. It could be in either graphic or syntax representation. It describes software entities, such as processes, threads, data, and their interactions. Garlan [55] summarized from related literature that ADLs provide notations and concrete syntax for modeling software architecture as components, connectors, and events.

There has been various types of ADLs. Each has different focus. For example, Terry et al. [56] contributed a suit of supporting tools to help specify, design, validate, package, and deploy distributed intelligent control and management (DICAM) applications, in the domain of vehicle management systems. Newton et al. [57] developed a graphic parallel programming system that models parallel architectures. Palsberg et al. [58] implemented a Demeter system that can be used to design and automatically generate adaptive programs specified by a so-called propagation pattern. Jahanian et al. [59] proposed a specification language for real-time systems called Modechart. Edwards et al. [60] proposed RESOLVE to describe a conceptual module as a RESEOLVE unit that specified an abstract component by defining its context and its interface structure and behavior for designing reusable components. Shaw et al. [61] sketched and implemented a model called UniCon for defining architectures as different types of components and different ways these components can interact. Allen et al. [62] proposed a model called Wright to distinguish between ″implementation″ and ″interaction″ relationships between modules. To help architecture understanding, Aldrich et al. [63] contributed an extension to Java, called *ArchJava*, that aimed at keeping architecture and implementation consistent.

Clements et al. [64] reviewed and compared the pros and cons of different types of ADLs. They claimed that a glaring commonality among different ADLs was the lack of in-depth experience and real-world application. In addition, they claimed that none of the ADLS could capture the design rationale and/or evolution history of architecture.

In comparison, this study aims at providing a general architecture model and systematic analysis techniques to discover poor architectural decisions that are responsible for quality problems related to bug-proneness. Our work is different from the work in ADL in the following aspects: 1) ADLs are mainly used in design, while our work is to retrospectively analyze and monitor the evolution of software architecture for addressing concerns related to maintenance quality; and 2) none of the ADLs captures the evolution history of a system's architecture, while in our DRSpace modeling, we express evolutionary coupling among source files as a special form of architectural connection to support the diagnosing of architectural flaws; (3) ADLs were not designed to directly support the analysis of a software system's architecture, while the ArchRoot and ArchDebt approach in this dissertation can automatically identify the architectural root causes of bug-proneness, and suggest refactoring opportunities.

Architecture Views and Reverse-Engineering. A software system may be analyzed using various architecture views ([65] [66]). However, the ground truth of software architecture is usually difficult to acquire. This is because there is almost never up-to-date and accurate architectural documentation [67]. In the past decades, a considerable amount of literature have been contributed to recovering the "as-built" software architecture views from project code base and other artifacts. These work provide a variety of approaches for recovering different architecture viewpoints of separate foci and goals.

A main thread of recovering techniques focus on clustering software elements based on different criteria. To name a few, Praditwong et al. [4] proposed a search-based approach to cluster software objects into modules based on high cohesion and low coupling. Corazza et al. [5] investigated the effectiveness of using lexical information, e.g., in comments, identifier and method names, to perform software system clustering. Erdemir et al. [6] proposed a software clustering approach based on the community structure: as object-oriented software directly represent the real life object, thus it displays the behavior and community structure. Naseem [7] analyzed the effectiveness of clustering approach based on various similarity measures between software entities. Kobayashi et al. [8] proposed a static-dependency-based clustering approach that gathers software modules from a feature viewpoint. Misra et al. [9] presented a software clustering approach to group relevant code elements into components based on multiple types of code features.

The above techniques recover architecture views based on different rationale, thus can generate drastically different high-level models for the same system. They mostly share one common feature: each software element in a system can usually only be assigned to one cluster. In reality, however, it is common for one artifact (i.g., source file) to participate in more than one design space. We propose DRSpace modeling to reflect the fact that design spaces, such as features and patterns, may overlap with each other.

Another thread of work focuses on recovering the feature models—features and their relationship are captured as graphs—and analyzing their variance over software product lines. She et al. and Acher et al. [68], [69] reverse-engineer the feature model of software systems to identify architecture variation points and the explicit constrains between features. Tizzei et al. [70] focus on modeling cross-cutting features in the architecture models. Most recently, Shatnawi et al. [71] proposed an approach to reverse-engineer the architecture elements variances for Software Product Line Engineering.

Architecture recovering work has been conducted in different, specific domains, such as object-oriented and service-oriented architecture. Kebir et al. [72] focus on identifying software components in object-oriented systems, combining hierarchical clustering and genetic algorithm. They [73] also evaluate a fitness function to measure the quality of a component for reuse. Weinreich et al. [74] proposed an approach for automatically extracting architectural information from "as-built" service-oriented architecture in the banking domain. More recently, Alshara et al. [75] proposed an approach to map recovered object-oriented software modules into component-based models.

