Architecture Anti-patterns: Automatically Detectable Violations of Design Principles

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

Abstract—In large-scale software systems, error-prone or change-prone files rarely stand alone. They are typically architecturally connected and their connections usually exhibit architecture problems causing the propagation of error-proneness or change-proneness. In this paper, we propose and empirically validate a suite of *architecture anti-patterns* that occur in all large-scale software systems and are involved in high maintenance costs. We define these architecture anti-patterns based on fundamental design principles and Baldwin and Clark's design rule theory. We can automatically detect these anti-patterns by analyzing a project's structural relationships and revision history. Through our analyses of 19 large-scale software projects, we demonstrate that these architecture anti-patterns have significant impact on files' bug-proneness and change-proneness. In particular, we show that 1) files involved in these architecture anti-patterns are more error-prone and change-prone; 2) the more anti-patterns a file is involved in, the more error-prone and change-prone it is; and 3) while all of our defined architecture anti-patterns contribute to file's error-proneness and change-proneness, *Unstable Interface* and *Crossing* contribute the most by far.

Index Terms—Software Architecture, Software Maintenance, Software Quality

1 Introduction

In long-lived software projects, bug-prone and change-prone files consume the majority of overall maintenance effort. Numerous bug prediction [2], [3], [4] or localization [5], [6], [7] approaches have been proposed to locate bug-prone or change-prone "units" in source code. It has typically been difficult for developers to modify a single file without unexpected changes to other files, and multiple files often need to be changed together for a maintenance task. Our prior work [8], [9], [10] revealed that bug-prone files in a project are usually connected, and their connections often exhibit architecture problems.[1] These anti-patterns, as we will show, propagate bug-proneness among files. For example, a buggy interface can propagate to the files implementing it; when this interface is changed, its concrete classes often need to be changed as well.

Researchers have proposed various methods to identify problematic areas within source code, such as code smells [11], architecture smells [12] or anti-patterns [13]. Existing tools like SonarQube[2], AiReviewer[3], Designite[4] etc. support the detection of such smells.

These tools like SonarQube often report large numbers of low-level smells, not architecture-level violations, and many of these low-level problems do not incur high maintenance costs [14], [15]. For example, cloned code that never changes may not need attention. Tools like AiReviewer and Designite also report many types of problems based on violations of design principles, but no empirical evidence has been offered to show if and to what extent these problems have severe consequences in terms of bug-proneness and change-proneness. Different from design smells or architecture smells proposed in previous studies, our objective is to pinpoint architectural problems with *severe* consequences, identified through the combination of structural information and revision history records.

In this paper, we present our approach to detecting *architecture anti-patterns*, defined as connections among files that violate design principles and impact bug-proneness and change-proneness[5]. Different from existing work [11], [13], [12], [16], our definition uses both a project's structural information and its revision history to detect anti-patterns. After examining the source code and revision histories of over 100 open-source and industrial software projects, we have observed that there are just a few distinct types of architecture anti-patterns that occur in *all* the projects we have studied. We will show that files involved in these anti-patterns tend to have more bugs, more changes, and consume more effort, and thus they should be identified. To improve the quality and productivity of a project, the architectural problems behind these file groups should be

- *This article is an extended version of [1] presented at the 12th Working IEEE / IFIP Conference on Software Architecture(WICSA) in 2015. We have extended our work as follows: 1) In this paper, we have formally defined one new Architecture anti-pattern,* Crossing*; 2) We have improved the algorithms for* Clique *and* Modularity Violation Group*; 3) we have added pairwise comparisons for all types of architecture anti-patterns to examine and discuss the severity of each type of anti-patterns in terms of maintenance costs; 4) we have extended the simple tool support described in our conference paper to a full functional tool; 5) in the original paper, we only studied ten projects, but in this paper, we studied 19 projects in total (including 15 open source projects and 4 commercial projects).*

[1] In prior work [1], we called these architecture problems "*issues*" or "*hotspots*". Since these terms were over-loaded, we chose to use the term *"architecture anti-pattern"*.

[2] https://www.sonarqube.org/

[3] http://www.aireviewer.com/

[4] http://www.designite-tools.com/

[5] In this paper, a *file* means a source file, which contains one or more classes

fixed first. We summarize the six architecture *anti-patterns* as follows:

1) Unstable Interface. Based on design rule theory [17] and prevailing design principles [18], an influential interface with many dependents should remain stable. In reality, we have observed that if influential files have high change rates, then multiple files depending on them have to be changed as a consequence.

2) Modularity Violation Groups. Based on design rule theory [17], truly independent modules should evolve independently. Our prior work [1] introduced the concept of *Implicit Cross-module Dependency* to identify modules that have changed together frequently, as evidenced in revision history, but have no structural dependency. In this paper, we modified the algorithm to detect the *Modularity Violation Group*, which contains a set of modularity violation file pairs (two files without structural relations but changed together frequently in revision history) .

3) Unhealthy Inheritance Hierarchy. This anti-pattern identifies the violation of the Liskov Substitution principle [18], [19] or Dependency Inversion Principle [18]. This anti-pattern includes cases where the parent class depends on one or more of this subclasses, or the client of an inheritance hierarchy depends on both the parent class and its children. These cases undermine the objective of an inheritance hierarchy to enable polymorphism. As a result, this structure propagates bugs and changes to files that depend on this inheritance hierarchy.

4) Crossing. In this paper, we extend our prior work [1] with a newly defined architecture anti-pattern, which we call Crossing: a file that has both high fan-in and high fan-out, and changes often together with its dependents and the files it depends on, is often at the center of maintenance activities. Since files involved in this anti-pattern form a cross shape in a Design Structure Matrix [17], we call it *Crossing*.

5) Clique. Cyclic dependency is a well-know design problem. Instead of detecting pair-wise cycles [1], we define Clique as a set of files that form a strongly connected graph with direct or indirect cycles among files. Files in each Clique instance are tightly coupled with one or more dependency cycles.

6) Package Cycle. Dependency cycles between packages violates the basic design principle of forming a hierarchical structure [20]. We have observed that changes to a file in one package often cause unexpected changes to files in other packages due to the cycle of dependencies between them.

To evaluate these architecture anti-patterns, we investigate following research questions:

- RQ1: Do the files involved in architecture anti-pattern consume significantly more maintenance effort than other files in a project?
- RQ2: If a file is involved in greater numbers of architecture anti-patterns, is it more error-prone and/or change-prone?
- RQ3: Do different types of architecture anti-patterns have different impacts on a file's overall error-proneness and change-proneness?

To answer these questions, we present the analysis of fifteen open source projects and four commercial projects. These projects have different sizes, ages, and domains. Our results show that these six architecture anti-patterns are strongly correlated with higher bug-proneness and change-proneness—files that are involved in these anti-patterns are more bug-prone or change prone. After analyzing files involved in 0 to 6 anti-patterns, we observed that their bug rates and change rates increase dramatically as the number of architecture anti-patterns in which they are involved increases. Furthermore, we identify and quantify the most severe anti-patterns in software systems. Of all 6 patterns, our analysis shows that *Unstable Interface* and *Crossing* have the most significant contributions to error-proneness and change-proneness.

This paper is an extension of our prior work [1], in which 5 hotspot patterns were defined. In this paper, we have added one more anti-pattern, Crossing, and improved the algorithms for detecting Clique and Modularity Violation Group. An industrial case study of the tool presented in this paper has been recently published [21]; this paper focuses on the detailed formalization and thorough empirical evaluation of the underlying technology.

2 Background

In this section, we introduce the conceptual foundations of our work.

Design Rule Theory. According to Baldwin and Clark's *Design Rule Theory* [17], software should be structured by *design rules* and *independent modules*. In a software system, design rules are often manifested as the important design decisions, which decouple the rest of the system into *independent modules*. A design rule is typically manifested as an interface or abstract class. For example, if an Observer Pattern [22] is used in a code base, then there must exist an observer interface that decouples the subject and concrete observers into independent modules. As long as the interface is stable, addition, removal, or changes to concrete observers should not influence the subject. In this case, the observer interface is considered as a *design rule*, decoupling the subject and concrete observers into two *independent modules*. For another example, if a Strategy Pattern [22] is implemented, then the strategy interface is considered as the design rule which decouples the context and concrete strategies into independent modules.

Design Rule Hierarchy (DRH). To automatically identify the design rules and independent modules in software systems, our prior work introduced a clustering algorithm—*Design Rule Hierarchy* (DRH) [23], [24], [25], which clusters the file relation of a system into a hierarchical structure. Within such a hierarchy, files in layer L_i should only depend on files in the higher layers, L_{i-1} to L_1, and files in layer L_i should not depend on files in the lower layers, L_{i+1} to L_n. Hence files in the first layer, L_1, should contain most influential interfaces or abstract classes, which do not depend on files in other layers. In addition, files in the same layer are decoupled into a set of *modules* that are mutually independent from each other. Thus the changes, addition, even replacement to a *module* will not influence other *modules* within the same layer. Accordingly, *independent modules* in the bottom layer of a design rule hierarchy are most valuable, because changes to these modules will not affect the rest of the system.

Design Structure Matrix (DSM). We use a *Design Structure Matrix (DSM)* to visualize file relations. A DSM is a square matrix, in which rows and columns are labeled with file names in the same order. An annotation in the cell in row x, column y, $c(rx, cy)$, indicates that there is a dependency relation between file x and file y: file x either structurally depends on file y, or file x and file y were changed together as recorded in the project's commit history.

The DSM in Figure 1 presents a design rule hierarchy (DRH) with 3 layers: $L1 : (rc1 - rc2)$, $L2 : (rc3 - rc11)$, $L3 : (rc12 - rc32)$. The first layer, $L1$, contains the most influential design rules that should remain stable. Files in $L2$ only depend on files in $L1$. Similarly, files in $L3$ only depend on files in the first two layers. Within each layer, files are grouped into mutually independent *modules*. Taking the bottom layer L3 as an example: it is grouped into 8 mutually independent modules: $M1 : (rc12)$, $M2 : (rc13 - rc16)$, $M3 : (rc17 - rc18)$, etc. We can see that there are no dependencies between these modules.

The text in a cell is used to indicate specific types of dependencies between the files. For example, $cell(r4, c1)$ in Figure 1 is marked with $''dp''$, which means *ExpressionBuilder_java "depends on"* (calls methods from) *ExpressionDefinition_java*. As we mentioned before, a DSM can also represent evolutionary coupling between files—the number of times two files were changed together. In Figure 2, a cell with just a number means that there is no structural relation between these two files, but they have been co-committed. For example, $cell(r8, c3)$ is only marked with "4", which means that there is no *structural* relation between *BeanExpression_java* and *MethodNotFoundException_java*, but they were changed together 4 times. A cell with both a number and text means that the two files have both *structural* and *evolutionary coupling* relations. For example, $cell(r22, c1)$ is marked with $''dp, 3''$, which means that *XMLTokenizerExpression_java* depends on *ExpressionDefinition_java*, and they were changed together 3 times.

Design Rule Space. We recently proposed the concept of *Design Rule Space* (DRSpace) [8], [9] to model the fact that the architecture of a software system can and should be represented as a set of overlapping design spaces, each reflecting an unique aspect of the architecture. For example, each feature implemented, or each pattern applied can be modeled as an individual design space. Each DRSpace contains one or more "leading files", typically *design rules* that all the other files within the design space depend on directly or indirectly. In other words, files within a DRSpace are architecturally connected. For example, if a strategy pattern is applied, all the files in the pattern can be represented as a DRSpace, and the strategy interface will be the "leading file" that all other files depend on. If an inheritance hierarchy is implemented, all files involved in the inheritance tree could form a DRSpace led by the parent class. More generally, a DRSpace is a subset of files connected by one or more relations, such as inheritance, call, etc. For any non-trivial project, there are numerous DRSpaces.

In these prior studies [8], [9], we demonstrated that the majority of error-prone files can be captured by just a few DRSpaces, suggesting that most error-prone files in a project are architecturally connected. We named these DRSpace *Architecture Roots*, and argued that these roots typically contain architectural design problems which could propagate error-proneness among multiple files.

3 Architecture Anti-Patterns

In this paper, we define a suite of architecture anti-patterns based on Baldwin and Clark's design rule theory [17] and widely accepted design principles, especially the SOLID principles proposed by Robert Martin [18]. We have observed that the design rule theory explains these informal principles so that they can be visualized and possibly quantified. According to design rule theory, a well-modularized system should have the following features: first, design rules have to be stable—neither error-prone nor change-prone. Second, if two modules are truly independent, then they should only depend on design rules, but not on each other. More importantly, independent modules should be able to be changed, or even replaced, without influencing each other, as long as the design rules themselves remain unchanged. The importance of design rules, that is, abstractions, are also reflected in the Liskov substitution, Interface segregation, and Dependency inversion principles. The Single responsibility and Open-closed principles suggest the importance of module independence. After analyzing a large number of industrial and open source projects, we have summarized six types of recurring problems into a suite of *Architecture Anti-Patterns*, each violating one or more design principles and/or design rule theory. Next we define some basic terms and introduce the rationale and formalization of these anti-patterns.

3.1 Definitions

We use the following terms to model the basic concepts used in our definition:
F—the set of all the files: $F = \{f_i \mid i \in \mathbb{N}\}$

We use the following notions to model structural and evolutionary relation among files of a project:

$depend(x, y)$: x depends y, i.e., x calls methods from y.

$inherit(x, y)$: x inherits from or realizes y, e.g., y is the parent class of x, or y is an interface and x implement it.

$\#cochange(x, y)$: the number of times x was committed together with y in a given period of time based on the revision history. Gall et al.'s [2] proposed that evolutionary coupling between two files could be reflected by how often they were committed together as recorded in the revision history. The more often two files change together, the stronger their evolution coupling.

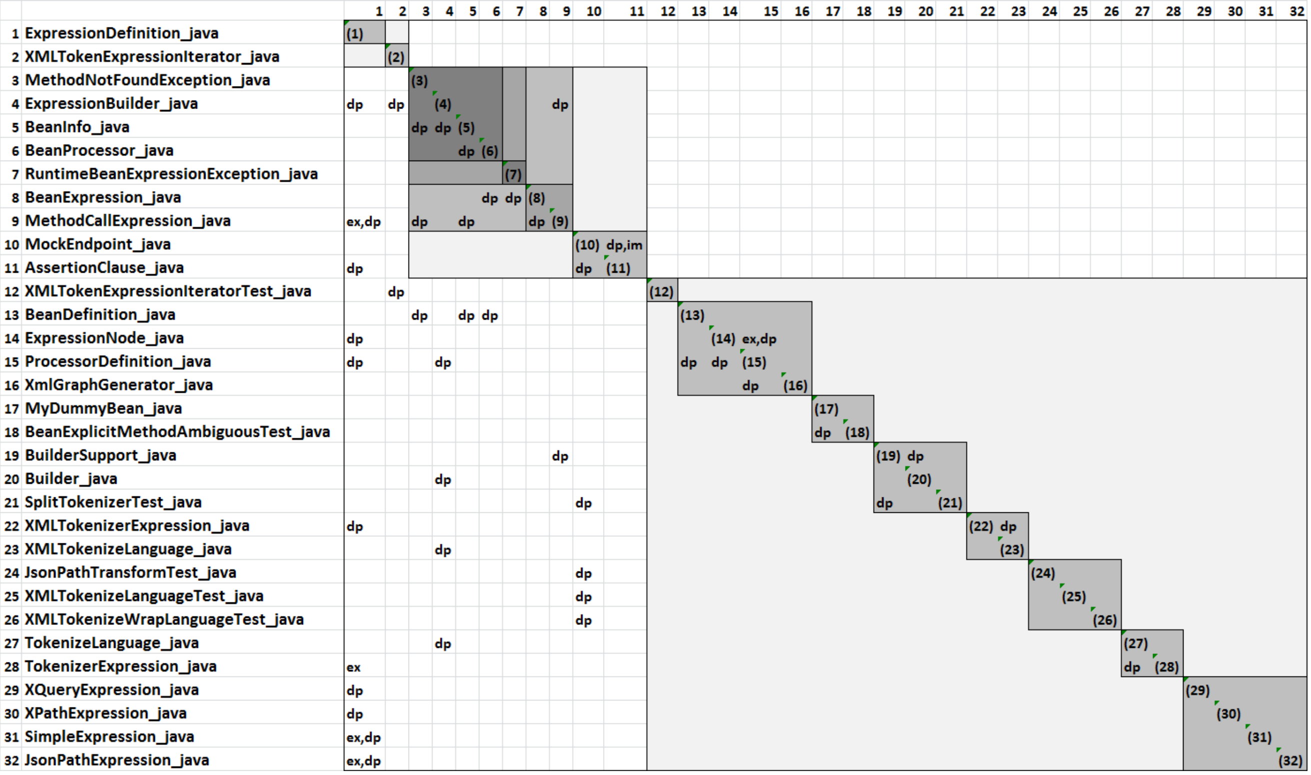
$SRelation(x, y)$: structural relations from file x to file y, such as *Implement*, *Extend*, *depend*, etc.

3.2 Architecture Anti-Patterns

For each architecture anti-pattern, we now introduce its rationale, description, and formalization.

1. Unstable Interface (UIF).
Rationale: According to the design rule theory [17] and design principles [18], important and influential abstractions (design rules) should be stable. Otherwise their bugs and changes can be propagated to multiple files. We have

Fig. 1: An example of DRH exhibiting structural relations among files



	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
1 ExpressionDefinition_java	(1)																															
2 XMLTokenExpressionIterator_java		(2)																														
3 MethodNotFoundException_java			(3)																													
4 ExpressionBuilder_java	dp	dp		(4)					dp																							
5 BeanInfo_java			dp	dp	(5)																											
6 BeanProcessor_java					dp	(6)																										
7 RuntimeBeanExpressionException_java							(7)																									
8 BeanExpression_java						dp	dp	(8)																								
9 MethodCallExpression_java	ex,dp		dp		dp			dp	(9)																							
10 MockEndpoint_java										(10)	dp,im																					
11 AssertionClause_java	dp									dp	(11)																					
12 XMLTokenExpressionIteratorTest_java		dp										(12)																				
13 BeanDefinition_java			dp		dp	dp							(13)																			
14 ExpressionNode_java	dp													(14)	ex,dp																	
15 ProcessorDefinition_java	dp			dp									dp	dp	(15)																	
16 XmlGraphGenerator_java															dp	(16)																
17 MyDummyBean_java																	(17)															
18 BeanExplicitMethodAmbiguousTest_java																	dp	(18)														
19 BuilderSupport_java									dp										(19)	dp												
20 Builder_java				dp																(20)												
21 SplitTokenizerTest_java										dp									dp		(21)											
22 XMLTokenizerExpression_java	dp																					(22)	dp									
23 XMLTokenizeLanguage_java				dp																			(23)									
24 JsonPathTransformTest_java										dp														(24)								
25 XMLTokenizeLanguageTest_java										dp															(25)							
26 XMLTokenizeWrapLanguageTest_java										dp																(26)						
27 TokenizeLanguage_java				dp																							(27)					
28 TokenizerExpression_java	ex																										dp	(28)				
29 XQueryExpression_java	dp																												(29)			
30 XPathExpression_java	dp																													(30)		
31 SimpleExpression_java	ex,dp																														(31)	
32 JsonPathExpression_java	ex,dp																															(32)

ex: Extend; im: Implement dp: Depend

observed that unstable or poorly-designed abstractions are often related to high-maintenance, and deserve special attention.

Description: If a highly influential file (files with a large number of dependents) is changed frequently with other files as shown in the revision history, then we call it an *Unstable Interface* (UIF).

Formalization:
$StructImpact_{thr}$: the threshold of the structural impact scope of a file, f_i. If the number of its dependents is larger than the threshold, we consider f_i to be a candidate unstable interface.

$HistoryImpact_{thr}$: the threshold of the number of co-changed dependents of f_i. This requires that f_i not only has more than $StructImpact_{thr}$ dependents, but also has more than $HistoryImpact_{thr}$ of these dependents changed together frequently with it.

$cochange_{thr}$: the threshold of co-change frequency. If two files are committed together more times than this threshold, we consider these two files to have changed together "frequently" and that they are *evolutionarily coupled*.

For a file f_i, if it has more than $StructImpact_{thr}$ dependents, and more than $HistoryImpact_{thr}$ of these dependents changed together with it more than $cochange_{thr}$ times, we consider f_i to be an *Unstable Interface*. Formally:

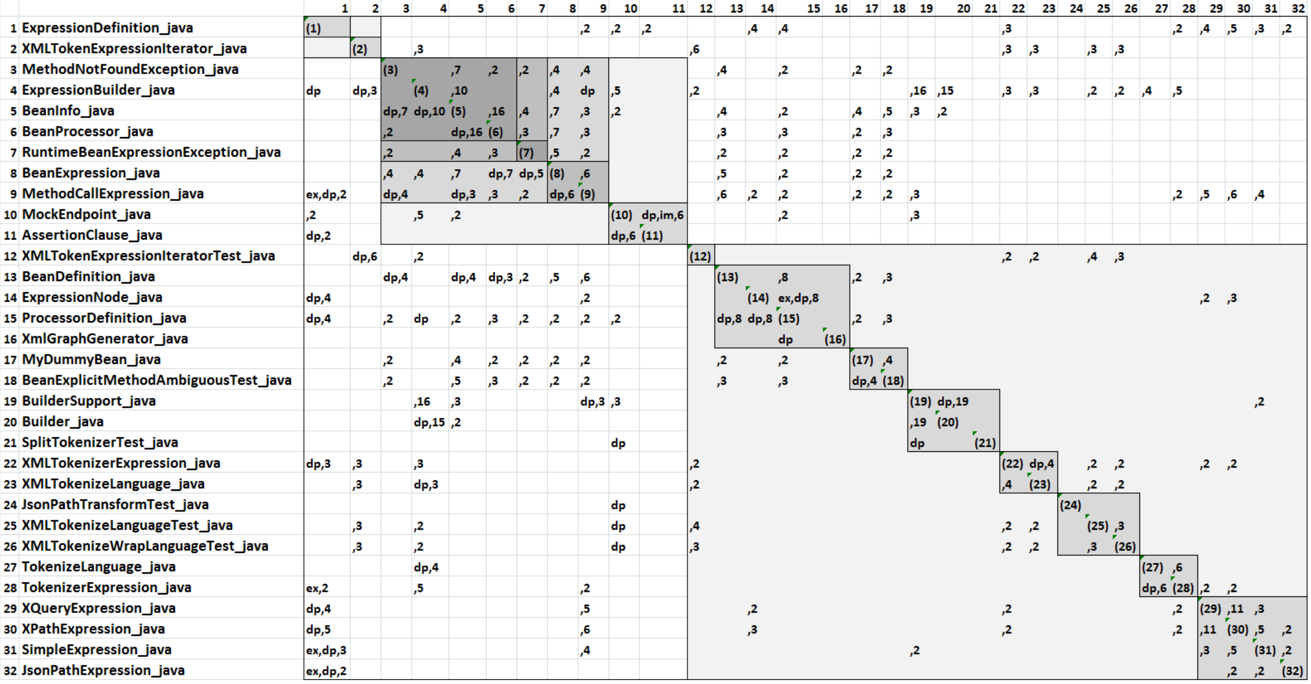
$$\begin{aligned}
&\exists f_1 \in F \mid \{F_{set\_S} : \forall f_i \in F_{set\_S} \mid SRelation(f_i, f_1)\} \wedge \\
&\{F_{set\_H} : \forall f_j \in F_{Set\_H} \mid \#cochange(f_j, f_1) > cochange_{thr}\} \\
&\wedge (|F_{Set}| > StructImpact_{thr}) \\
&\wedge (|F_{Set\_S} \cap F_{set\_H}| > HistoryImpact_{thr})
\end{aligned} \tag{1}$$

where $i \in \{1, 2, 3, ..., n\}$, n is the number of files in F_{set_S}, $j \in \{1, 2, 3, ..., m\}$, m is the number of files in F_{set_H}. $StructImpact_{thr}$, $cochange_{thr}$ and $HistoryImpact_{thr}$ are the thresholds that are configurable.

Figure 3 depicts an instance of Unstable Interface in the Cassandra project. An $''x''$ in a cell indicates a structural dependency between the file on the row and the file on the column; a number represents the historical co-change frequency of these two files. We can see that multiple files structurally depend on *StreamSession_java* and that these files have changed together frequently with it as evidenced by the project's revision history.

2. Modularity Violation Group (MVG).
Rationale: Baldwin and Clark's Design Rule theory [17] proposed that independent modules can be changed or even replaced without influencing each other. Wong et al. introduced the term *Modularity Violation* [26], which describes two structurally independent modules that change together frequently, meaning that they are not truly independent. The more often two structurally unrelated files change together, the more likely that there are implicit dependencies between them [26], [27]. In this paper, we extend our prior work [1] to calculate the minimal number of file groups with modularity violations.

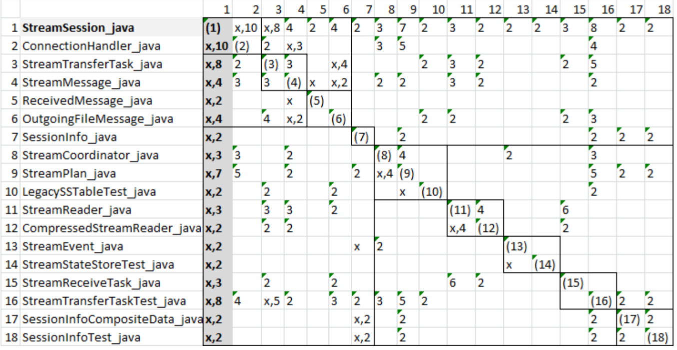
Fig. 2: An example of DRH exhibiting structural relations and evolutionary coupling among files



		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
1	ExpressionDefinition_java	(1)								,2	,2	,2			,4	,4							,3						,2	,4	,5	,3	,2
2	XMLTokenExpressionIterator_java		(2)		,3								,6										,3	,3		,3	,3						
3	MethodNotFoundException_java			(3)		,7	,2	,2	,4	,4				,4		,2		,2	,2														
4	ExpressionBuilder_java	dp	dp,3		(4)	,10			,4	dp	,5		,2							,16	,15		,3	,3		,2	,2	,4	,5				
5	BeanInfo_java			dp,7	dp,10	(5)	,16	,4	,7	,3	,2			,4		,2		,4	,5	,3	,2												
6	BeanProcessor_java			,2		dp,16	(6)	,3	,7	,3				,3		,3		,2	,3														
7	RuntimeBeanExpressionException_java			,2		,4	,3	(7)	,5	,2				,2		,2		,2	,2														
8	BeanExpression_java			,4	,4	,7	dp,7	dp,5	(8)	,6				,5		,2		,2	,2														
9	MethodCallExpression_java	ex,dp,2		dp,4		dp,3	,3	,2	dp,6	(9)				,6	,2	,2		,2	,2	,3									,2	,5	,6	,4	
10	MockEndpoint_java	,2			,5	,2					(10)	dp,im,6				,2				,3													
11	AssertionClause_java	dp,2									dp,6	(11)																					
12	XMLTokenExpressionIteratorTest_java		dp,6		,2								(12)										,2	,2		,4	,3						
13	BeanDefinition_java			dp,4		dp,4	dp,3	,2	,5	,6				(13)		,8		,2	,3														
14	ExpressionNode_java	dp,4								,2					(14)	ex,dp,8														,2	,3		
15	ProcessorDefinition_java	dp,4		,2	dp	,2	,3	,2	,2	,2	,2			dp,8	dp,8	(15)		,2	,3														
16	XmlGraphGenerator_java															dp	(16)																
17	MyDummyBean_java			,2		,4	,2	,2	,2	,2				,2		,2		(17)	,4														
18	BeanExplicitMethodAmbiguousTest_java			,2		,5	,3	,2	,2	,2				,3		,3		dp,4	(18)														
19	BuilderSupport_java				,16	,3				dp,3	,3									(19)	dp,19											,2	
20	Builder_java				dp,15	,2														,19	(20)												
21	SplitTokenizerTest_java										dp									dp		(21)											
22	XMLTokenizerExpression_java	dp,3	,3		,3								,2										(22)	dp,4		,2	,2			,2	,2		
23	XMLTokenizeLanguage_java		,3		dp,3								,2										,4	(23)		,2	,2						
24	JsonPathTransformTest_java										dp														(24)								
25	XMLTokenizeLanguageTest_java		,3		,2						dp		,4										,2	,2		(25)	,3						
26	XMLTokenizeWrapLanguageTest_java		,3		,2						dp		,3										,2	,2		,3	(26)						
27	TokenizeLanguage_java				dp,4																							(27)	,6				
28	TokenizerExpression_java	ex,2			,5					,2																		dp,6	(28)	,2	,2		
29	XQueryExpression_java	dp,4								,5					,2								,2						,2	(29)	,11	,3	
30	XPathExpression_java	dp,5								,6					,3								,2						,2	,11	(30)	,5	,2
31	SimpleExpression_java	ex,dp,3								,4										,2										,3	,5	(31)	,2
32	JsonPathExpression_java	ex,dp,2																													,2	,2	(32)

ex: Extend; im: Implement dp: Depend

Fig. 3: An example of Unstable Interface



		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	**StreamSession_java**	(1)	x,10	x,8	4	2	4	2	3	7	2	3	2	2	2	3	8	2	2
2	ConnectionHandler_java	x,10	(2)	2	x,3				3	5							4		
3	StreamTransferTask_java	x,8	2	(3)	3		x,4				2	3	2			2	5		
4	StreamMessage_java	x,4	3	3	(4)	x	x,2		2	2		3	2				2		
5	ReceivedMessage_java	x,2			x	(5)													
6	OutgoingFileMessage_java	x,4		4	x,2		(6)				2	2				2	3		
7	SessionInfo_java	x,2						(7)		2							2	2	2
8	StreamCoordinator_java	x,3	3		2				(8)	4				2			3		
9	StreamPlan_java	x,7	5		2			2	x,4	(9)							5	2	2
10	LegacySSTableTest_java	x,2		2			2			x	(10)						2		
11	StreamReader_java	x,3		3	3		2					(11)	4			6			
12	CompressedStreamReader_java	x,2		2	2							x,4	(12)			2			
13	StreamEvent_java	x,2						x	2					(13)					
14	StreamStateStoreTest_java	x,2												x	(14)				
15	StreamReceiveTask_java	x,3		2			2					6	2			(15)			
16	StreamTransferTaskTest_java	x,8	4	x,5	2		3	2	3	5	2						(16)	2	2
17	SessionInfoCompositeData_java	x,2						x,2		2							2	(17)	2
18	SessionInfoTest_java	x,2						x,2		2							2	2	(18)

x indicates structural dependencies, such as extend, depend, etc.

Description: A *Modularity Violation Group* (MVG) contains a set of modularity violation files. We calculate the minimal number of MVGs so that their union covers all *violated* file pairs (two files without structural relations but changed together) in a project. In a *Modularity Violation Group*, there exists a *core file*, f_{core}, which all other files are not structurally related to, but have frequently changed together with. To identify a *Modularity Violation Group (MVG)*, our tool first generates all filesets by considering each file in a project as a *core file*, then greedily searches a fileset that covers most *violated* file pairs as a MVG, until the union of all the MVGs covers all *violated* file pairs in a project.

Formalization:

$$\begin{aligned} & MVG_1 \cup MVG_2 \cup MVG_3 \cup \cdots \cup MVG_n = P \\ & \mid f_{core} \in MVG_i, \forall f_j \in MVG_i, f_j \neq f_{core} \mid \\ & \neg SRelation(f_{core}, f_j) \wedge \neg SRelation(f_j, f_{core}) \\ & \wedge (\#cochange(f_{core}, f_j) > cochange_{thr}) \end{aligned} \tag{2}$$

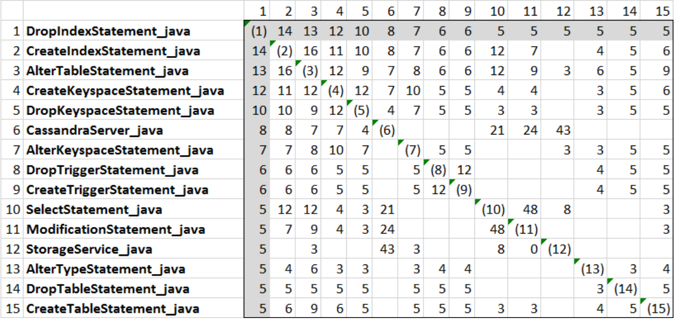
where $i \in \{1, 2, 3, ..., n\}$, n is the minimal number of MVGs whose union covers all *violated* file pairs. P is the set of all *violated* file pairs. f_j is a file that is structurally independent from f_{core}, but frequently changes together with it. $cochange_{thr}$ is a configurable threshold of co-change frequency, above which we consider the two files as evolutionary coupled.