One of the purposes of reverse-engineering an architecture is to support quality analysis. The majority of architecture analysis methods created to date have either focused on questionnaires [76] or scenarios [77], [78]. These approaches are, however, labor-intensive and their success depends heavily on the skill of the analysts. More recently, architecture recovering has been toward to different goals. Tekinerdogan et al. [79] proposed a solution to define architecture viewpoints to reflect two quality concerns: recoverability and adaptability. They applied their approach on two the open source media player application, MPLayer. Boussaidi et al. [80] presented an approach to re-construct and document the software architecture of legacy systems, targeting at two types of views 1) a layered view and 2) a feature-based view. Garcia et al. [81], [82] contributed a framework to recover the "ground-truth" architecture, and applied on four open source projects. The authors [83] further compared six state-of-the-art recovering approaches, e.g., [84], [85], [86], to evaluate the accuracy toward to the "ground-truth". Detten et al. [87] proposed an semi-automated approach, called "Archimetrix" to recover software architecture and detect/remove design deficiency in the meantime.

To the best of our knowledge, no existing methodologies can efficiently and automatically identify the architectural flaws that contribute to software quality problems, such as high bug-proneness.

Technical Debt Technical debt (TD) [88] is a metaphor used to describe the consequences of short-sighted code changes that may have the cumulative effect of compromising long-term system goals. For example, possible symptoms of technical debts include (but are not limited to) missing documentation, growing project to-do list, minor changes requiring disproportionately large effort, code cannot be understood, architectural inconsistency [89].

Research related to TD has drawn significant attention in the past decades [90], [91], [92], [93], [94], [95], [96], [97], [98], [99], [100], [101], [102], [103], [104], [105], [106], [107]. For example, Li et al. [108] identified 94 studies, covering 10 types of technical debt, involving 29 tools for technical debt management between 1992 and 2013. Tom et al. [90] contributed a theoretical framework that provides a holistic view of technical debt comprising a set of technical debts dimensions, attributes, precedents, and outcomes, and a taxonomy that describes different forms of the technical debt phenomenon. Technical debt can be categorized to code debt, design and architectural debt, environmental debt, knowledge distribution and documentation debt, and testing debt.

To locate TD in source code, a number of heuristics have been suggested, which attempt to identify code anomalies, such as code clones, long methods, and god classes. These problems can be detected by code analysis tools such as SonarQube.[22] Similarly, Marinescu [94] identified design flaws, such as god class, class with low cohesion, data class, code clone, as proxies of technical debt. Zazworka et al. [109] compared four different TD detection approaches in terms of their correlation with change- and bug-proneness. They found that only a subset of TD detected by the four approaches were associated with software change- and bug-proneness. The implication is that not all code anomalies are certain to cause maintenance consequences or quality problems, and, not all quality problems are associated with code anomalies. For example, a stabilized code clone will typically not lead to any quality problems. Kruchten et al. [91] argue that once tools are developed to assist technical debt identification, there is a danger of equating it with whatever the tools can detect. Large amounts of potential technical debts, such as structural or architectural debt, will be left undetectable. While, Li et al. suggest to have dedicated tools for managing various types of TD [108].

Technical debt entails a serials of balancing acts for practitioners: short-term benefits versus long-term pain, software quality versus business reality, customers' expectations versus their needs and wants [99]. The management of technical debt ultimately requires decision making in terms of paying off or deferring technical debt [100]. Nugroho et al. [101] contributed an approach to quantify debts to support such decision making based on an empirical data of 44 systems. Their approach models the cost to fix technical quality issues, and interest, extra cost spent on maintenance due to technical quality issues. Their approach helps practitioners to gain important insights on return of investment in software quality improvement. Guo et al. [102] contributed a portfolio approach, borrowed from the finance domain, to determine the optimal decision for managing technical debt. The portfolio management is a decision making process similar to the process of technical debt management, where practitioners make decisions on when and what technical debt items should be paid or kept. Guo et al. [103] found that decision made without careful analysis could aggravate the negative effect of technical debt, while attention and explicit management of technical debt could make a difference. Letouzey [104] presented the software quality assessment based on lifecycle expectations method to understand the impacts of technical debt.

There are still a set of open research questions in investigating technical debt. For instance, refactoring opportunities that identified through problematic design, e.g., bad code smell, detection are difficult to evaluate the outcome of refactoring suggestions, even if the suggestions were accepted and implemented [105]. Similarly, Curties et al. [106] suggest that technical debt emerges from poor structural quality. When significant architectural change is needed, small, local refactoring cannot compensate for the lack of a coherent system-wide architecture [105]. Nord et al. [107]'s study suggests that the short-term value and the long-term cost to be incurred by architecture and architecture-related technical debt must be taken into account in decision-making related to delivering a product. Thus making the architectural debt visible provides necessary information for such decision-making to be informed.

The architectural analysis approach contributed in this study has the potential to provide a coherent system-wide architecture for guiding non-trivial refactoring opportunities. And the *ArchRoots*, as refactoring suggestions, have visible goal of reducing bug-proneness of a software system as the goals of the refactoring outcome. Our work is the first, to our knowledge, to investigate the (flawed) architecture connections among files that have long-term consequences

22. http://www.sonarqube.org/

on the over-all bug-proneness of a project. We have used a DRSpace-based analysis to help quantify the costs of the technical debt incurred by architecture problems [29], [110].

8 Limitations and Threats to Validity

In this section, we first discuss the limitations and then talk about threats to validity of DRSpace modeling and ArchRoot analysis.

Limitations: First, the 15 open source projects studied in this paper are all implemented in Java, and the industrial projects studied earlier are implemented in either Java or C++. Thus, we cannot claim that our approach can work as effectively for projects implemented in other programming languages, particularly non-object-oriented languages. The concept of *design rule* is naturally embraced by object-oriented programming languages, such as Java and C++, in the form of the abstraction mechanisms that they provide. How well the *DRSpace* model can capture the modular structure of projects implemented in other languages has not been tested. To overcome this limitation, in our future work, we plan to apply our approach to projects implemented in other programming languages, such as C and Python.

Second, like most reverse-engineering research, our DRSpace modeling and analysis approach is conducted in retrospective. The *ArchRoots* are identified based on reverse-engineering the "as-built" code base and the bug spaces are extracted from the already conducted changes. Our approach provides means to help software practitioners reveal and understand the flawed architectural connections that are potentially harmful for the project quality. However, in this paper, we didn't conduct any investigation on the predicting power of the approach, for example, how flawed architecture connections among files are likely to "grow" or which *DRSpaces* are likely to remain bug-prone in the future has not been investigated.

Third, the investigation of long-lived *ArchRoots* requires ample revision history. The 15 open source projects we studied have revision histories covering four to eight years. The adequacy of revision history allowed the investigation of the evolution of *ArchRoots* over time. The analysis of long-lived is limited to software projects with substantially shorter revision histories.

Last, the evolutionary coupling captured in our *DRSpace* modeling is calculated as the number of times two files were committed together as recorded in the revision history, without considering other factors such as time differences, or contributing developers etc. [111], [112]. This definition may not be sufficient to capture all implicit dependencies. There could be different ways to improve the representation of co-changes, such as dynamic binding [113], run-time data-flow [114], and shared code ownership [115]. The DRSpace modeling itself, however, is independent of how co-changes are defined. Different types of co-changes can be added as new types of relations in a DRSpace. Exploring these options is our future work.

Threats to Validity: Like other studies that rely on mining bug-fixing changes from project repositories, the accuracy of our *ArchRoot* detection algorithm is impacted by the quality of available data, i.e., how well developers classify and tag bug-fixing changes. As discovered by prior research [116], [117], [118], there are two challenges in mining bug fixes from software revision history: 1) there is often no explicit link between the version control system and the bug tracking system to pinpoint bug fixing changes; 2) a bug tracking system not only contains bug fixing activities, but also records activities like adding new features, refactoring, code cleanup, and maintaining documents. In our study, the input bug space to the *ArchRoot* detection algorithm is mined from bug tracking and version control systems. We use the pattern matching method proposed by Sliwerski et al. [116]. A change in a version control system is recognized as a bug fix if the developers mentioned a bug ticket ID in the change message. To weaken the interference of noise in data result from the two challenges, we studied different bug-proneness levels: $\text{Bug}_{30\%}$ and $\text{Bug}_{10\%}$. We believe that it is possible that a file might be misclassified as a bug file once or twice, but as the bug-proneness level increases to $\text{Bug}_{30\%}$ and $\text{Bug}_{10\%}$, such files are unlikely to be mis-classifications. With that being said, we admit that this still poses an external threat to validity in identifying the architectural root causes of error-proneness. Project insiders may have better knowledge to fine tune the bug-prone level of the input bug space to focus on the bug-prone files of their interests, and our tool will allow the users to enter the bug-proneness threshold as a parameter.

Second, we acknowledge the possible bias in the analysis of flawed architectural connection patterns. In this paper, we mainly focused on four hot-spot flawed patterns when investigating the connections in *ArchRoots*: modularity violations, unstable interface, cyclic dependencies, unhealthy inheritance. We acknowledge the potential bias in two perspectives: first, we can't guarantee that these patterns comprehensively cover all possible harmful patterns. Second, the "flawed" patterns sometimes are unavoidable under certain circumstances and thus are not truly problematic. For example, a cyclic dependency is an intrinsic part of the visitor pattern due to employing double-dispatch for binding the visitor and the element being visited: the visitor contains a method to "visit" the element while the element contains a method to "accept" the visitor. In this paper, we did not specifically distinguish such cases or give them special treatment due to limited knowledge regarding the studied subjects (concrete scenarios, applied design patterns, etc.). Therefore, we acknowledge this as an internal threat to validity and leave the deeper investigation of this issue to future work.

9 Conclusion

In this paper, we first proposed *DRSpace*, a new software architecture model that can be used to analyze implemented architecture in source code. DRSpace modeling captures multiple, overlapping design spaces in a complex system, so that distinct aspects of a system, such as a feature or a pattern, can be viewed and analyzed separately.

As the first exploration of using DRSpace to reveal the architectural impact on software quality, we proposed the concept of *ArchRoots*—the DRSpaces that attract and propagate errors to multiple files, and their detection algorithm. Our research is unique in that it explicitly ties software quality—in terms of bug-proneness—to the modular structure of software architecture and architecture flaws.