Figure 4 shows an instance of MVG detected in Apache *Cassandra*. There are no structural dependencies between *DropIndexStatement_java* and the other files. However, the cells annotated with a number in the DSM reveal that all other files changed together with *DropIndexStatement_java*, the *core file*. Here, we uniformly assume a $cochange_{thr}$ of 2.

3. Unhealthy Inheritance Hierarchy (UIH). *Rationale:* We have encountered a surprisingly large number of cases where basic object-oriented design principles are violated in the implementation of an inheritance hierarchy. The two most frequent problems are: (1) a parent class depends on one of its children; and (2) a client class of the hierarchy depends on both the base class and its children. Both cases violates Liskov Substitution principle[6] [19], since the parent class can no longer be a placeholder substitutable by any of its children. They also violates the Design Rule Theory [17] because the parent class cannot be a decoupling

[6] https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start

Fig. 4: An example instance of Modularity Violation Group



		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	**DropIndexStatement_java**	(1)	14	13	12	10	8	7	6	6	5	5	5	5	5	5
2	**CreateIndexStatement_java**	14	(2)	16	11	10	8	7	6	6	12	7		4	5	6
3	**AlterTableStatement_java**	13	16	(3)	12	9	7	8	6	6	12	9	3	6	5	9
4	**CreateKeyspaceStatement_java**	12	11	12	(4)	12	7	10	5	5	4	4		3	5	6
5	**DropKeyspaceStatement_java**	10	10	9	12	(5)	4	7	5	5	3	3		3	5	5
6	**CassandraServer_java**	8	8	7	7	4	(6)				21	24	43			
7	**AlterKeyspaceStatement_java**	7	7	8	10	7		(7)	5	5			3	3	5	5
8	**DropTriggerStatement_java**	6	6	6	5	5		5	(8)	12				4	5	5
9	**CreateTriggerStatement_java**	6	6	6	5	5		5	12	(9)				4	5	5
10	**SelectStatement_java**	5	12	12	4	3	21				(10)	48	8			3
11	**ModificationStatement_java**	5	7	9	4	3	24				48	(11)				3
12	**StorageService_java**	5		3			43	3			8	0	(12)			
13	**AlterTypeStatement_java**	5	4	6	3	3		3	4	4				(13)	3	4
14	**DropTableStatement_java**	5	5	5	5	5		5	5	5				3	(14)	5
15	**CreateTableStatement_java**	5	6	9	6	5		5	5	5	3	3		4	5	(15)

Each number indicates the co-changes between two files

design rule. They violate the Dependency Inversion Principle [18] since a client should depend on abstractions, not on concretions.

Description: We consider an inheritance hierarchy to be problematic if it falls into one of the following two cases:

1) Given an inheritance hierarchy containing one parent file, f_{parent}, and one or more children, F_{child}, there exists a child file f_i satisfying $depend(f_{parent}, f_i)$
2) Given an inheritance hierarchy containing one parent file, f_{parent}, and one or more children, F_{child}, there exists a client f_j of the hierarchy, that depends on both the parent and one or more of its children.

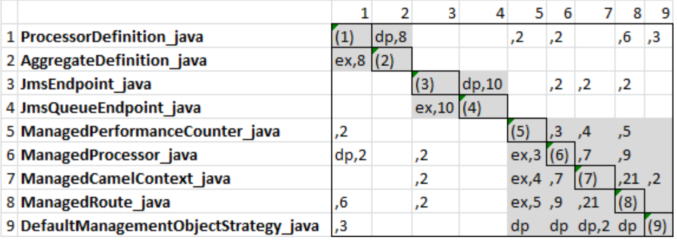
Formalization:

$$\exists f_{parent}, F_{child} \in F \wedge \exists f_i \in F_{child} \mid depend(f_{parent}, f_i) \vee \\ [\exists f_j \in F \mid depend(f_j, f_{parent}) \wedge \exists f_i \in F_{child} \mid depends(f_j, f_i)] \tag{3}$$

where $i, j \in [1, 2, 3, ..., n]$, $f_j \notin F_{child}$ and $f_j \neq f_{parent}$.

Figure 5 presents several instances of *Unhealthy Inheritance Hierarchy*: 1) the parent file, *ProcessorDefinition_java* depends on its child file *AggregateDefinition_java*; 2) the parent file, *JmsEndpoint_java* depends on its child file *JmsQueueEndpoint_java*; 3) the client file *DefaultManagementObjectStrategy_java* depends on the parent file *ManagedPerformanceCounter_java* and all of its children.

Fig. 5: Instances of Unhealthy Inheritance Hierarchy architecture



		1	2	3	4	5	6	7	8	9
1	**ProcessorDefinition_java**	(1)	dp,8			,2	,2		,6	,3
2	**AggregateDefinition_java**	ex,8	(2)							
3	**JmsEndpoint_java**			(3)	dp,10		,2	,2	,2	
4	**JmsQueueEndpoint_java**			ex,10	(4)					
5	**ManagedPerformanceCounter_java**	,2				(5)	,3	,4	,5	
6	**ManagedProcessor_java**	dp,2		,2		ex,3	(6)	,7	,9	
7	**ManagedCamelContext_java**			,2		ex,4	,7	(7)	,21	,2
8	**ManagedRoute_java**	,6		,2		ex,5	,9	,21	(8)	
9	**DefaultManagementObjectStrategy_java**	,3				dp	dp	dp,2	dp	(9)

dp: depend, ex: extend

4. Crossing (CRS).

Rationale: If a file has both a large number of dependents and depends on a large number of other files, i.e., with both high fan-in and high fan-out, it is unlikely that this file follows Single Responsibility Principle [18]. We observe that if such a file also changes frequently with its dependents and the files it dependents on, it is often the center of error- and change-proneness.

Description: If a file is changed frequently with its dependents and the files that it depends on, then we consider these files to follow a *Crossing* anti-pattern (CRS).

Formalization:

$cochange_{thr}$: the co-change frequency threshold. If two files change together more than the threshold, they are considered to be evolutionarily coupled.

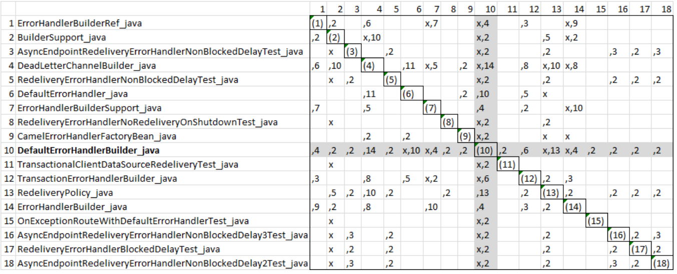
$crossing_{thr}$: the threshold of fan-in and fan-out of a file. If the numbers of dependents and dependees of the file are both larger than the threshold, we consider this group of files as a candidate Crossing.

$$\begin{aligned} &\exists f_i, f_j, f_c \in F \mid (|SRelation(f_i, f_1)| > crossing_{thr} \wedge \\ &|\#cochange(f_i, f_c) > cochange_{thr}| > crossing_{thr}) \\ &\wedge (|SRelation(f_c, f_j)| > crossing_{thr} \wedge \\ &|\#cochange(f_c, f_j) > cochange_{thr}| > crossing_{thr}) \end{aligned} \tag{4}$$

where $i, j \in \{2, 3, 4, ..., n\}$, n is the number of files in a project's DSM. $cochange_{thr}$ and $crossing_{thr}$ are configurable thresholds. In a DSM, f_c will show up at the center of a cross shape, and we call such a file as the *center file* of a crossing instance.

Figure 6 shows an instance of Crossing. We can see that the center file, *DefaultErrorHandlerBuilder_java*, was changed frequently with its dependents and dependees in the revision history.

Fig. 6: An example of Crossing



		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	ErrorHandlerBuilderRef_java	(1)	,2		,6			x,7			x,4		,3		x,9				
2	BuilderSupport_java	,2	(2)		x,10						x,2			,5	x,2				
3	AsyncEndpointRedeliveryErrorHandlerNonBlockedDelayTest_java		x	(3)		,2					x,2			,2			,3	,2	,3
4	DeadLetterChannelBuilder_java	,6	,10		(4)		,11	x,5		,2	x,14		,8	x,10	x,8				
5	RedeliveryErrorHandlerNonBlockedDelayTest_java		x	,2		(5)					x,2			,2			,2	,2	,2
6	DefaultErrorHandler_java				,11		(6)			,2	,10		,5	x					
7	ErrorHandlerBuilderSupport_java	,7			,5			(7)			,4		,2		x,10				
8	RedeliveryErrorHandlerNoRedeliveryOnShutdownTest_java		x						(8)		x,2			,2					
9	CamelErrorHandlerFactoryBean_java				,2		,2			(9)	x,2			x	x				
10	**DefaultErrorHandlerBuilder_java**	,4	,2	,2	,14	,2	x,10	x,4	,2	,2	(10)	,2	,6	x,13	x,4	,2	,2	,2	,2
11	TransactionalClientDataSourceRedeliveryTest_java		x								x,2	(11)							
12	TransactionErrorHandlerBuilder_java	,3			,8		,5	x,2			x,6		(12)	,2	,3				
13	RedeliveryPolicy_java		,5	,2	,10	,2			,2		,13		,2	(13)	,2		,2	,2	,2
14	ErrorHandlerBuilder_java	,9	,2		,8			,10			,4		,3	,2	(14)				
15	OnExceptionRouteWithDefaultErrorHandlerTest_java		x								x,2					(15)			
16	AsyncEndpointRedeliveryErrorHandlerNonBlockedDelay3Test_java		x	,3		,2					x,2			,2			(16)	,2	,3
17	RedeliveryErrorHandlerBlockedDelayTest_java		x	,2		,2					x,2			,2			,2	(17)	,2
18	AsyncEndpointRedeliveryErrorHandlerNonBlockedDelay2Test_java		x	,3		,2					x,2			,2			,3	,2	(18)

x indicates structural dependencies, such as extend, depend, etc.

5. Clique (CLQ).

Rationale: It is widely accepted the cyclical dependencies should be avoid. In our prior work [1], we proposed *Cross-Module Cycle* to detect dependency cycles among source files. In order to reduce the number of instances the user has to examine, in this paper, we define *Clique* as a set of files whose structural relations form a strongly connected graph, so that changes to any files can be propagated to any other files within the group.

Description: If there is a subset of files that form a strongly connected component based on their structural relations, we consider this file group as a Clique instance.

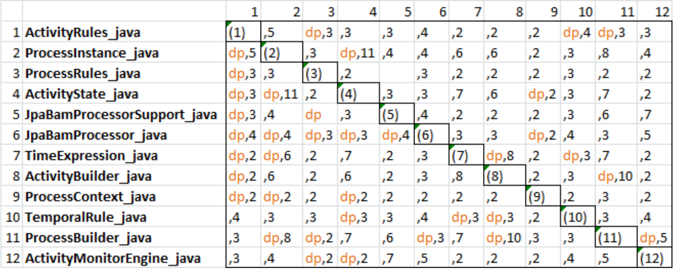
Formalization:

$$\begin{aligned} &\forall f_i \in F_{cq}, \forall f_j \in F_{cq}, i \neq j \mid \\ &[depend(f_i, f_j) \wedge depend(f_j, f_i)] \cup [\exists f_1, ...f_n \in F_{cq}, n \geq 1 \mid \\ &depend(f_i, f_1) \wedge depend(f_1, f_2) \wedge \cdots \wedge depend(f_n, f_j)] \end{aligned} \tag{5}$$

where $i, j \notin \{1, 2, 3, ..., n\}$, F_{cq} is the set of files in a clique.

Figure 7 shows an instance of Clique. Files in this example are highly coupled with each other through multiple dependency cycles, such as, *ActivityRules_java* ↔ *ProcessRules_java*, *ActivityRules_java* → *TimeExpression_java* → *TemporalRule_java* → *ActivityRules_java*, etc.

Fig. 7: An example of Clique



		1	2	3	4	5	6	7	8	9	10	11	12
1	ActivityRules_java	(1)	,5	dp,3	,3	,3	,4	,2	,2	,2	dp,4	dp,3	,3
2	ProcessInstance_java	dp,5	(2)	,3	dp,11	,4	,4	,6	,6	,2	,3	,8	,4
3	ProcessRules_java	dp,3	,3	(3)	,2		,3	,2	,2	,2	,3	,2	,2
4	ActivityState_java	dp,3	dp,11	,2	(4)	,3	,3	,7	,6	dp,2	,3	,7	,2
5	JpaBamProcessorSupport_java	dp,3	,4	dp	,3	(5)	,4	,2	,2	,2	,3	,6	,7
6	JpaBamProcessor_java	dp,4	dp,4	dp,3	dp,3	dp,4	(6)	,3	,3	dp,2	,4	,3	,5
7	TimeExpression_java	dp,2	dp,6	,2	,7	,2	,3	(7)	dp,8	,2	dp,3	,7	,2
8	ActivityBuilder_java	dp,2	,6	,2	,6	,2	,3	,8	(8)	,2	,3	dp,10	,2
9	ProcessContext_java	dp,2	dp,2	,2	dp,2	,2	,2	,2	,2	(9)	,2	,3	,2
10	TemporalRule_java	,4	,3	,3	dp,3	,3	,4	dp,3	dp,3	,2	(10)	,3	,4
11	ProcessBuilder_java	,3	dp,8	dp,2	,7	,6	dp,3	,7	dp,10	,3	,3	(11)	dp,5
12	ActivityMonitorEngine_java	,3	,4	dp,2	dp,2	,7	,5	,2	,2	,2	,4	,5	(12)

dp: depend

6. Package Cycle (PKC).

Rationale: Ideally, the package structure of a software system should form a hierarchical structure [20]. As with the Clique anti-pattern, a cycle among *packages* reduces the understandability and maintainability of a system.

Description: Given two packages P_a, P_b in the DSM, there exists a file f_1 in P_a and a file f_2 in P_b. Given another file f_j in P_b and f_i in P_a, if $depend(f_1, f_j)$ and $depend(f_2, f_i)$, then we consider that these two packages create a *Package Cycle*, that is, a cycle of dependencies between the *packages*.

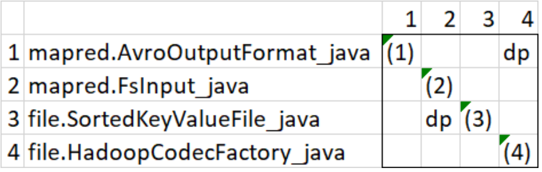
Formalization:

$$\exists f_1, f_i \in P_a \land \exists f_2, f_j \in P_b \mid depend(f_1, f_j) \land depend(f_2, f_i) \tag{6}$$

where P_a, P_b are the packages of the system, $i, j \in [1, 2, 3, 4, ..., n]$, n is the number of files in the system.

Figure 7 shows an instance of Package Cycle, in which *AvroOutputFormat_java* in package *mapred* depends on *HadoopCodecFactory_java* in package *file*, and *SortedKeyValueFile_java* in package *file* depends on *FsInput_java* in package *mapred*, forming a dependency cycle between package *mapred* and package *file*.

Fig. 8: An example of Package Cycle



		1	2	3	4
1	mapred.AvroOutputFormat_java	(1)			dp
2	mapred.FsInput_java		(2)		
3	file.SortedKeyValueFile_java		dp	(3)	
4	file.HadoopCodecFactory_java				(4)

dp: depend

4 Tool Support

Figure 9 depicts the framework for the detection of Architecture anti-pattern instances, with the following steps:

First, we collected two types of data from a project repository: 1) the source code of a project snapshot that will be processed using a commercial reverse-engineering tool, Understand[7], to generate a XML report of file structural dependencies. 2) A specified period of revision history of the project, extracted from Git or SVN repositories.

Second, given the XML file dependency report and selected revision history as inputs, the *DSM Generator* automatically generates SDSM (Structural DSM) and *HDSM* (History DSM) files. The *SDSM* file contains structural relations among files, and the HDSM file contains their pairwise co-change information.

Third, given the *SDSM* and *HDSM* files, our *Architecture Anti-pattern Detector* automatically identifies all architecture anti-pattern instances, captures the involved files, and outputs a summary of the detection results. Each instance will be exported as a DSM presenting how the involved files are connected structurally or evolutionarily.

These DSMs can be viewed using DV8[8], an architecture analysis tool that has been used by many industrial practitioners[9] and academic researchers [21]. The architecture anti-pattern detector has been integrated into DV8.