From the 15 projects that we selected, with varying sizes, domains, and ages, we observed that 1) an bug-prone design rule can make a large number of files within it DRSpace also error-prone; 2) the impacts of architecture connections among bug-prone files are *significant* and *persistent* over time. In each project, a substantial percentage of bug-prone files, with different bug-proneness levels and in different time-spans, are *constantly* concentrated in the top five *ArchRoots*. Our study also indicates that the impacts of architecture connections among bug-prone files are persistent: there are long-lived architecture roots appearing in multiple releases of a project, connecting bug-prone files fixed in different time intervals.

This suggests that, when fixing errors in a file, changes may propagate to other files that architecturally connect to it, and this impact can be persistent, lasting from months to years. The quantitative analysis of the architecture connections among bug-prone files shows that the architecture problems contained in the *ArchRoots* are likely to be the root causes of bug-proneness.

The take-away message of our study is this: to increase the quality and reduce the number of bugs within a software system, practitioners should pay attention to high-impact bug-prone *design rules*, as well as the architecture roots aggregating bug-prone files. Our study also justified the value of further investigation of concrete architecture problems: how they contribute to file bug-proneness, and their potential solutions, that is, the possible ways of refactoring.

ACKNOWLEDGMENTS

This paper is an extension to our work "Design Rule Spaces: A New Form of Architecture Insight"' published in the 36th International Conference on Software Engineering in 2014. We have extended our work in the following aspects. First, extending the simple tool support described in our conference paper, we have developed and presented in this paper an Architecture Root Detection algorithm to *automatically* identify the top bug-prone design spaces. Second, in our ICSE publication, we only studied three open source projects; in this work, we have further studied and reported results from 15 open source projects with diverse characteristics. Third, in the conference paper, we only studied and reported one release of each project, while, in this work, we have investigated multiple (8 to 15) stable releases in multiple (up to 9) years of each project to investigate the persistence of architecture impacts on software quality. Fourth, in our conference paper, we used the number of bugs to approximate bug-proneness; in this paper, we rank bug-proneness in term of bug-fixing frequency and use percentiles to measure bug-proneness. Fifth, we normalized bug rates by LOC in our analysis, to ensure that the results that we reported—high bug rates—were not merely due to finding large files. Finally, we have added a discussion of architectural design flaws and their impact on bug-proneness in a software system.

REFERENCES

[1] C. Y. Baldwin and K. B. Clark, *Design Rules, Vol. 1: The Power of Modularity*. Cambridge, MA, USA: MIT Press, 2000.
[2] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2003.
[3] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt," in *Proc. 10th Joint Meet. Found. Softw. Eng.*, 2015, pp. 50–60. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786848
[4] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Trans. Softw. Eng.*, vol. 37, no. 2, pp. 264–282, Mar. 2011. [Online]. Available: http://dx.doi.org/10.1109/TSE.2010.26
[5] A. Corazza, S. D. Martino, V. Maggio, and G. Scanniello, "Investigating the use of lexical information for software system clustering," in *Proc. 15th Eur. Conf. Softw. Maintenance Reengineering*, Mar. 2011, pp. 35–44.
[6] U. Erdemir, U. Tekin, and F. Buzluca, "Object oriented software clustering based on community structure," in *Proc. 18th Asia-Pacific Softw. Eng. Conf.*, Dec. 2011, pp. 315–321.
[7] R. Naseem, O. Maqbool, and S. Muhammad, "Improved similarity measures for software clustering," in *Proc. 15th Eur. Conf. Softw. Maintenance Reengineering*, Mar. 2011, pp. 45–54.
[8] K. Kobayashi, M. Kamimura, K. Kato, K. Yano, and A. Matsuo, "Feature-gathering dependency-based software clustering using dedication and modularity," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2012, pp. 462–471. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2012.6405308
[9] J. Misra, K. M. Annervaz, V. Kaulgud, S. Sengupta, and G. Titus, "Software clustering: Unifying syntactic and semantic features," in *Proc. 19th Work. Conf. Reverse Eng.*, Oct. 2012, pp. 113–122.
[10] R. Kazman and S. J. Carriere, "Playing detective: Reconstructing software architecture from available evidence," *Autom. Softw. Eng.*, vol. 6, no. 2, pp. 107–138, Apr. 1999.
[11] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, pp. 653–661, Jul. 2000.
[12] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *Proc. 13th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, Jul. 2004, pp. 86–96.
[13] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. 27th Int. Conf. Softw. Eng.*, 2005, pp. 284–292.
[14] S. Kim, T. Zimmermann, J. Whitehead, and A. Zeller, "Predicting faults from cached history," in *Proc. 29st Int. Conf. Softw. Eng.*, May 2007, pp. 489–498.
[15] M. Martinez and M. Monperrus, "Mining software repair models for reasoning on the search space of automated program fixing," *Empirical Softw. Eng.*, vol. 20, no. 1, pp. 176–205, Feb. 2015. [Online]. Available: http://dx.doi.org/10.1007/s10664-013-9282-8
[16] V. Musco, A. Carette, M. Monperrus, and P. Preux, "A learning algorithm for change impact prediction," in *Proc. IEEE/ACM 5th Int. Workshop Realizing Artif. Intell. Synergies Softw. Eng.*, May 2016, pp. 8–14.
[17] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. Hassan, "A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes," *IEEE Trans. Softw. Eng.*, vol. 43, no. 7, pp. 641–657, Jul. 2017.
[18] R. Schwanke, L. Xiao, and Y. Cai, "Measuring architecture quality by structure plus history analysis," in *Proc. 35rd Int. Conf. Softw. Eng.*, May 2013, pp. 891–900.
[19] S. Huynh, Y. Cai, and K. Sethi, "Design rule hierarchy and analytical decision model transformation," Drexel University, Philadelphia, PA, USA, Tech. Rep. DU-CS-08-04, Nov. 2008, https://www.cs.drexel.edu/node/13664.
[20] S. Huynh, Y. Cai, and K. Sethi, "Design rule hierarchy and model transformations," *Proc. Presented Student Res. Forum 16th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2008.
[21] Y. Cai, H. Wong, S. Wong, and L. Wang, "Leveraging design rules to improve software architecture recovery," in *Proc. 9th Int. ACM Sigsoft Conf. Quality Softw. Archit.*, Jun. 2013, pp. 133–142.
[22] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proc. 14th IEEE Int. Conf. Softw. Maintenance*, Nov. 1998, pp. 190–197.
[23] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting software modularity violations," in *Proc. 33rd Int. Conf. Softw. Eng.*, May 2011, pp. 411–420.
[24] G. G. Chowdhury, *Introduction to Modern Information Retrieval*. London, U.K.: Facet publishing, 2010.
[25] 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 Int. Conf. Autom. Softw. Eng.*, Nov. 2009, pp. 197–208.
[26] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in *Proc. 15th Work. IEEE/IFIP Int. Conf. Softw. Archit.*, May 2015, pp. 51–60.