Tool Complexity The run-time complexity of our tool chain is now discussed, step by step:

1. Downloading revision history and generating a file dependency report using third-party tools: a) We download each projects revision history from GitHub or SVN, and this step typically takes just a few minutes. b) Using source code as input, we the used Understand reverse-engineering tool to generate a report of file dependencies; the time expense of this step is related to the size of a project. For Apache Camel, which is the largest project studied in this work, this step completes in 10 minutes on an average PC (Memory: 16GB, Processor: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz).

2. Using our DSM generator to automatically generate the SDSM and HDSM: a) the time complexity of SDSM generation is $O(V + E)$, where V is the number of files in a project, E is the number of structural dependencies among these files, that is, the number of marked cells in this project's SDSM; b) the time complexity of HDSM generation is O(m*n), where m is the number of commits in the studied revision history, n is the number of files in a project.

3. Given SDSM and HDSM files, detecting architecture anti-patterns: the worst-case time complexity of detection is $O(n^3)$, where n is the number of files in a project. Using Apache Avro (Size: 301 files), Hadoop (Size: 4,519 files), and Camel (Size: 11,732 files) as examples, this step completes in 4, 17, and 76 seconds respectively, on an average PC (Memory: 16GB, Processor: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz).

5 Evaluation

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

5.1 Research Questions

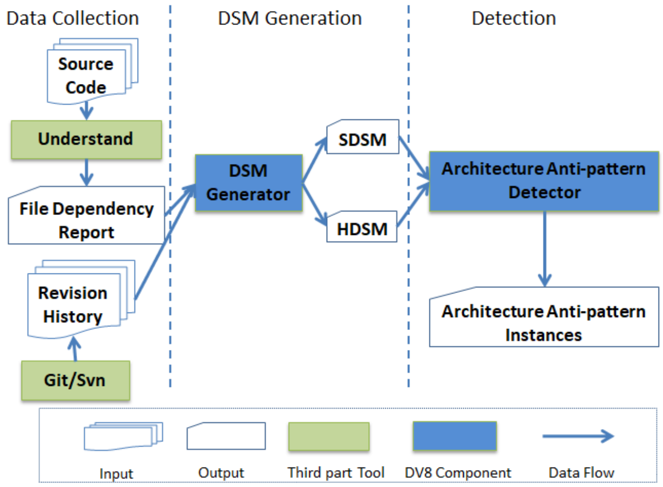
To evaluate whether the 6 architecture anti-patterns have significant impact on error-proneness and change-proneness, and hence deserve special attention when mak-

[7] https://scitools.com/
[8] https://www.archdia.net/products-and-services/
[9] https://www.archdia.net

TABLE 1: Researched Projects

Subjects	Release	#Commits	#Bugs	History Length	#Files	LOC	Description
Avro	1.7.7	1,278	495	63 months	301	178K	Serialization system
Camel	2.15.5	21,655	1,931	113 months	11,732	934K	Integration framework
Cassandra	3.1	19,333	3,270	89 months	1,765	331k	Distributed database
CXF	3.1.4	11,160	2,278	96 months	5,960	708K	Services framework
Derby	10.12.1.1	8,058	3,214	78 months	2,770	760K	Relational database
Hadoop	2.7.2	12,506	1,785	86 months	4,519	1.6M	Tool for distributed Big Data processor
HBase	0.98.16.1	11,329	4,848	105 months	1,556	744K	Hadoop database
Ivy	2.4.0	2,518	641	106 months	613	309K	Tool for dependency management
Mahout	0.11.1	3,377	567	94 months	1,158	125K	Scalable machine learning libraries
OpenJPA	2.4.1	4,729	1,575	116 months	3,579	494k	Java persistence project
PDFBox	1.8.10	4,337	1,203	95 months	738	122k	Library for manipulating PDF documents
Pig	0.15.0	2,754	1,072	75 months	1,150	369K	Platform for analyzing large data sets
Tika	1.10	2,717	735	102 months	718	86K	Content analyzer
Wicket	7.1.0	18,963	2,786	132 months	3,090	288K	Java web application framework
ZooKeeper	3.5.1	1,364	707	86 months	402	134K	Tool provides centralized services
Comm_1	-	360	186	25 months	491	198K	Tool for HMI engineering
Comm_2	-	1,668	560	41 months	1541	422K	Platform for engineering tool
Comm_3	-	2,568	482	72 months	6,948	988K	Tool for client/server-based integration
Comm_4	-	39,074	5,268	72 months	7,754	834K	Tool for harmony controller

Fig. 9: Framework of Architecture Anti-pattern Detection



ing maintenance and refactoring decisions, we investigate the following research questions (RQs):

RQ1. *Do the files involved in these anti-patterns consume significantly more maintenance effort than other files?* This question examines whether files participated in these anti-patterns are truly error-prone and/or change-prone. A positive answer would suggest that when we make changes or fix bugs in these files, we have to consider their architectural connections, which could be problematic and have significant impact on files' maintenance.

RQ2. *If a file participates in more anti-patterns, then is it more error-prone/change-prone?* The answer to this question will indicate how strongly these anti-patterns impact error-proneness and change-proneness of a file. If the more anti-patterns a file participates in, the more error-prone/change-prone it is, then files involved in multiple anti-patterns should have higher priority for refactoring, and the refactoring for these files will be more complicated.

RQ3. *Do different types of anti-patterns have different impacts on error-proneness and change-proneness?* The answer to this question will advance our understanding regarding to if and how these anti-patterns differ from each other in terms of their impact on maintainability.

5.2 Subjects

We chose a total of 19 projects as our experimental subjects, including 15 Apache open source projects and 4 commercial projects from our industrial collaborators. These projects differ in size, age, domain and other project characteristics. Table 1 shows their basic demographic facts.

The first column indicates the version of each project we selected. The column "$\#Commits$" shows the number of revisions we examined; this data is calculated from a project's revision history, from the beginning to the selected version. All revision histories were extracted from Git[10] or SVN version control systems. "$\#Bugs$" indicates the number of bug reports, as recorded in a project's issue tracking system. For open source projects, bug reports are extracted from their JIRA[11] archives; for commercial projects, each collaborator provided us with bug reports from their issue tracking system. "$\#HistoryLength$" shows the number of months of each project's revision history we studied, from its beginning to the selected release date. The fifth and sixth columns present the size of each project, measured by the number of files and LOC. The last column describes the domain of each project.

For each project, we selected its latest version as our subject. Using a project's source code only, our tool chain can detect files involve in Unhealthy Hierarchy, Clique or Package Cycle. If the project's revision history is also available, we can detect the other three anti-patterns. The user can select any period of revision history as input to the tool. For evaluation purposes, we studied all the history from the beginning to the selected version.

5.3 Evaluation Methods

For our analyses, we applied our detection tool on the nineteen projects. For each project, our tool detects architecture anti-patterns (and the associated set of files). We then investigated the error-proneness and change-proneness of the involved files.

[10] https://git-scm.com/
[11] https://www.atlassian.com/software/jira

To quantitatively estimate the maintenance effort spent on each file, we adopted the following history measures:

- *Bug Frequency* (BF): the number of times a file participated in bug-fixing commits. The higher the score, the more error-prone the file is. We extract bug data from issue tracking systems and revision history. We used the pattern matching method in [28] to locate bug-fixing commits. A commit was considered as a bug fix if a bug ticket ID recorded in the issue tracking system is identified in its change message.
- *Bug Churn* (BC): the number of added and deleted lines of code (*"churn"*) to a file by bug-fixing commits.
- Change Frequency (CF): the number of times a file was committed in the revision history. The higher the value, the more change-prone it is.
- Change Churn (CC): the number of changed LOC in a file committed for any issues.

These measures manifest how frequently a file was changed and how costly (in terms of churn) the file was during maintenance. These measures were calculated by: 1) mining a project's revision history from its beginning to the selected release date; 2) calculating the measure values of each file based on the corresponding definitions. Bug frequency and bug churn were calculated by mining both revision history and the bug reports extracted from a project's issue tracking system.

Table 2 presents a summary of the detected anti-patterns for each project. Some files are involved in multiple anti-patterns, so the total *'$\#Files$'* shows the total number of *distinct* files involved in each anti-pattern. The following % indicates the percentage to the total number of files in a project. The first observation to make is that these anti-patterns occurred in *all* the studied projects, and they are indeed ubiquitous. Every project had at least one instance (ins) of each anti-pattern, and in some cases there were hundreds of instances and thousands of files implicated. Architecture anti-patterns are real and frequently recurring.

5.4 Quantitative Analysis

Given the measures of maintenance effort, Bug Frequency (BF), Bug Churn (BC), Change Frequency (CF), Change Churn (CC), we now present and analyze our results to answer these research questions. For the rest of the paper, we call files participated in anti-patterns as *infected files*, and other files as *non-infected files*.

RQ1. *Do infected files consume significantly more maintenance effort than non-infected files?* To answer this question, we need to validate whether infected files were changed more frequently and consumed substantially more effort. For each project, we calculated scores of the four measures for each file, and for each measure, we calculate its average value for infected files as $avg\_af\_measure_i$, and the average value for non-infected files as $avg\_non\_af\_measure_i$.

Figure 10 (a) - (d) presents the average values of all measures. The *blue* bars present results for infected files. The *red* bars present the values for non-infected files. For each measure we also calculated the relative increases between infected and non-infected files:

$$measure_i\_inc = \frac{avg\_af\_measure_i - avg\_non\_af\_measure_i}{avg\_non\_af\_measure_i}$$

Table 3 shows the differences in average values for each project. These results show that the average values of all measures of infected files are larger than the average values of non-infected files. Using the *Camel* project as example (Figure 10c), we can observe that its *avg_af_CF* is 7.68, but its *avg_non_af_CF* is 1.04. According to Table 3, its CF_inc is calculated as 640%, meaning that its infected files were changed more than 6 times as much as non-infected files. The greatest increase is 11,968%, Bug Frequency (BF) of *Cassandra*, meaning that its infected files were changed for bug-fixes about 120 times more than non-infected files. In Figure 10 (a), we can see that the avg_af_BF of *Cassandra* is 3.54, but its $avg\_non\_af\_BF$ is only 0.03. All of these results indicate that infected files are far more error-prone and change-prone than non-infected files.

To substantiate this claim, we employ the *Wilcoxon signed-rank test*, a non-parametric statistical hypothesis test for comparing two related samples, to test whether the population of $avg\_af\_measure_i$ is significantly larger than the population of $avg\_non\_af\_measure_i$ over the 19 projects. We defined the hypotheses as follows:

Null Hypothesis: H_0, *the population of* $avg\_af\_measure_i$ *is not significantly larger than the population of* $avg\_non\_af\_measure_i$.

Alternative Hypothesis: H_1, *the population of* $avg\_af\_measure_i$ *is significantly larger than the population of* $avg\_non\_af\_measure_i$.

For all measures, the p-values of tests are less than 0.01 (From BF to CC, the p-values are 3.5E-6, 2.8E-6, 4E-6 and 1.2E-5 respectively), so that H_1 is accepted for all the tests. The results indicate that the differences between $avg\_af\_measure_i$ and $avg\_non\_af\_measure_i$ across all 19 projects are statistically significant. In addition, we leveraged the *Hedges'g* [29] to indicate the effect size [30], [31] to test the difference between two populations. For all measures, the effect sizes are large than 0.8 (from BF to CC, the effect sizes are 1.8, 1.2, 2.7 and 1.8 respectively). Based on the rule of thumb interpretation [32], [33] of the effect size, we can conclude that there exists significant differences between $avg\_af\_measure_i$ and $avg\_non\_af\_measure_i$.

Our results demonstrate that the four measures are consistently and significantly greater for infected files. Therefore, we have strong evidence to believe that files participating in anti-patterns will be more error-prone and change-prone, consuming much higher maintenance efforts.

RQ2. *If a file is involved in* greater numbers *of architecture anti-patterns, then is it more error-prone/change-prone?* To answer this question, we need to investigate whether the files participating in *multiple* architecture anti-patterns incur greater maintenance costs.

For each file in each project, we first calculated its four measures, then calculated how many architecture anti-patterns the file participates in. After that, we categorized the files based on the number of participated anti-patterns, and calculated the average measures for each category.

TABLE 2: Identified architecture anti-patterns for all 19 projects

	Avro		Camel		Cassandra		CXF		Derby		Hadoop		HBase	
Type	#Ins	#Files (%)	#Ins	#Files (%)	#Ins	#Files (%)	#Ins	#Files (%)	#Ins	#Files (%)	#Ins	#Files (%)	#Ins	#Files (%)
UIF	5	76 (25%)	23	995 (8%)	55	750 (42%)	12	408 (7%)	48	768 (28%)	30	780 (17%)	61	751 (48%)
MVG	34	141 (47%)	1,139	5,524 (47%)	134	974 (55%)	505	2,373 (52%)	282	1,440 (25%)	339	1,665 (37%)	143	909 (58%)
UIH	17	57 (19%)	149	1,085 (9%)	86	343 (19%)	196	887 (15%)	211	680 (25%)	271	987 (22%)	94	299 (19%)
CRS	7	56 (19%)	145	1,360 (12%)	83	643 (36%)	70	659 (11%)	117	742 (27%)	140	1,039 (23%)	85	601 (39%)
CLQ	9	41 (14%)	156	1,064 (9%)	7	562 (32%)	71	394 (7%)	54	766 (28%)	93	760 (17%)	26	306 (20%)
PKC	17	113 (38%)	179	2,822 (24%)	279	1,332 (75%)	209	1,548 (26%)	100	985 (36%)	305	2,193 (49%)	119	1,050 (67%)
	Ivy		Mahout		OpenJPA		PDFBox		Pig		Tika		Wicket	
	#Ins	#Files (%)	#Ins	#Files (%)	#Ins	#Files (%)	#Ins	#Files (%)	#Ins	#Files (%)	#Ins	#Files (%)	#Ins	#Files (%)
UIF	17	175 (29%)	4	115 (10%)	26	378 (11%)	20	299 (25%)	14	180 (16%)	6	71 (10%)	19	543 (18%)
MVG	62	263 (43%)	126	650 (56%)	184	1,021 (29%)	88	506 (42%)	83	430 (37%)	84	333 (46%)	319	1,386 (45%)
UIH	33	131 (21%)	34	152 (213%)	136	508 (14%)	33	122 (10%)	49	177 (15%)	24	219 (31%)	84	394 (13%)
CRS	23	161 (26%)	21	185 (16%)	53	405 (11%)	26	237 (20%)	19	141 (12%)	9	83 (12%)	44	491 (16%)
CLQ	7	153 (25%)	13	41 (4%)	77	651 (18%)	10	222 (18%)	10	415 (36%)	14	48 (7%)	28	251 (8%)
PKC	97	354 (58%)	34	260 (22%)	57	843 (24%)	60	389 (32%)	118	758 (66%)	26	205 (29%)	312	1,593 (52%)
	ZooKeeper		Comm_1		Comm_2		Comm_3		Comm_4					
	#Ins	#Files (%)	#Ins	#Files (%)	#Ins	#Files (%)	#Ins	#Files (%)	#Ins	#Files (%)				
UIF	9	128 (32%)	1	16 (3%)	58	652 (42%)	19	197 (3%)	17	208 (3%)				
MVG	26	158 (39%)	19	53 (11%)	87	858 (56%)	31	287 (4%)	140	676 (9%)				
UIH	20	86 (21%)	9	46 (9%)	68	257 (17%)	83	328 (5%)	73	263 (3%)				
CRS	19	108 (27%)	1	6 (1%)	81	368 (24%)	26	153 (2%)	68	414 (5%)				
CLQ	7	66 (16%)	21	89 (18%)	26	322 (21%)	116	1,379 (20%)	140	1,684 (22%)				
PKC	16	259 (64%)	67	173 (35%)	175	499 (32%)	140	2,825 (41%)	253	3,194 (41%)				

TABLE 3: Average measure increases for all projects

Project	BF_inc	BC_inc	CF_inc	CC_inc
Avro	1,252%	1,096%	840%	714%
Camel	759%	296%	640%	379%
Cassandra	11,968%	5,170%	4,566%	2,533%
CXF	849%	441%	750%	547%
Derby	740%	417%	1,180%	458%
Hadoop	219%	429%	1,874%	1,416%
Hbase	3,226%	3,279%	4,357%	1,712%
Ivy	2,250%	1,018%	1,357%	898%
Mahout	903%	897%	757%	540%
OpenJPA	117%	14%	334%	332%
PDFBox	4,152%	4,375%	3,093%	1,811%
Pig	1,594%	1,864%	1,439%	661%
Tika	765%	563%	729%	458%
Wicket	1,029%	744%	1,117%	1,001%
ZooKeeper	2,330%	1,571%	1,850%	1,125%
Comm_1	539%	1,540%	400%	1,197%
Comm_2	2,565%	11,424%	1,773%	6,222%
Comm_3	476%	1,307%	360%	1,188%
Comm_4	606%	869%	449%	958%

***BF, BC, CF and CC_inc**: the increases of average measure values*

Finally, we analyzed the relations between the number of anti-patterns and the average measure values. In this way, we were able to investigate whether there is a correlation between the number of architecture anti-patterns a file participates in and its error-proneness and change-proneness.

Table 4 presents the results relevant to this research question. Column $\#AF$ indicates the number of architecture anti-patterns that a file participates in. "$a.measure_i$" means the average measure of files involved in a particular number of anti-patterns. The table shows that the more anti-patterns a file is involved in, the more maintenance effort it has incurred. Consider the *a.BF* of Apache Avro[12] as an example. The files involved in six anti-patterns exhibit the greatest average bug frequency—12.0—which is substantially higher than the average bug frequency of files which are only involved in five anti-patterns, where the average value is 4.4. The files involved in four anti-patterns have an even smaller average value of 3.5. The files not involved in any anti-patterns have the smallest average bug frequency value, 0.3.

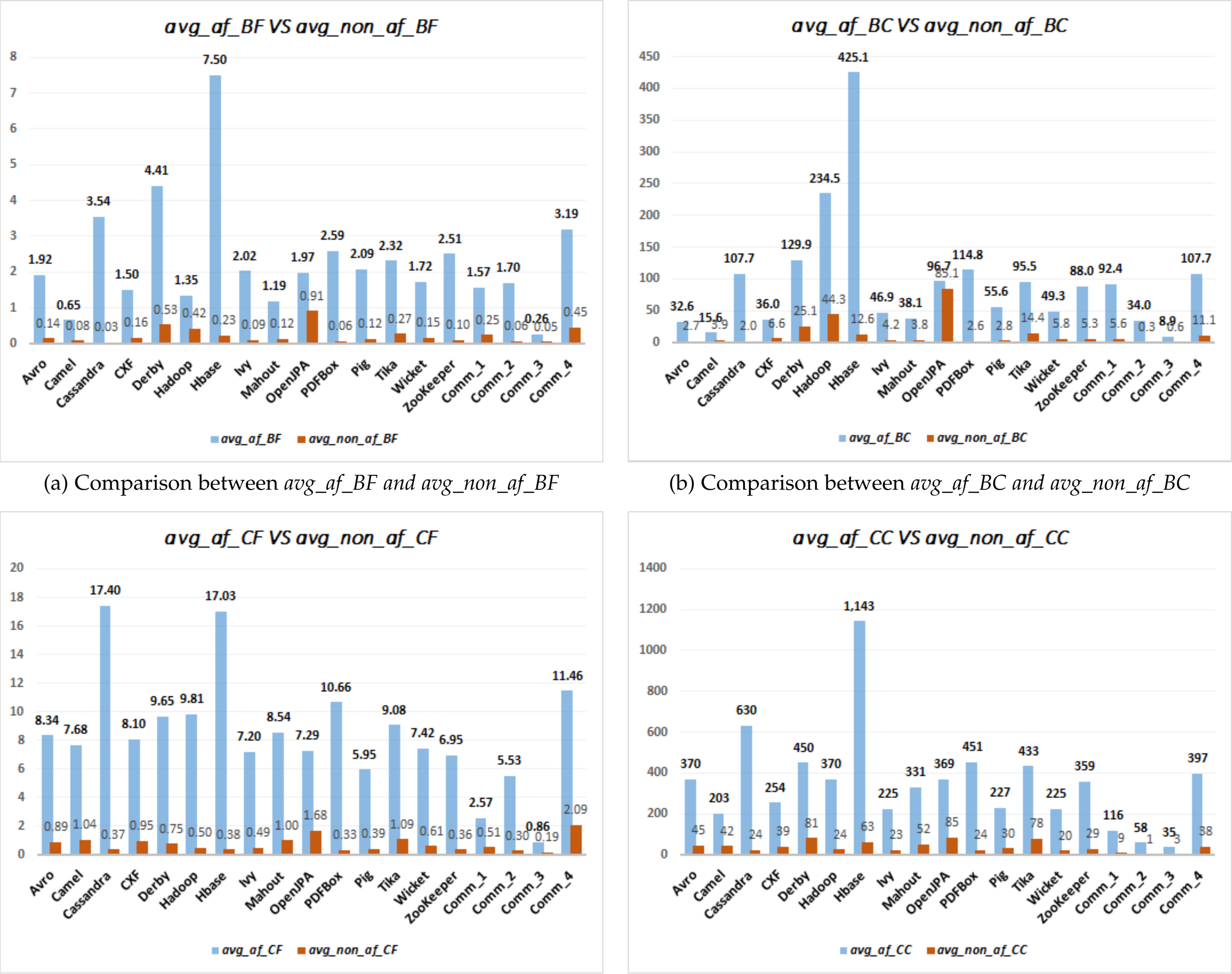
To answer this question more rigorously, we conducted a *Pearson Correlation Analysis*—a measure of the strength of correlation between two sets of variables—to test the relations between the number of anti-patterns a file participates in ($\#AF$) and the average values of its four measures. Using *Pearson Analysis*, we investigated whether each measure would increase as the number of involved anti-patterns increased from 0 to 6. The r row of Table 4 shows the *Pearson Correlation Coefficient* for each measure. The pv row shows the $p-value$ of the correlation analysis, indicating the significance of the correlations. The values of r and the p-values in Table 4 indicate the average measures ($a.measure_i$) are significantly correlated to $\#AF$. That is, the more architecture anti-patterns a file participates in, the more maintenance effort it will incur. Our analysis indicates that *all* of these anti-patterns have a significant impact on the error-proneness and change-proneness of files.

RQ3. *Do different architecture anti-patterns have different impacts on error-proneness and change-proneness?* We first analyze to what extent each anti-pattern influences file error- or change-proneness, and which anti-pattern has the greatest influence, and hence contribute most to technical debt.

For each project, we first calculated all four measures for each file, and the average measures for each anti-pattern, $avg\_archType_j\_measure_i$. After that, we calculated average measures of files not involved in a given anti-pattern as $avg\_non\_archType_j\_measure_i$, and made comparison. This way, we could investigate each architecture anti-pattern independently, and compare the degree to which each affects file's error-proneness and change-proneness. As an example, for *Unstable Interface*, we calculated $avg\_UIF\_measure_i$, and compare it with the measures for files not involved in this anti-pattern, $avg\_non\_UIF\_measure_i$. The comparison reveals whether

[12] http://openjpa.apache.org/

Fig. 10: Histogram of $average\ measures_i$ of files participated in architecture anti-patterns or not



(a) Comparison between *avg_af_BF and avg_non_af_BF*

(b) Comparison between *avg_af_BC and avg_non_af_BC*

(c) Comparison between *avg_af_CF and avg_non_af_CF*

(d) Comparison between *avg_af_CC and avg_non_af_CC*

avg_af_BF, BC, CF and CC: *the average bug frequency, bug churn, change frequency and change churn of infected files;*
avg_non_af_BF, BC, CF and CC: *the average bug frequency, bug churn, change frequency and change churn of non-infected files*

the files affected by $UnstableInterface$ are more error-prone and change-prone, and by how much.

Tables 5-10 present the results of our analysis. Columns 2-5 report the average measures for files involved in a particular anti-pattern. Columns 6-9 report the average measures for files not involved in the same anti-pattern. Columns 10-13 report the differences. From these results, we first observe that almost all the measures increased, over all projects and all anti-patterns, as expected, and revealed by Tables 5 to 10. To explore which anti-pattern has the highest impact, we conduct pairwise comparisons among anti-patterns using *Paired Wilcoxon signed-rank tests*. For each pair of anti-patterns, we defined the hypotheses as follows:

Null Hypothesis: H_0, *the average measures of anti-pattern* i *are not different than the average measures of anti-pattern* j.

Alternative Hypothesis: H_1, *the average measures of anti-pattern* i *are significantly different from the average measures of anti-pattern* j.

where, i and j indicate different anti-patterns, and $i \neq j$.

Pairwise comparisons requires more strict p-values to indicate the significance of each test. We leveraged Benjamini and Hochberg correction [34] to adjust the p-values. This correction helps to control for the fact that small p-values (less than 0.05) can occasionaly happen by chance, which could lead to incorrectly rejecting the true null hypotheses. We reported all p-values of the tests in Table 11. A cell in row x, column y, $cell(x, y)$ presents the p-value from the analysis, which tests whether the average measures of anti-pattern x are significantly different than the average measures of anti-pattern y. P-values less than 0.05 are highlighted.

From Table 11, we can observe that the measures of *Unstable Interface* and *Crossing*, are significantly different from the others. We also calculated the *effect size* using

TABLE 4: Average values of files involved in different numbers of architecture anti-patterns

	Avro				Camel				Cassandra				CXF			
#AF	a.BF	a.BC	a.CF	a.CC	a.BF	a.BC	a.CF	a.CC	a.BF	a.BC	a.CF	a.CC	a.BF	a.BC	a.CF	a.CC
0	0.3	5.7	1.9	95.6	0.1	6.4	1.7	69.1	0.2	11.5	2.1	134.9	0.2	8.8	1.3	52.2
1	0.6	14.1	3.5	166	0.3	9.5	4.4	134	0.3	18.4	2.9	175.3	0.6	18.8	4.1	145.7
2	0.9	22.5	5.5	219.5	0.5	13.5	7.0	193.7	0.6	23.6	5	226.7	1.2	28.8	7.2	239.6
3	1.5	32.4	6.4	251	0.9	18.2	10.0	254.8	1.4	41.3	9.7	397.2	2.4	59.8	13	395.4
4	3.5	49	13.6	764.9	1.8	35.2	16.8	394.2	3.7	95.5	19.6	569.1	4	87.6	19.3	543.5
5	4.4	56.8	19.2	1,012.8	3.6	65.2	33.4	704.3	7.1	272.5	33.4	1,539.2	6.9	135.8	29.7	772.3
6	12.0	169.7	43	1,358.7	6.1	107.8	53.6	1,103.5	17.8	453.5	71.6	2,000.9	15.1	267.5	57	1,553.6
r	0.85	0.83	0.87	0.94	0.91	0.9	0.91	0.92	0.84	0.87	0.87	0.9	0.88	0.9	0.91	0.91
pv	0.02	0.02	0.01	2E-3	0.01	0.01	4E-3	3E-3	0.02	0.01	0.01	0.01	0.01	0.01	4E-3	4E-3
	Derby				Hadoop				HBase				Ivy			
#AF	a.BF	a.BC	a.CF	a.CC	a.BF	a.BC	a.CF	a.CC	a.BF	a.BC	a.CF	a.CC	a.BF	a.BC	a.CF	a.CC
0	1.4	65.3	2	209.6	1.0	103.0	1.2	56.8	1.2	64.8	2.0	325	0.3	16.4	1.9	88
1	2.5	77.6	3.9	273.6	0.9	119.5	2.3	105.8	1.9	231.6	3.6	473.6	0.4	10.5	2.2	71.4
2	3.3	119	6.7	437.8	1.3	192.2	4.8	211.5	2.9	279.2	7.2	910.2	0.9	16.8	3.9	109.1
3	4.6	146.1	10.1	537.6	1.3	176.1	8.2	324	3.9	373.8	10.2	1980.1	2.3	46.8	8	222
4	5.3	132.8	12.1	405.5	1.8	350.9	13.8	502.6	6.7	277.1	14.7	598.9	3.1	81	10.3	358.9
5	7.1	172.5	17.3	561.7	2	463.5	29.3	987.6	14.2	697	31.8	1375.3	4.4	85.3	15.4	411.9
6	10.5	283.4	26.7	952.3	3.5	997.5	61.6	2,214.5	36.6	1,522.2	84.2	3,701.2	10.4	285.6	30.8	1,154.0
r	0.96	0.91	0.96	0.88	0.88	0.86	0.87	0.86	0.82	0.83	0.82	0.75	0.88	0.81	0.9	0.84
pv	5E-4	4E-3	6E-4	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.02	0.05	0.01	0.03	0.01	0.02
	Mahout				OpenJPA				PDFBox				Pig			
#AF	a.BF	a.BC	a.CF	a.CC	a.BF	a.BC	a.CF	a.CC	a.BF	a.BC	a.CF	a.CC	a.BF	a.BC	a.CF	a.CC
0	0.2	8	2.1	108.8	0.8	76.7	1.5	77	0.5	20.5	2.7	188.9	0.4	9.1	1.2	96
1	0.7	21.1	5.4	216.8	0.8	65.8	2.9	145.6	0.8	57.6	4	215.4	0.7	19.8	2.4	126.8
2	1	35.6	8.7	341.1	0.9	72.9	4.2	284.4	1.7	66.8	7.9	244.9	1	25	3.2	134.6
3	1.8	47.2	13.4	496.9	1.4	64.6	6.7	331.2	1.9	74.9	8	332.9	1.5	51.3	4.7	169.1
4	3	106.2	18.7	726	3	160.2	12.9	701.5	2.8	117.4	12.9	531.4	4.1	112.3	10.6	382.7
5	5.6	195.4	30.6	988	4.5	158.9	16.6	842.5	6.4	257	24.9	1050.4	5	120.5	12.6	388.8
6	6.9	330.9	25.1	919	12.1	332.3	36.5	1527.2	11.8	564.2	38.3	1,893.8	11.8	301.1	34	1,171.1
r	0.95	0.91	0.95	0.98	0.82	0.82	0.89	0.93	0.87	0.84	0.91	0.87	0.87	0.88	0.85	0.81
pv	1E-3	5E-3	1E-3	2E-4	0.02	0.02	0.01	2E-3	0.01	0.02	4E-3	0.01	0.01	0.01	0.02	0.03
	Tika				Wicket				ZooKeeper				Comm_1			
#AF	a.BF	a.BC	a.CF	a.CC	a.BF	a.BC	a.CF	a.CC	a.BF	a.BC	a.CF	a.CC	a.BF	a.BC	a.CF	a.CC
0	0.6	34.8	2.6	188	0.4	17.1	1.8	60	0.4	20.5	1.4	114.3	0.2	4.9	0.4	7.7
1	1	61.8	4.3	220.9	0.7	24.3	3.2	102.1	0.4	27.6	1.5	107	0.6	37	1.1	46.9
2	2.8	117.5	10.2	482	1.2	39.1	6	203	0.5	24	2.4	229.5	1	60.7	1.5	70.8
3	4.2	145.6	16.3	599.5	2.3	58.8	9.5	328.8	1.4	45.8	4.8	331.5	4.5	228.6	7.5	270.3
4	4.2	109.2	14.9	1001.9	2.9	64.8	12.5	300.6	2.6	105	8.2	412.1	14.9	949.1	22.8	1,251.9
5	6.6	204.9	24.2	1,295.8	6.1	190	22.4	608	7.4	189.5	18.7	639.3	8.3	289.8	13.3	386.8
6	6.2	136.2	41.8	1,472	16.6	382.3	66	1,911	15.7	571.2	37.2	1,939.5				
r	0.97	0.8	0.93	0.98	0.82	0.84	0.81	0.79	0.84	0.8	0.86	0.8	0.80	0.65	0.82	0.65
pv	3E-4	0.03	0.002	7E-5	0.02	0.02	0.03	0.03	0.02	0.03	0.01	0.03	0.05	0.16	0.05	0.16
	Comm_2				Comm_3				Comm_4							
#AF	a.BF	a.BC	a.CF	a.CC	a.BF	a.BC	a.CF	a.CC	a.BF	a.BC	a.CF	a.CC				
0	0.2	0.7	0.7	2.2	0	0.6	0.2	2.7	0.4	10.9	2	36.7				
1	0.4	1.7	2.2	5.1	0.1	4.1	0.4	11	1.4	43.3	5.1	144.7				
2	1	11	4.1	26.9	0.2	2.5	0.6	13.4	2.7	91.3	9.9	338.9				
3	3.3	66.5	9.1	112.9	1	36.2	2.8	139.9	5.3	190.2	20.4	667.5				
4	3.7	74.3	9.2	115	1.6	36	6	308.2	11.3	303.5	43.1	1,411.1				
5	2.1	56.2	7.7	91.3	2.4	85.9	8.2	411.1	25.4	1,058.2	81.6	3,759				
6	1.8	77.7	5.8	117.9	4.2	231	11.5	803	43.5	1,985.7	116.8	6,505.5				
r	0.61	0.89	0.72	0.87	0.93	0.83	0.95	0.92	0.89	0.85	0.93	0.88				
pv	0.14	0.01	0.07	0.01	2E-2	0.02	8E-4	4E-3	7E-4	0.014	2E-3	9E-3				

#AF: *the number of architecture anti-patterns a file is involved in;* ***a.BF, a.BC, a.CF and a.CC:*** *average bug frequency, bug churn, change frequency and change churn of files participating in #AF anti-patterns*

Hedge's [29] method. Based on the rule of thumb interpretation [32], [33] for the effect size, we consider values from 0.2 to 0.5 to indicate a small difference; values from 0.5 to 0.8 indicate a medium difference; and values larger than 0.8 indicate a large difference.