[27] Q. Feng, R. Kazman, Y. Cai, R. Mo, and L. Xiao, "An architecture-centric approach to security analysis," in *Proc. 16th Work. IEEE/IFIP Int. Conf. Softw. Archit.*, May 2016, pp. 221–230.

[28] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

[29] R. Kazman, et al., "A case study in locating the architectural roots of technical debt," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 179–188.

[30] V. U. Challagulla, F. B. Bastani, I.-L. Yen, and R. A. Paul, "Empirical assessment of machine learning based software defect prediction techniques," in *Proc. 10th IEEE Int. Workshop Object-Oriented Real-Time Dependable Syst.*, 2005, pp. 263–270.

[31] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Trans. Softw. Eng.*, vol. 22, no. 12, pp. 886–894, Dec. 1996.

[32] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 452–461.

[33] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, Apr. 2008.

[34] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 25, no. 5, pp. 675–689, Nov./Dec. 1999.

[35] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: Current results, limitations, new approaches," *Autom. Softw. Eng.*, vol. 17, no. 4, pp. 375–407, 2010.

[36] T. Menzies, et al., "Local versus global lessons for defect prediction and effort estimation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 822–834, Jun. 2013.

[37] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 181–190.

[38] A. G. Koru and H. Liu, "Building effective defect-prediction models in practice," *IEEE Softw.*, vol. 22, no. 6, pp. 23–29, Nov./Dec. 2005.

[39] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 340–355, Apr. 2005.

[40] K. Gao, T. M. Khoshgoftaar, H. Wang, and N. Seliya, "Choosing software metrics for defect prediction: An investigation on feature selection techniques," *Softw.: Practice Experience*, vol. 41, no. 5, pp. 579–606, 2011.

[41] X. Y. Jing, F. Wu, X. Dong, and B. Xu, "An improved SDA based defect prediction framework for both within-project and cross-project class-imbalance problems," *IEEE Trans. Softw. Eng.*, vol. 43, no. 4, pp. 321–339, Apr. 2017.

[42] F. Zhang, A. E. Hassan, S. McIntosh, and Y. Zou, "The use of summation to aggregate software metrics hinders the performance of defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 5, pp. 476–491, May 2017.

[43] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 481–490.

[44] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Trans. Rel.*, vol. 62, no. 2, pp. 434–443, Jun. 2013.

[45] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data versus domain versus process," in *Proc. 7th Joint Meet. Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 91–100.

[46] Y. Ma, G. Luo, X. Zeng, and A. Chen, "Transfer learning for cross-company software defect prediction," *Inform. Softw. Technol.*, vol. 54, no. 3, pp. 248–256, 2012.

[47] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Softw. Eng.*, vol. 14, no. 5, pp. 540–578, 2009.

[48] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the imprecision of cross-project defect prediction," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng* 2012, Art. no. 61.

[49] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in *Proc. 5th Int. Conf. Predictor Models Softw. Eng.*, 2009, Art. no. 7.

[50] M. DAmbros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empirical Softw. Eng.*, vol. 17, no. 4–5, pp. 531–577, 2012.

[51] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proc. 6th Int. Conf. Predictive Models Softw. Eng.*, 2010, Art. no. 9.