Using each of four history measures, we calculated the effect size by comparing the measure's average of files infected with the anti-pattern i versus files infected with the anti-pattern j, where $i \neq j$. Tables 12–15 present the results of all pairwise comparisons based on different history measure. We only report the effect sizes that indicate at least a small difference. Considering Table 15 as an example, "0.93" is the effect size between avg_UIF_CC and avg_MVG_CC over all projects, meaning that there exists a large difference (the effect size is larger than 0.8) between the average *change churn* of files involved in *Unstable Interface* and that of files involved in *Modularity Violation Group*. Tables from 11 to 15 indicate that, *Unstable Interface* and *Crossing* have most impact on error-proneness and change-proneness, while *Package Cycle* has smaller impact. In other words, all the architecture anti-patterns we defined contribute to maintenance costs, in terms of bug frequency, change frequency, bug churn and change churn, but *Unstable Interface* and *Crossing* have contributed by far the most.

5.5 Evaluation Summary

In summary, our analysis leads to the following conclusions: First, files infected with architecture anti-patterns we defined have significantly higher error-proneness and change-proneness than files that are not infected. Second, the more anti-patterns a file participates in, the more error-prone and change-prone it is. Third, all anti-patterns contribute to

TABLE 5: Average measures for files involved in, vs. not involved in, Unstable Interface (UIF)

Subjects	$avg\_UIF\_measure_i$				$avg\_non\_UIF\_measure_i$				$measure_i\_inc$			
	BF	BC	CF	CC	BF	BC	CF	CC	BF	BC	CF	CC
Avro	3.8	57.8	15.7	647.4	0.6	12.5	3.1	158.1	563%	362%	413%	310%
Camel	2.5	46.4	22.0	461.0	0.3	8.9	3.9	123.3	856%	420%	468%	274%
Cassandra	6.7	197.0	30.5	1041.8	0.3	16.5	3.7	196.5	1,928%	1,093%	723%	430%
CXF	5.7	121.8	25.4	710.1	0.6	17.2	3.7	127.5	866%	609%	591%	457%
Derby	7.4	226.3	17.8	783.8	2.1	68.0	3.6	229.3	249%	233%	400%	242%
Hadoop	2.2	479.8	27.9	970.9	1.0	135.5	2.9	131.1	116%	254%	865%	641%
Hbase	11.5	596.3	26.4	1685.9	1.8	152.1	3.6	379.9	553%	292%	633%	344%
Ivy	4.5	105.6	14.3	447.6	0.5	14.8	2.8	96.7	732%	615%	405%	363%
Mahout	3.0	102.7	18.2	668.9	0.7	20.2	5.2	214.1	361%	408%	253%	212%
OpenJPA	5.7	185.8	19.3	921.7	0.9	74.4	2.5	132.0	569%	150%	676%	598%
PDFBox	4.3	179.7	16.8	700.6	1.0	53.0	5.0	232.4	324%	239%	238%	202%
Pig	6.1	152.0	16.7	549.4	0.9	24.6	2.6	130.8	610%	518%	532%	320%
Tika	4.4	142.5	18.4	971.3	1.5	70.6	6.0	294.7	185%	102%	207%	230%
Wicket	4.5	114.8	17.5	465.2	0.7	25.4	3.5	123.0	518%	352%	398%	278%
ZooKeeper	5.5	172.1	14.3	697.9	0.5	28.5	1.9	127.2	1,010%	505%	669%	449%
Comm_1	10.9	583.9	16.6	727.2	0.5	27.4	0.9	35.3	2,074%	2,033%	1,700%	1,960%
Comm_2	2.5	53.7	7.6	89.0	0.3	2.5	1.5	6.7	706%	2,089%	408%	1,234%
Comm_3	2.3	93.7	7.6	416.2	0.1	2.1	0.3	7.4	2,574%	4,288%	2,317%	5,551%
Comm_4	21.5	768.4	69.7	2887.4	1.3	39.2	5.0	141.2	1,619%	1,862%	1,305%	1,945%

$\boldsymbol{avg\_UIF\_measure_i}$: *average measures of files involved in UIF;* $\boldsymbol{avg\_non\_UIF\_measure_i}$: *average measures of files not involved in UIF*

TABLE 6: Average measures for files involved in, vs. not involved in, Crossing (CRS)

Subjects	$avg\_CRS\_measure_i$				$avg\_non\_CRS\_measure_i$				$measure_i\_inc$			
	BF	BC	CF	CC	BF	BC	CF	CC	BF	BC	CF	CC
Avro	4.3	65.5	17.0	752.9	0.7	14.5	3.8	173.9	486%	352%	350%	333%
Camel	2.0	37.0	18.8	423.5	0.2	8.8	3.7	116.3	723%	319%	413%	264%
Cassandra	7.4	219.6	33.6	1173.6	0.6	20.7	4.5	201.5	1,228%	959%	649%	482%
CXF	3.8	76.6	18.6	534.6	0.6	17.8	3.5	121.7	540%	329%	432%	339%
Derby	6.7	172.2	16.2	626.8	2.4	89.9	4.3	293.8	174%	92%	274%	113%
Hadoop	1.9	392.1	22.5	793.1	1.1	136.1	2.6	121.7	76%	188%	757%	552%
Hbase	13.2	651.0	30.2	1844.6	2.2	187.5	4.7	485.1	491%	247%	539%	280%
Ivy	4.7	106.4	15.0	458.7	0.6	17.3	2.9	103.6	658%	516%	410%	343%
Mahout	2.4	80.4	16.6	588.9	0.6	18.5	4.5	196.6	300%	334%	266%	199%
OpenJPA	5.2	181.5	18.7	903.1	0.9	74.0	2.4	127.7	495%	145%	677%	607%
PDFBox	4.6	193.1	18.6	807.8	1.3	62.3	5.6	239.6	261%	210%	234%	237%
Pig	6.4	171.3	17.9	609.1	1.0	26.8	3.0	138.6	521%	539%	497%	340%
Tika	3.8	92.1	16.1	820.4	1.6	75.9	6.0	301.6	140%	21%	167%	172%
Wicket	4.6	115.6	18.3	522.9	0.8	27.0	3.7	118.9	473%	328%	400%	340%
ZooKeeper	6.6	213.5	16.6	728.3	0.4	23.1	1.9	154.8	1,522%	826%	797%	370%
Comm_1	8.2	678.0	14.8	941.8	0.8	37.7	1.3	46.9	985%	1,699%	1,074%	1,908%
Comm_2	2.9	68.4	8.5	109.4	0.7	10.2	2.7	20.2	291%	568%	217%	441%
Comm_3	2.6	91.3	8.1	443.5	0.1	2.8	0.4	9.4	2,584%	3,178%	2,211%	4,619%
Comm_4	13.1	457.8	46.6	1802.2	1.2	36.2	4.4	125.3	1,035%	1,164%	947%	1,338%

$\boldsymbol{avg\_CRS\_measure_i}$: *average measures of files involved in CRS;* $\boldsymbol{avg\_non\_CRS\_measure_i}$: *average measures of files not involved in CRS*

high-maintenance, but *Unstable Interface* and *Crossing* have the largest impact, and *Package Cycle* has the smallest impact.

Given that the files that participated in each anti-pattern instance can be determined, to compare the impact of different instances of the same anti-pattern, we calculate the history measures for each file involved in each instance. Based on this information, we can rank the detected instances of the same anti-pattern. Table 16 shows the size and total history measures of each identified *Unstable Interface* instance detected from Apache Avro. Developers could use this information to prioritize the instances for scrutiny and plan possible refactoring activities based on their own needs.

Using DSMs, the user can visualize how files involved in each anti-pattern are connected (structurally or evolutionarily) with each other. And this information provides direct guidance to the architect in terms of how to address the underlying problem. For example, given a set of the files infected by Unhealthy Inheritance, it is clear that either the dependency from the parent to its children need to be removed, or, the hierarchy need to be improved to follow Liskov Substitution Principle.

5.6 Industry Impact.

Our approach has also demonstrated its value in real-world industrial projects. In one industrial case study [10], we detected architecture roots [8]—a set of file groups covering most problematic files. In that study we detected 6 roots, each containing multiple architecture anti-patterns. Feedback from the development team was encouraging: our collaborators confirmed that the problems that we reported are the root cause of the maintenance difficulty they experienced, and they initiated refactoring actions based on our results.

In our most recent case study on eight industrial projects [21], we detected more than 6,500 instances of anti-patterns over the 8 studied projects (the number of instances detected in each project ranged from 54 to 2,188). The practitioners confirmed that these anti-patterns indeed effectively identified architecture problems responsible for high-maintenance within their projects. Six of the eight projects we studied planned to refactor, or have already embarked upon refactoring, based on our reported anti-patterns. And our architecture anti-pattern detection techniques have been

TABLE 7: Average measures for files involved in, vs. not involved in, Modularity Violation Group (MVG)

Subjects	$avg\_MVG\_measure_i$				$avg\_non\_MVG\_measure_i$				$measure_i\_inc$			
	BF	BC	CF	CC	BF	BC	CF	CC	BF	BC	CF	CC
Avro	2.5	41.7	10.8	447.7	0.4	8.3	2.2	135.3	533%	401%	385%	231%
Camel	0.8	18.1	9.4	239.1	0.1	6.8	1.9	74.4	512%	166%	406%	221%
Cassandra	5.3	158.7	25.3	904.3	0.2	12.6	2.5	126.4	2,376%	1,161%	923%	615%
CXF	2.0	46.7	10.7	334.0	0.3	9.5	1.5	57.1	693%	391%	630%	485%
Derby	5.3	167.1	12.5	583.6	1.7	52.2	2.1	165.8	210%	220%	499%	252%
Hadoop	1.6	307.1	16.8	614.1	1.1	129.6	1.6	78.9	48%	137%	956%	679%
Hbase	10.0	576.0	23.1	1577.1	1.6	72.2	2.6	213.8	533%	698%	784%	638%
Ivy	3.5	79.7	11.8	367.4	0.3	11.4	1.8	68.8	979%	601%	545%	434%
Mahout	1.3	39.9	9.8	372.7	0.3	13.6	2.2	114.2	317%	193%	337%	226%
OpenJPA	2.8	126.0	10.3	509.7	0.8	70.3	1.9	97.9	269%	79%	455%	420%
PDFBox	3.1	134.3	12.9	548.7	0.7	39.0	3.0	145.8	335%	244%	333%	276%
Pig	3.6	92.0	10.2	368.6	0.5	16.2	1.7	93.3	584%	470%	516%	295%
Tika	3.1	116.5	12.0	556.2	0.7	44.2	3.0	193.2	330%	164%	298%	188%
Wicket	2.4	64.0	10.7	310.9	0.5	22.5	2.1	79.2	346%	184%	403%	292%
ZooKeeper	4.7	150.9	12.4	613.2	0.4	24.5	1.6	111.8	1,075%	515%	685%	448%
Comm_1	5.4	347.7	8.9	431.0	0.3	8.9	0.5	12.7	1,789%	3,793%	1,596%	3,294%
Comm_2	2.0	42.4	6.7	72.4	0.2	1.2	0.8	2.8	788%	3,304%	780%	2,526%
Comm_3	2.2	83.0	6.5	339.3	0.1	1.4	0.3	5.2	3,321%	6,009%	2,350%	6,484%
Comm_4	12.1	401.6	41.2	1564.3	0.8	26.0	3.4	86.0	1,384%	1,446%	1,114%	1,720%

$\boldsymbol{avg\_MVG\_measure_i}$: *average measures of files involved in MVG;* $\boldsymbol{avg\_non\_MVG\_measure_i}$: *average measures of files not involved in MVG*

TABLE 8: Average measures for files involved in, vs. not involved in, Clique (CLQ)

Subjects	$avg\_CLQ\_measure_i$				$avg\_non\_CLQ\_measure_i$				$measure_i\_inc$			
	BF	BC	CF	CC	BF	BC	CF	CC	BF	BC	CF	CC
Avro	4.2	60.0	16.4	650.4	1.0	18.3	4.7	223.5	347%	228%	253%	191%
Camel	1.7	30.5	18.5	437.7	0.3	10.3	4.1	123.4	438%	197%	352%	255%
Cassandra	5.7	146.2	25.7	772.0	1.8	68.4	10.1	454.6	216%	114%	154%	70%
CXF	3.4	71.5	14.4	395.8	0.8	21.0	4.5	151.2	347%	241%	220%	162%
Derby	5.8	148.4	13.1	471.1	2.7	98.0	5.4	349.4	115%	51%	142%	35%
Hadoop	2.0	469.3	22.9	865.5	1.1	139.5	4.0	156.9	87%	236%	468%	452%
Hbase	18.4	820.3	41.8	1816.2	3.5	255.4	7.9	812.9	420%	221%	426%	123%
Ivy	3.0	79.1	10.0	333.0	1.2	27.9	4.8	151.6	147%	183%	105%	120%
Mahout	3.2	97.2	16.3	543.8	0.8	25.9	6.1	248.8	306%	276%	167%	119%
OpenJPA	2.9	119.7	9.9	456.4	1.0	78.7	3.0	161.8	187%	52%	233%	182%
PDFBox	4.1	169.8	16.8	715.8	1.6	76.2	6.8	295.7	151%	123%	148%	142%
Pig	3.2	85.7	8.5	309.4	0.8	21.3	2.7	132.4	277%	303%	211%	134%
Tika	4.1	170.7	17.8	920.7	1.7	71.1	6.4	321.5	148%	140%	176%	186%
Wicket	4.9	124.0	20.5	642.9	1.1	33.8	4.7	142.5	348%	267%	336%	351%
ZooKeeper	7.6	236.8	18.6	838.2	1.0	42.3	3.3	204.9	667%	460%	464%	309%
Comm_1	1.5	63.1	2.3	85.2	0.7	41.6	1.2	51.8	121%	52%	81%	65%
Comm_2	2.0	51.4	5.6	81.7	1.0	16.9	3.7	30.9	92%	204%	50%	164%
Comm_3	0.3	10.4	1.0	48.6	0.1	3.3	0.4	11.7	139%	213%	142%	316%
Comm_4	4.2	160.8	14.8	554.9	1.1	30.4	4.4	120.5	278%	429%	233%	360%

$\boldsymbol{avg\_CLQ\_measure_i}$: *average measures of files involved in CLQ;* $\boldsymbol{avg\_non\_CLQ\_measure_i}$: *average measures of files not involved in CLQ*

integrated into DV8[13], an architecture analysis tool that has been used by many industrial practitioners.

The *take-away* message is clear: to reduce maintenance difficulties in a project, developers should focus on the files involved in architecture anti-patterns. Our approach also quantifies the *severity* of each anti-pattern instance, which could help architects with selecting and prioritizing potential refactorings.

6 Limitations and Threats to Validity

In this section, we discuss the limitations of our tool, and threats to validity.

6.1 Limitations

First, the detection of three anti-patterns, *Unstable Interface*, *Modularity Violation Group*, and *Crossing*, requires both structural dependencies and revision history records. The detection of these anti-patterns depends on the availability of the project's revision history. For projects without this information, these three anti-patterns cannot be detected.

Second, our architecture anti-pattern detection is a retrospective analysis. The anti-patterns are detected by reverse-engineering the *as-built* code base and the revision history. Our approach provides means to help developers identify and understand the architecture problems that have incurred maintenance costs. In this paper, we did not explore the predictive power of these anti-patterns. It would be interesting to analyze which anti-pattern is likely to remain bug-prone or change-prone in the future.

Third, while our approach is scalable in terms of the number of detectable anti-patterns, so far we only defined just six. We have no way to prove that these six anti-patterns have covered all the most significant architecture problems. However, we have reason to believe that these six anti-patterns are at least adequate, as we have been pursuing this work for nearly five years and, over that time, the number of anti-patterns has only increased by one. If we identify more types of anti-patterns, it is easy to extend our detection tool.

Forth, when detecting *Unstable Interface*, *Crossing* and *Modularity Violation Group*, the results depend on the selec-

[13] https://www.archdia.net/products-and-services/

TABLE 9: Average measures for files involved, in vs. not involved in, Unhealthy Inheritance Hierarchy (UIH)

Subjects	$avg\_UIH\_measure_i$				$avg\_non\_UIH\_measure_i$				$measure_i\_inc$			
	BF	BC	CF	CC	BF	BC	CF	CC	BF	BC	CF	CC
Avro	2.9	40.2	12.1	572.6	1.1	20.2	4.9	213.7	170%	99%	147%	168%
Camel	1.1	27.0	11.9	305.2	0.4	10.6	4.7	136.3	202%	155%	150%	124%
Cassandra	7.9	256.1	34.2	1308.9	1.9	53.9	10.5	374.0	324%	375%	226%	250%
CXF	2.1	49.3	10.1	327.6	0.7	20.0	4.3	139.3	176%	147%	135%	135%
Derby	5.4	135.6	12.0	441.1	3.0	104.2	6.0	364.1	80%	30%	98%	21%
Hadoop	1.5	341.5	15.6	587.9	1.2	154.0	4.9	188.9	35%	122%	221%	211%
Hbase	14.0	597.3	32.5	1489.8	4.7	311.7	10.3	896.1	199%	92%	215%	66%
Ivy	3.3	77.6	11.3	394.1	1.2	30.7	4.7	143.3	166%	153%	141%	175%
Mahout	1.4	59.2	8.5	346.7	0.8	23.7	6.2	246.1	76%	149%	39%	41%
OpenJPA	3.6	146.3	12.9	653.4	1.0	76.3	2.8	142.9	262%	92%	357%	357%
PDFBox	4.9	224.9	17.6	819.2	1.9	80.5	8.2	343.4	166%	180%	114%	139%
Pig	3.9	98.1	10.6	375.8	1.3	34.8	3.8	163.6	203%	182%	179%	130%
Tika	3.1	105.6	11.8	470.5	1.3	65.5	5.2	313.8	149%	61%	127%	50%
Wicket	3.2	96.9	13.3	433.0	1.1	33.0	4.9	146.6	187%	194%	171%	195%
ZooKeeper	5.1	183.0	12.9	637.4	1.2	44.6	3.9	219.5	311%	310%	234%	190%
Comm_1	3.5	174.4	5.8	235.7	0.6	32.2	1.0	39.5	503%	442%	494%	497%
Comm_2	1.9	52.5	5.9	85.6	1.1	18.5	3.7	32.7	68%	184%	60%	162%
Comm_3	0.6	21.3	1.6	84.8	0.1	3.9	0.5	15.7	318%	445%	248%	440%
Comm_4	4.4	185.4	14.8	633.6	1.7	54.3	6.4	200.1	159%	242%	131%	217%

$\boldsymbol{avg\_UIH\_measure_i}$: *average measures of files involved in UIH;* $\boldsymbol{avg\_non\_UIH\_measure_i}$: *average measures of files not involved in UIH*

TABLE 10: Measures' average values for the files involved in vs. not involved in Package Cycle (PKC)

Subjects	$avg\_PKC\_measure_i$				$avg\_non\_PKC\_measure_i$				$measure_i\_inc$			
	BF	BC	CF	CC	BF	BC	CF	CC	BF	BC	CF	CC
Avro	2.9	45.9	10.9	503.8	0.5	10.8	3.4	148.1	487%	325%	217%	240%
Camel	0.8	18.2	8.3	209.1	0.3	10.2	4.5	133.8	133%	79%	84%	56%
Cassandra	3.8	113.7	18.3	644.2	0.7	30.0	5.2	283.2	442%	279%	252%	127%
CXF	1.7	37.8	8.6	240.2	0.7	19.6	4.0	141.8	156%	93%	115%	69%
Derby	4.8	116.6	10.9	366.9	2.9	109.3	5.7	391.9	70%	7%	93%	-6%
Hadoop	1.5	271.7	11.5	421.1	1.0	122.6	3.2	139.3	52%	122%	263%	202%
Hbase	8.5	375.7	18.8	783.9	2.3	347.4	5.9	1479.9	268%	8%	218%	-47%
Ivy	2.1	52.3	7.6	247.2	1.0	24.8	4.1	128.2	105%	110%	84%	93%
Mahout	1.5	56.9	9.1	375.5	0.7	20.1	5.7	225.6	114%	182%	60%	66%
OpenJPA	2.9	108.8	10.4	503.5	0.9	79.2	2.4	126.6	217%	37%	340%	298%
PDFBox	3.1	149.5	11.3	498.0	1.5	54.0	8.1	337.4	113%	177%	40%	48%
Pig	2.2	59.2	6.2	221.7	0.8	16.1	2.3	147.1	183%	269%	172%	51%
Tika	2.3	100.0	8.9	475.4	1.6	68.8	6.5	316.1	40%	45%	35%	50%
Wicket	1.9	54.8	7.8	230.2	0.9	26.5	4.1	133.1	123%	107%	92%	73%
ZooKeeper	2.8	98.3	7.9	375.9	0.7	30.6	2.1	187.6	306%	221%	278%	100%
Comm_1	1.3	79.0	2.1	95.8	0.6	27.3	1.1	37.2	137%	190%	91%	158%
Comm_2	1.6	37.9	4.8	59.9	1.1	17.6	3.7	32.7	44%	115%	30%	83%
Comm_3	0.2	8.5	0.8	32.7	0.1	2.2	0.3	9.5	161%	293%	158%	244%
Comm_4	2.8	95.5	10.7	364.4	1.1	33.0	3.9	110.1	162%	189%	176%	231%

$\boldsymbol{avg\_PKC\_measure_i}$: *average measures of the files involved in PKC;* $\boldsymbol{avg\_non\_PKC\_measure_i}$: *average measures of the files not involved in PKC*

TABLE 11: P-values of the Paired Wilcoxon signed-rank tests

Type	UIF	CRS	MVG	CLQ	UIH	PKC
UIF	-		**2.8E-13**	**5.0E-5**	**2.3E-8**	**2.8E-13**
CRS		-	**7.5E-13**	**4.6E-4**	**1.5E-9**	**5.3E-13**
MVG			-			**7.9E-10**
CLQ				-		**2.1E-11**
UIH					-	**2.0E-11**
PKC						-

TABLE 12: Effect sizes based on the average values of BF

Type	UIF	CRS	MVG	CLQ	UIH	PKC
UIF	-		**0.99**	0.35	**0.51**	**0.82**
CRS		-	**1.91**	0.38	**0.74**	**1.20**
MVG			-			**0.58**
CLQ				-	0.28	**0.76**
UIH					-	**0.88**
PKC						-

TABLE 13: Effect sizes based on the average values of BC

Type	UIF	CRS	MVG	CLQ	UIH	PKC
UIF	-		**0.78**	0.34	**0.50**	**0.74**
CRS		-	**0.76**	0.29	**0.50**	**0.76**
MVG			-			**0.61**
CLQ				-		**0.61**
UIH					-	**0.89**
PKC						-

TABLE 14: Effect sizes based on the average values of CF

Type	UIF	CRS	MVG	CLQ	UIH	PKC
UIF	-		**1.14**	0.41	**0.6**	**0.98**
CRS		-	**2.55**	0.46	**0.88**	**1.51**
MVG			-			**0.68**
CLQ				-	0.41	**1.08**
UIH					-	**1.00**
PKC						-

tion of thresholds. In our evaluation, we set these thresholds based on our observations and experience. We set the impact thresholds for *Unstable Interface* detection to be 10, meaning that an unstable interface has to structurally and evolutionarily impact at least 10 other files. When we decrease this threshold, more trivial instances are detected. For example, when we set the value to be 2, hundreds of trivial instances were identified within *Cassandra*. It is intuitive that, with a smaller threshold, the detected "interfaces" become less influential, and the problems detected are less significant. Higher thresholds may results in fewer instances but important ones may be missed. To evaluate *Crossing*, we set

TABLE 15: Effect sizes based on the average values of CC

Type	UIF	CRS	MVG	CLQ	UIH	PKC
UIF	-		**0.93**	0.46	**0.56**	**0.89**
CRS		-	**1.84**	**0.58**	**0.84**	**1.27**
MVG			-			**0.69**
CLQ				-		**0.88**
UIH					-	**0.92**
PKC						-

TABLE 16: Maintenance effort of Unstable Interface instances in Avro

Index	Size	Ins. BF	Ins. BC	Ins. CF	Ins. CC
UnstableInterface1	46	228	3,302	889	34,676
UnstableInterface2	38	196	2,749	761	25,213
UnstableInterface3	23	184	2,333	715	28,489
UnstableInterface4	13	10	330	113	3,076
UnstableInterface5	13	67	1,001	229	12,620

Ins. BF, BC, CF and CC: the total bug frequency, bug churn, change frequency and change churn of all files in each instance.

both fan-in and fan-out threshold be to 4, to avoid trivial instances.

In addition, the detection of all three history-related anti-patterns relies on the co-change threshold. In our evaluation, we set this threshold to be 2, to avoid trivial cases where two files were changed together just once. Decreasing this value would identify more instances, since more files will be considered as evolutionarily coupled. In a project with a long history, a higher value might be appropriate. We intend to try different strategies to model the evolutionary coupling between files, such as change probability [35] or analyzing the issue reports to determine whether and how often two files were changed for the same maintenance tasks.

With that being said, we admit that the chosen threshold is a limitation and a threat to external validity of our evaluation. Project architects or developers may have better knowledge to adjust the thresholds, and our tool allows the user to configure these thresholds. To explore how to best determine these thresholds is our future work.

6.2 Threats to Validity

Threats to construct validity. First, we cannot guarantee that the four history measures that we used in our evaluation are the best proxies for maintenance effort. Ideally, maintenance effort should be measured by actual effort, in terms of hours or budgets, spent on each file. But we cannot directly extract such data; we thus employed four history measures: bug frequency, bug churn, change proneness and change churn, that can be extracted from a project's revision history. We admit this is a threat to construct validity, since the measures we proposed may not reflect the true maintenance effort in terms of bug-proneness and change-proneness. To address this issue partially, we not only considered the frequency of a files involved in bug fixes or changes, but also calculated the lines of code changed in bug fixes or changes. For example, we calculated both bug frequency and bug churn to reflect a file's bug-proneness; it seems reasonable that a bug-prone file should be involved in more bug fixes and have more lines of code changed in bug fixes. We will keep looking for better approximations for maintenance effort, and plan to compare them with our proxy measures.

Second, we cannot guarantee the accuracy of the measures. To calculate the bug frequency and bug churn, we use the pattern matching method in [28] to identify a bug-fixing commit if its change message contains a bug ticket ID. A file's bug frequency will be the number of times a file is involved in bug-fixing commits, and the bug churn will be the LOC changed in these commits. However, we cannot guarantee that the bug data extracted from revision history are not biased. As prior studies [36], [37] have shown: 1) a file changed in a bug-fixing commit doesn't necessarily means this file is modified for a bug; 2) there is often no explicit link that can be used for targeting the bug-fixing commit in revision history. Thus findings with respect to the research questions could be impacted by the accuracy of available data. We acknowledge this is a threat to construct validity and it requires more investigation.

Threats to external validity. A threat to external validity is in our data set. We only analyzed nineteen projects implemented in Java or C#. Although the 15 open source projects have different sizes and domains, all of them are implemented in Java. The 4 selected industrial projects are provided by different teams but within the same company, and they are all implemented in C#. Thus we can not claim that our results are generalizable across all software projects, especially for those using non-object-oriented languages. For example, the *Unhealthy Inheritance* anti-pattern does not exist in projects using non-object-oriented languages. To overcome this issue, we are in process of applying our approach to more projects implemented in other programming languages and from different industrial collaborators. Due to the limitations of our data set, we only processed projects that use SVN or Git, hence we can not claim that our results are generalizable to other projects managed by other version control tools. Extending the experiments to a broader set of projects is future work.

Threats to internal validity. In our experiment, we used *Understand* to derive structural dependencies among files, and used this dependency information as input to our detection algorithm. Consequently, any imprecision in Understand could have negatively affected our detection results. However, our approach does not inherently depend on Understand: any reverse engineering tool that can extract and export file dependencies, e.g., inheritance, dependency, into readable formats could be used in our tool chain.

We also admit that an effect of file size could be a threat to internal validity. Many studies [38], [39], [40], [41], [42] have shown that file size is correlated with bug rates and change rates. That is, if a file has more LOC, it is more likely to be involved in bugs or changes. To partially address this problem, we investigated whether the identified anti-pattern instances are not merely a set of large files. Using the instances of Unstable interface in *Avro* project as an example, we found that only 24% of them are in the top 10% largest files (in terms of LOC). We achieve consistent results for the other anti-patterns across all projects, which indicates that the detected anti-patterns are not dominated by large files.

In addition, we acknowledge a possible bias in our detection in that we do not consider the architectural role of each file [43], [44], [45]. For example, a dependency cycle is an intrinsic part of the visitor pattern: the visitor will *visit* the element while the element *accepts* the visitor. Consider another example, a utility file often has more dependents than other files, so changes to this file are likely to cause more files to be changed together. In this paper, we did not distinguish such cases due to our limited knowledge about the analyzed projects. Thus, we consider this to be a threat to validity, and exploring the impact of such architectural decisions is our future work.

7 Related work

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

Defect and Change Prediction: Selby and Basili [46] have investigated the relation between dependency structure and software defects. There have been numerous studies of the relationship between evolutionary coupling and error-proneness [2], [47], [48]. For example, Cataldo et al.'s [48] work reported a strong correlation between density of change coupling and failure proneness. And Ostrand et al. [49]'s study demonstrated file size and file change information were very useful to predict defects. The relation between various metrics and software defects has also been widely studied. For example, Nagappan et al. [3] investigated various complexity metrics and demonstrated that these metrics are useful and successful for defect prediction. However, they also reported that, in different projects, the best metrics for prediction may be different. Besides, defect localization has also been widely studied [5], [6], [7]. For example, Jones et al. [5] used the ranking information of each statement to assist fault location.

In the field of change-proneness prediction, researchers have proposed numerous studies as well. Alshayeb and Li [50] investigated the effectiveness of six code metric for predicting change-proneness in terms of changed LOC. Catolino et al.'s work [51] demonstrated that developer-related factor could be used to improve the performance of change prediction. Lu et al.'s [52] work evaluated the ability of different object-oriented (OO) metrics for change-proneness prediction. The evolution-based metrics have widely studied for change-proneness prediction as well. Girba et al. [53] studied the changes in the evolution of OO projects by defining historical measurements and presented that classes that changed most in recent history would have high potential to be changed in the near future. Tsantalis et al. [54] presented a probabilistic approach to estimate the change proneness of an object-oriented design by assessing the probability that each class will change in a future modification. Elish and Al-Khiaty [55] proposed a suite of evolution-based metrics and presented 4 models to predict change-prone classes. Various learning method have been used for change-proneness prediction. For example, Ge et al. [56] proposed a Deep Metric Learning model to detect change-proneness of classes. Amoui et al. [57] presented a temporal change prediction method using neural network to predict future change dates of software entities. Malhotra and Khanna [58] used code metrics as features and 6 machine learning methods for change-prediction.

Different from our work, all the above research focuses on individual files or classes as the unit of analysis, but do not take architectural relations among files into consideration. Our study focuses on architectural anti-patterns, e.g., file relations that violate design principles and incur high maintenance costs. Both file complexity and architectural complexity are, we conjecture, contributors to overall error-proneness in a software system.

In addition, the goal of our work is not to predict defects or changes in a software system, but to reveal the underlying architectural relations among files that have significant impact on maintainability.

Architecture Smell Detection: Architecture smells describe the problems that identified on architecture level [59], [60]. Various techniques have been widely studied to detect design or architecture smells. Garcia et al.'s [12], [60] studies reported a catalog of architectural bad smells using a format similar to the design patterns [22], which presented possible identifications on these architecture smells. Le and Nenad [16] extended this catalog by formalizing and identifying eight new architecture smells, which could be categorized as Interface-based smells or Change-based smells. In the recent study of Le et al. [61], they detected multiple types of architecture smells and investigated the relationships between smells and a project's issues. Marinescu [62] presented detection strategies that exploit metrics-based rules to detect design flaws. Garcia et al. [63] presented a machine learning-based method to recover an architecture view which helps detect architecture drift or erosion. Lenhard et al. [64] selected a set of source code metrics, such as WMC, code size, etc., and investigated the relationship between architecture degradation and these source code metrics. Sharma et al. [65] presented a tool named Designite[14] to detect several structure design smells. In the studies of Fontana et al. [66], [67], they described three representative architecture smells and presented a tool, Arcan, to support their detection. Instead of investigating detection techniques, Sousa et al.'s work [68] focused on how developers actually proceed to identify design problems by using the design problem symptoms.