[52] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 1–18, Jan. 2017.

[53] M. Leszak, D. E. Perry, and D. Stoll, "A case study in root cause defect analysis," in *Proc. 22rd Int. Conf. Softw. Eng.*, 2000, pp. 428–437.

[54] J. Ratzinger, T. Sigmund, and H. Gall, "On the relation of refactoring and software defects," in *Proc. 5th Int. Workshop Mining Softw. Repositories*, May 2008, pp. 35–38.

[55] D. Garlan, *Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events*. Berlin Heidelberg, Germany: Springer, 2003.

[56] A. Terry, et al., "Overview of teknowledge's domain-specific software architecture program," *SIGSOFT Softw. Eng. Notes*, vol. 19, no. 4, pp. 68–76, Oct. 1994. [Online]. Available: http://doi.acm.org/10.1145/190679.190686

[57] P. Newton and J. C. Browne, "The code 2.0 graphical parallel programming language," in *Proc. 6th Int. Conf. Supercomputing*, 1992, pp. 167–177. [Online]. Available: http://doi.acm.org/10.1145/143369.143405

[58] J. Palsberg, C. Xiao, and K. Lieberherr, "Efficient implementation of adaptive software," *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 2, pp. 264–292, Mar. 1995. [Online]. Available: http://doi.acm.org/10.1145/201059.201066

[59] F. Jahanian and A. K. Mok, "Modechart: A specification language for real-time systems," *IEEE Trans. Softw. Eng.*, vol. 20, no. 12, pp. 933–947, Dec. 1994. [Online]. Available: http://dx.doi.org/10.1109/32.368134

[60] S. H. Edwards, W. D. Heym, T. J. Long, M. Sitaraman, and B. W. Weide, "Part II: Specifying components in resolve," *ACM SIGSOFT Softw. Eng. Notes*, vol. 19, no. 4, pp. 29–39, 1994.

[61] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for software architecture and tools to support them," *IEEE Trans. Softw. Eng.*, vol. 21, no. 4, pp. 314–335, Apr. 1995. [Online]. Available: http://dx.doi.org/10.1109/32.385970

[62] R. Allen and D. Garlan, "Beyond definition/use: Architectural interconnection," in *Proc. Workshop Interface Definition Languages*, 1994, pp. 35–45. [Online]. Available: http://doi.acm.org/10.1145/185084.185101

[63] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: Connecting software architecture to implementation," in *Proc. 24th Int. Conf. Softw. Eng.*, May 2002, pp. 187–197.

[64] P. C. Clements, "A survey of architecture description languages," in *Proc. 8th Int. Workshop Softw. Specification Des.*, 1996, Art. no. 16. [Online]. Available: http://dl.acm.org/citation.cfm?id=857204.858261

[65] P. B. Kruchten, "The 4+1 view model of architecture," *IEEE Softw.*, vol. 12, no. 6, pp. 42–50, Nov. 1995.

[66] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2012.

[67] R. Kazman, D. Goldenson, I. Monarch, W. Nichols, and G. Valetto, "Evaluating the effects of architectural documentation: A case study of a large scale open source project," *IEEE Trans. Softw. Eng.*, vol. 42, no. 3, pp. 222–247, Mar. 2016.

[68] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 461–470. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985856

[69] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire, *Reverse Engineering Architectural Feature Models*. Berlin, Heidelberg, Germany: Springer, 2011, pp. 220–235. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23798-0_25

[70] L. P. Tizzei, C. M. F. Rubira, and J. Lee, "An aspect-based feature model for architecting component product lines," in *Proc. 38th Euromicro Conf. Softw. Eng. Adv. Appl.*, Sep. 2012, pp. 85–92.

[71] A. Shatnawi, A.-D. Seriai, and H. Sahraoui, "Recovering software product line architecture of a family of object-oriented product variants," *J. Syst. Softw.*, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121216301327

[72] S. Kebir, A.-D. Seriai, A. Chaoui, and S. Chardigny, "Comparing and combining genetic and clustering algorithms for software component identification from object-oriented code," in *Proc. 5th Int. Conf. Comput. Sci. Softw. Eng.*, 2012, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/2347583.2347584

[73] S. Kebir, A. D. Seriai, S. Chardigny, and A. Chaoui, "Quality-centric approach for software component identification from object-oriented code," in *Proc. Joint Work. IEEE/IFIP Conf. Softw. Archit. Eur. Conf. Softw. Archit.*, Aug. 2012, pp. 181–190.

[74] R. Weinreich, C. Miesbauer, G. Buchgeher, and T. Kriechbaum, "Extracting and facilitating architecture in service-oriented software systems," in *Proc. Joint Work. IEEE/IFIP Conf. Softw. Archit. Eur. Conf. Softw. Archit.*, Aug. 2012, pp. 81–90.

[75] Z. Alshara, A.-D. Seriai, C. Tibermacine, H. L. Bouziane, C. Dony, and A. Shatnawi, *Materializing Architecture Recovered from Object-Oriented Source Code in Component-Based Languages*. Cham, Switzerland: Springer International Publishing, 2016, pp. 309–325. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-48992-6_23

[76] J. Maranzano, S. Rozsypal, G. Zimmerman, G. Warnken, P. Wirth, and D. Weiss, "Architecture reviews: Practice and experience," *IEEE Softw.*, vol. 22, no. 2, pp. 34–43, Mar./Apr. 2005.

[77] R. Kazman, G. Abowd, L. Bass, and M. Webb, "SAAM: A method for analyzing the properties of software architectures," in *Proc. 16th Int. Conf. Softw. Eng.*, May 1994, pp. 81–90.

[78] R. Kazman, M. Barbacci, M. Klein, S. J. Carriere, and S. G. Woods, "Experience with performing architecture tradeoff analysis," in *Proc. 16th Int. Conf. Softw. Eng.*, May 1999, pp. 54–64.

[79] B. Tekinerdogan and H. Sozer, *Defining Architectural Viewpoints Quality Concerns*. Berlin, Heidelberg, Germany: Springer, 2011, pp. 26–34. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23798-0_3

[80] G. E. Boussaidi, A. B. Belle, S. Vaucher, and H. Mili, "Reconstructing architectural views from legacy systems," in *Proc. 19th Work. Conf. Reverse Eng.*, Oct. 2012, pp. 345–354.

[81] J. Garcia, I. Krka, N. Medvidovic, and C. Douglas, "A framework for obtaining the ground-truth in architectural recovery," in *Proc. Joint Work. IEEE/IFIP Conf. Softw. Archit. Eur. Conf. Softw. Archit.*, Aug. 2012, pp. 292–296.

[82] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *Proc. 28th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Nov. 2013, pp. 486–496.

[83] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic, "Obtaining ground-truth software architectures," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 901–910. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486911

[84] 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 Int. Conf. Softw. Maintenance*, Aug. 1999, pp. 50–59.

[85] V. Tzerpos and R. C. Holt, "ACDC: An algorithm for comprehension-driven clustering," in *Proc. 7th Work. Conf. Reverse Eng.*, Nov. 2000, pp. 258–267.

[86] 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.*, vol. 23, no. 1, pp. 4:1–4:33, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2559935

[87] M. von Detten, M. C. Platenius, and S. Becker, "Reengineering component-based software systems with archimetrix," *Softw. Syst. Model.*, vol. 13, no. 4, pp. 1239–1268, Oct. 2014. [Online]. Available: http://dx.doi.org/10.1007/s10270-013-0341-9

[88] W. Cunningham, "The WyCash portfolio management system," in *Proc. 7th ACM SIGPLAN Conf. Object-Oriented Program. Syst. Languages Appl.*, Oct. 1992, pp. 29–30.

[89] C. Seaman and Y. Guo, "Measuring and monitoring technical debt," *Advances Comput.*, vol. 82, no. 25–46, 2011, Art. no. 44.

[90] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *J. Syst. Softw.*, vol. 86, no. 6, pp. 1498–1516, 2013.

[91] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Softw.*, vol. 29, no. 6, pp. 18–21, Nov./Dec. 2012.

[92] F. Shull, D. Falessi, C. Seaman, M. Diep, and L. Layman, "Technical debt: Showing the way for better transfer of empirical results," in *Perspectives on the Future of Software Engineering*. Berlin Heidelberg, Germany: Springer, 2013, pp. 179–190. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37395-4_12

[93] D. Falessi, P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt at the crossroads of research and practice: Report on the fifth international workshop on managing technical debt," *ACM SIGSOFT Softw. Eng. Notes*, vol. 39, no. 2, pp. 31–33, 2014. [Online]. Available: http://doi.acm.org/10.1145/2579281.2579311

[94] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM J. Res. Develop.*, vol. 56, no. 5, pp. 9–1, 2012.

[95] L. Peters, "Technical debt: The ultimate antipattern - the biggest costs may be hidden, widespread, and long term," Washington, DC, USA, pp. 8–10, 2014. [Online]. Available: http://dx.doi.org/10.1109/MTD.2014.7

[96] N. S. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spinola, "Towards an ontology of terms on technical debt," in *Proc. 6th Int. Workshop Manag. Tech. Debt*, 2014, pp. 1–7.

[97] C. Seaman, R. L. Nord, P. Kruchten, and I. Ozkaya, "Technical debt: Beyond definition to understanding report on the sixth international workshop on managing technical debt," *SIGSOFT Softw. Eng. Notes*, vol. 40, no. 2, pp. 32–34, Apr. 2015. [Online]. Available: http://doi.acm.org/10.1145/2735399.2735419

[98] N. Zazworka, R. O. Spínola, A. Vetro, F. Shull, and C. Seaman, "A case study on effectively identifying technical debt," in *Proc. 17th Int. Conf. Eval. Assessment Softw. Eng.*, 2013, pp. 42–47.

[99] E. Lim, N. Taksande, and C. Seaman, "A balancing act: What software practitioners have to say about technical debt," *IEEE Softw.*, vol. 29, no. 6, pp. 22–27, Nov./Dec. 2012.