Researchers have studied the relation between code smells [11] and architecture design smells. For example, Macia et al.'s [69] study presented a tool, SCOOP, to detect code anomalies which are related to architecture problems. Bertran [70] analyzed software architecture quality based on code smells relations, and presented that code anomalies could be used for detecting architectural problems. Yamashita et al. [71] studied two kinds of code smells: collocated smells — code smells occurring together in the same file; and coupled smells — code smells interacting across coupled files. They observed that only considering collocated smells would affect the accuracy for the detection of design problems, since coupled smells may reveal critical design problems. Lenhard et al. [72] used JabRef to investigate the suitability of code smell detection tools for detecting the problems related to architectural degradation. Oizumi et al. [73] proposed the concept of agglomerations and demonstrated that certain forms of agglomerations could be used to locate architecture design problems.

Multiple commercial tools have been proposed to support the detection of design or architecture smells. For example, AiReviewer[15] supports the detection of both code

[14] http://www.designite-tools.com/

[15] http://www.aireviewer.com/

smells [11] and several design or architectural smells. SonarGraph[16] is a commercial tool helping detect violations to software architecture. Structure 101[17] could also detect different kinds of architecture smells.

These detection techniques use static code or architecture information only. By contrast, our approach integrates code dependency with history records. A few studies have also considered evolution history when defining architecture smells. For example, Le et al. [74] proposed a taxonomy of architecture smells, two of which are based on evolution history, but without rigorous formal definitions. Roveda et al. [75] proposed an Architecture Debt Index (ADI) that integrates a suite of existing architecture smells, including some of our anti-patterns, and takes into account their severity as revealed in a project's evolution history. Unlike our approach, these studies did not formalize these architecture smells, nor did they explicitly demonstrate the feasibility of their automatic detection. Moreover, these studies did not empirically demonstrate that these architecture smells have significant impact on maintenance effort.

We note that various researchers and tools define architecture "smells", "anti-patterns" in different ways based on different rationales. Our anti-patterns are defined based on design rule theory and prevailing design principles. We introduced 5 anti-patterns in our prior work [1] and added one more after observing numerous open source and industrial projects. It is our goal to minimize overlaps between detected anti-pattern instances, so that the user can examine, rank, and prioritize these problems more efficiently.

Architecture Representation and Non-automated Analysis: Our work is also related to research on software architecture representation and analysis. There has been substantial study on the uses of architecture representations (views) [76], [77], [78], [79], [80] and how they support design and analysis. For example, Kruchten [77] proposed the 4+1 view model of architecture. Our architecture representation, DRSpace, focuses on just a single architecture view—the module view. As one type of module view, DRSpaces are organized based on design rules and independent modules.

The analysis of architecture has also been widely studied. Kazman et al. [81] created the Architecture Tradeoff Analysis Method for analyzing architectures. Gamma et al. [22] presented a catalog of design patterns to present generally reusable solutions to recurring problems in software design environment. Andrew [13] proposed the anti-pattern to represent recurring problems that are harmful to software systems. These methods are not automated, and hence depend on the skill of the architecture analysts. Our approach, by contrast, can automatically detect architecture anti-patterns that impact error- and change-proneness of files software, and guide the user to diagnose software quality problems, and prioritize refactoring plans accordingly.

8 Conclusion

In this paper, we have formally defined six architecture anti-patterns that help to identify recurring and high-maintenance software problems. We defined these anti-pattern based on the violation of well-known design principles. We presented a tool to automatically detect the existence of these anti-patterns and the files involved. We evaluated our approach using fifteen open source projects and four commercial projects. The results show that all six anti-patterns are highly associated with error-proneness and change-proneness of source files: Files involved in any anti-patterns have higher bug and change frequencies, and consume more effort to fix bugs and make changes. We also demonstrated that the more anti-patterns a file is involved in, the more likely it is to be error-prone, and the more effort it takes to fix and modify. The data also revealed that Unstable Interface and Crossing are most severe anti-patterns because they are associated with much higher level of error-proneness and change-proneness.

This knowledge can be used by an architect to pinpoint architecture problems, quantify their severity, and determine priorities for refactoring. DSMs can be used to visualize each instance of anti-patterns. Paired with anti-pattern descriptions and the concrete sets of files involved, this visualization provides direct guidance to the architect on how each problem should be addressed.

This, we claim, is a powerful technique. This technique provides a foundation for architects and analysts to strategically address technical debt in a software system. And, as we have shown in our prior work [10], [21], this technical analysis can be accompanied by a return-on-investment analysis which equips architects with the information they need to choose, prioritize, and justify to management the areas of technical debt that they most urgently need to address.

Acknowledgments

This work was supported in part by the National Science Foundation of the US under grants CCF-1514315 and CCF-1514561.

References

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

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

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

[4] 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 International Conference on Software Engineering*, 2008.

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

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

[7] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: statistical model-based bug localization," in *13rd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2005.

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

[16] https://www.hello2morrow.com/products/sonargraph

[17] https://structure101.com/

[9] Y. Cai, L. Xiao, R. Kazman, R. Mo, and Q. Feng, "Design rule spaces: A new model for representing and analyzing software architecture," *IEEE Transactions on Software Engineering*, 2018.

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

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

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

[13] A. Koenig, *Patterns and antipatterns*. The patterns handbooks, 1998.

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

[15] A. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 242–251.

[16] D. Le and N. Medvidovic, "Architectural-based speculative analysis to predict bugs in a software system," in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 807–810.

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

[18] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, 2003.

[19] L. Barbara, "Keynote address - data abstraction and hierarchy," in *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*, 1987, pp. 17–34.

[20] D. L. Parnas, "Software fundamentals," 2001, ch. On a Buzzword: Hierarchical Structure, pp. 161–170.

[21] R. Mo, W. Snipes, Y. C. S. Ramaswamy, R. Kazman, and M. Naedele, "Experiences applying automated architecture analysis tool suites," in *Proc. 33rdIEEE/ACM International Conference on Automated Software Engineering*, 2018, pp. 779–789.

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

[23] Y. Cai and K. J. Sullivan, "Modularity analysis of logical design models," in *Proc. 21st IEEE/ACM International Conference on Automated Software Engineering*, Sep. 2006, pp. 91–102.

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

[25] Y. Cai, H. Wang, S. Wong, and L. Wang, "Leveraging design rules to improve software architecture recovery," in *Proc. 9th International ACM Sigsoft Conference on the Quality of Software Architectures*, Jun. 2013, pp. 133–142.

[26] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting software modularity violations," in *Proc. 33rd International Conference on Software Engineering*, May 2011, pp. 411–420.

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

[28] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, 2005, pp. 1–5.

[29] L. V. Hedges, "Distribution theory for glasss estimator of effect size and related estimators," *Journal of Educational Statistics*, vol. 6, no. 2, pp. 107–128, 1981.

[30] J. Cohen, *Statistical power analysis for the behavioral sciences*, 1977.

[31] G. M. Sullivan and R. Feinn, "Using effect sizeor why the p value is not enough," *Journal of graduate medical education*, vol. 4, no. 3, pp. 279–282, 2012.

[32] D. JA, "How to select, calculate, and interpret effect sizes," *Journal of Pediatric Psychology*, vol. 34, no. 9, pp. 917–128, 2009.

[33] P. D. Ellis, *The Essential Guide to Effect Sizes: Statistical Power, Meta-Analysis, and the Interpretation of Research Results*, 2010.

[34] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: A practical and powerful approach to multiple testing," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.

[35] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proc. 26th International Conference on Software Engineering*, May 2004, pp. 563–572.

[36] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, 2005, pp. 1–5.

[37] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 392–401.

[38] K. E. Emam, S. Benlarbi, N. Goel, , and S. N. Rai, "The confounding effect of class size on the validity of object-oriented metrics," *IEEE Transactions on Software Engineering*, vol. 27, no. 7, pp. 630–650, 2001.

[39] S. M. Olbrich, D. S. Cruzes, and D. I. Sjoberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *IEEE International Conference on Software Maintenance*, 2010, pp. 1–10.

[40] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proceedings of the 2Nd Workshop on Managing Technical Debt*, 2011, pp. 17–23.

[41] D. Landman, A. Serebrenik, and J. Vinju, "Empirical analysis of the relationship between cc and sloc in a large corpus of java methods," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 221–230.

[42] M. Aniche1, G. Bavota, C. Treude, A. van Deursen, and M. A. Gerosa, "A validated set of smells in model-view-controller architectures," in *IEEE International Conference on Software Maintenance*, 2016, pp. 233–243.

[43] M. Aniche, C. Treude, A. Zaidman, A. van Deursen, and M. A. Gerosa, "Satt: Tailoring code metric thresholds for different software architectures," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2016, pp. 41–50.

[44] N. A. Ernst, "Bayesian hierarchical modelling for tailoring metric thresholds," in *15th International Conference on Mining Software Repositories*, 2018.

[45] M. Dosea, C. Sant'Anna, and B. C. da Silva, "How do design decisions influence the distribution of software metrics?" in *The IEEE/ACM International Conference on Program Comprehension*, 2018.

[46] R. W. Selby and V. R. Basili, "Analyzing error-prone system structure," *IEEE Transactions on Software Engineering*, vol. 17, no. 2, pp. 141–152, Feb. 1991.

[47] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[48] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, "Software dependencies, work dependencies, and their impact on failures," *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 864–878, Jul. 2009.

[49] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, 2005.

[50] M. Alshayeb and W. Li, "An empirical validation of object-oriented metrics in two different iterative software processes," *IEEE Transactions on Software Engineering*, vol. 29, no. 11, pp. 1043–1049, 2003.

[51] G. Catolino, F. Palomba, A. D. Lucia, F. Ferrucci, and A. Zaidman, "Developer-related factors in change prediction: An empirical assessment," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017, pp. 186–195.

[52] H. Lu, Y. Zhou, B. Xu, H. Leung, and L. Chen, "The ability of object-oriented metrics to predict change-proneness: A meta-analysis," *Empirical Software Engineering*, vol. 17, no. 3, pp. 200–242, 2012.

[53] T. Girba, S. Ducasse, and M. Lanza, "Yesterday's weather: guiding early reverse engineering efforts by summarizing the evolution of changes," in *20th IEEE International Conference on Software Maintenance*, 2004, pp. 40–49.

[54] N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides, "Predicting the probability of change in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 7, pp. 601–614, 2005.

[55] M. O. Elish and M. A.-R. Al-Khiaty, "A suite of metrics for quantifying historical changes to predict future changeprone classes in object-oriented software," *Journal of Software: Evolution and Process*, vol. 25, no. 5, pp. 407–437, 2013.

[56] Y. Ge, M. Chen, C. Liu, F. Chen, S. Huang, and H. Wang, "Deep metric learning for software change-proneness prediction," in *Intelligence Science and Big Data Engineering*, 2018, pp. 287–300.

[57] M. Amoui, M. Salehie, and L. Tahvildari, "Temporal software change prediction using neural networks," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 7, pp. 995–1014, 2009.

[58] R. Malhotra and M. Khanna, "An empirical study for software change prediction using imbalanced data," *Empirical Software Engineering*, vol. 22, no. 6, pp. 2806–2851, 2017.

[59] M. Lippert and S. Roock, *Refactoring in Large Software Projects: Performing Complex Restructurings Successfull*, 2006.

[60] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Toward a catalogue of architectural bad smells," in *Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems*, 2009, pp. 146–162.

[61] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural decay in open-source software," in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 176–17 609.

[62] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," in *20th IEEE International Conference on Software Maintenance*, 2004, pp. 350–359.

[63] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, pp. 552–555.

[64] J. Lenhard, M. Blom, and S. Herold, "Exploring the suitability of source code metrics for indicating architectural inconsistencies," *Software Quality Journal*, 2018.

[65] T. Sharma, P. Mishra, and R. Tiwari, "Designite - a software design quality assessment tool," in *2016 IEEE/ACM 1st International Workshop on Bringing Architectural Design Thinking Into Developers' Daily Activities (BRIDGE)*, 2016, pp. 1–4.

[66] F. A. Fontana, I. Pigazzini, R. Roveda, and M. Zanoni, "Automatic detection of instability architectural smells," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 433–437.

[67] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. D. Nitto, "Arcan: A tool for architectural smells detection," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 282–285.

[68] L. Sousa, A. Oliveira, W. Oizumi, S. Barbosa, A. Garcia, J. Lee, M. Kalinowski, R. de Mello, B. Fonseca, R. Oliveira, C. Lucena, and R. Paes, "Identifying design problems in the source code: A grounded theory," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 921–931.

[69] I. Macia, R. Arcoverde, E. Cirilo, A. Garcia, and A. von Staa, "Supporting the identification of architecturally-relevant code anomalies," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 662–665.

[70] I. M. Bertran, "Detecting architecturally-relevant code smells in evolving software systems," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 1090–1093.

[71] A. Yamashita, M. Zanoni, F. A. Fontana, and B. Walter, "Inter-smell relations in industrial and open source systems: A replication and comparative analysis," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 121–130.

[72] J. Lenhard, M. M. Hassan, M. Blom, and S. Herold, "Are code smell detection tools suitable for detecting architecture degradation?" in *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings (ECSA)*, 2017, pp. 138–144.

[73] W. Oizumi, A. Garcia, L. da Silva Sousa, B. Cafeo, and Y. Zhao, "Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 440–451.

[74] D. M. Le, C. Carrillo, R. Capilla, and N. Medvidovic, "Relating architectural decay and sustainability of software systems," in *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2016, pp. 178–181.

[75] R. Roveda, F. A. Fontana, I. Pigazzini, and M. Zanoni, "Towards an architectural debt index," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2018, pp. 408–416.

[76] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley, 2012.

[77] P. B. Kruchten, "The 4+1 view model of architecture," *IEEE Software*, vol. 12, pp. 42–50, 1995.

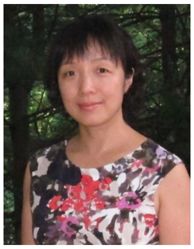
[78] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[79] D. Falessi, G. Cantone, R. Kazman, and P. Kruchten, "Decision-making techniques for software architecture design: A comparative survey," *ACM Computing Surveys*, vol. 43, no. 4, pp. 1–28, Oct. 2011.

[80] 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 Transactions on Software Engineering*, vol. 42, no. 3, pp. 220–260, 2016.

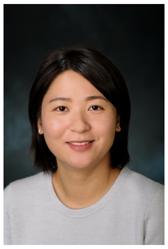
[81] R. Kazman, M. Barbacci, M. Klein, S. J. Carriere, and S. G. Woods, "Experience with performing architecture tradeoff analysis," in *Proc. 16th International Conference on Software Engineering*, May 1999, pp. 54–64.

Ran Mo is an Associate Professor at Central China Normal University. Dr. Mo received his Ph.D degree in Computer Science at Drexel University in 2018, advised by Dr. Yuanfang Cai. His research mainly focuses on measuring and analyzing the quality of software architecture.



Yuanfang Cai is currently an Associate Professor at Drexel University, USA. In 2006, Dr. Cai received her Ph.D degree in Computer Science from the University of Virginia. Her primary research interests are 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. Dr. Cai is currently serving on program committees and organizing committees for multiple top conferences, and she also serves as an associate editor and editorial board member for top journals in the area of software engineering. The tools and technologies from Dr Cai's research, including Decoupling Level (DL) and Titan tool suite, have been licensed and adopted by multiple multinational corporations.

Rick Kazman is a Professor at the University of Hawaii and a Research Scientist at the Software Engineering Institute of Carnegie Mellon University. His primary research interests are software architecture, design and analysis tools, software visualization, and software engineering economics. Kazman 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-Benet Analysis Method) and the Dali and Titan tools. He is the author of over 200 publications, and co-author of several books, including Software Architecture in Practice, Designing Software Architectures: A Practical Approach, Evaluating Software Architectures: Methods and Case Studies, and Ultra-Large-Scale Systems: The Software Challenge of the Future.



Lu Xiao is an Assistant Professor in the School of Systems and Enterprises at Stevens Institute of Technology. Her research focuses 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. She earned her PhD in Computer Science at Drexel University in 2016, advised by Dr. Yuanfang Cai.

Qiong Feng is a Ph.D. candidate in the Computer Science Department at Drexel University. Her research mainly focuses on analyzing the evolution of software architecture.