[100] C. Seaman, et al., "Using technical debt data in decision making: Potential decision approaches," in *Proc. 3rd Int. Workshop Manag. Tech. Debt*, 2012, pp. 45–48.

[101] A. Nugroho, J. Visser, and T. Kuipers, "An empirical model of technical debt and interest," in *Proc. 2nd Workshop Manag. Tech. Debt.*, 2011, pp. 1–8.

[102] Y. Guo and C. Seaman, "A portfolio approach to technical debt management," in *Proc. 2nd Workshop Manag. Tech. Debt.*, 2011, pp. 31–34.

[103] Y. Guo, et al., "Tracking technical debtan exploratory case study," in *Proc. 27th IEEE Int. Conf. Softw. Maintenance*, 2011, pp. 528–531.

[104] J.-L. Letouzey and M. Ilkiewicz, "Managing technical debt with the SQALE method," *IEEE Softw.*, vol. 29, no. 6, pp. 44–51, Nov./Dec. 2012.

[105] N. Brown, et al., "Managing technical debt in software-reliant systems," pp. 47–52, 2010. [Online]. Available: http://doi.acm.org/10.1145/1882362.1882373

[106] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the size, cost, and types of technical debt," in *Proc. 3rd Int. Workshop Manag. Tech. Debt*, 2012, pp. 49–53.

[107] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In search of a metric for managing architectural technical debt," in *Proc. Joint Work. IEEE/IFIP Conf. Softw. Archit. (WICSA) Eur. Conf. Softw. Archit.*, 2012, pp. 91–100.

[108] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *J. Syst. Softw.*, vol. 101, no. C, pp. 193–220, Mar. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2014.12.027

[109] N. Zazworka, C. Izurieta, S. Wong, Y. Cai, F. Shull, and others, "Comparing four approaches for technical debt identification," *Softw. Quality J.*, vol. 22, no. 3, pp. 403–426, 2014.

[110] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 488–498.

[111] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage API usage patterns from source code," in *Proc. 10th Work. Conf. Mining Softw. Repositories*, May 2013, pp. 319–328.

[112] M. Bruch, T. Schäfer, and M. Mezini, "Fruit: IDE support for framework understanding," in *Proc. OOPSLA Workshop Eclipse Technol. eXchange*, 2006, pp. 55–59. [Online]. Available: http://doi.acm.org/10.1145/1188835.1188847

[113] A. Shatnawi, et al., "Analyzing program dependencies in java EE applications," in *Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories*, May 2017, pp. 64–74.

[114] L. Xiao and T. Yu, "Ripple: A test-aware architecture modeling framework," in *Proc. 1st Int. Workshop Establishing Community-Wide Infrastructure Archit.-Based Softw. Eng.*, 2017, pp. 14–20. [Online]. Available: https://doi.org/10.1109/ECASE.2017.2

[115] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: Examining the effects of ownership on software quality," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.*, 2011, pp. 4–14. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025119

[116] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. Int. Workshop Mining Softw. Repositories*, 2005, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083147

[117] 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 *Proc. Conf. Center Adv. Stud. Collaborative Res.: Meet. Minds*, 2008, pp. 23:304–23:318. [Online]. Available: http://doi.acm.org/10.1145/1463788.1463819

[118] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 392–401. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486840

Yuanfang Cai received the PhD degree in computer science from the University of Virginia. She is currently an associate professor with Drexel University. Her research focuses on software design, software architecture, software evolution, and software economics. Her recent work investigates architecture issues that are the root cause of software defects, and the quantification of architectural debt. She is currently serving on program committees and organizing committees for multiple top conferences and the editorial board of top journals in the area of software engineering. The tools and technologies from her research have been licensed and adopted by multiple multinational corporations.

Lu Xiao received the PhD degree in computer science from Drexel University, in 2016, advised by Dr. Yuanfang Cai. She is an assistant professor in the School of Systems and Enterprises at Stevens Institute of Technology. Her research focues on software architecture, software evolution and maintenance. In particular, she is interested in modeling and analyzing software architecture and its evolution for addressing quality problems, such as maintenance quality and performance.

Rick Kazman is a professor with the University of Hawaii and a research scientist in the Software Engineering Institute of Carnegie Mellon University. His primary research interests include the software architecture, design and analysis tools, software visualization, and software engineering economics. He has created several highly influential methods and tools for architecture analysis, including the SAAM (Software Architecture Analysis Method), the ATAM (Architecture Tradeoff Analysis Method), the CBAM (Cost-Benefit Analysis Method) and the Dali and Titan tools. He is the author of more then 200 publications, and co-author of several books, including the *Software Architecture in Practice*, the *Designing Software Architectures: A Practical Approach*, the *Evaluating Software Architectures: Methods and Case Studies*, and the *Ultra-Large-Scale Systems: The Software Challenge of the Future*.

Ran Mo is working toward the PhD degree in the Computer Science Department, Drexel University. His research mainly focuses on measuring and analyzing the quality of software architecture.

Qiong Feng is working toward the PhD degree in the Computer Science Department, Drexel University. Her research mainly focuses on analyzing the evolution of software architecture.