Design Rule Spaces: A New Form of Architecture Insight

Lu Xiao, Yuanfang Cai
Drexel University
Philadelphia, PA, USA
{lx52, yfcai}@cs.drexel.edu

Rick Kazman
University of Hawaii & SEI/CMU
Honolulu, HI, USA
kazman@hawaii.edu

ABSTRACT

In this paper, we investigate software architecture as a set of overlapping *design rule spaces*, formed by one or more structural or evolutionary relationships and clustered using our design rule hierarchy algorithm. Considering evolutionary coupling as a special type of relationship, we investigated (1) whether design rule spaces can reveal structural relations among error-prone files; (2) whether design rule spaces can reveal structural problems contributing to error-proneness. We studied three large-scale open source projects and found that error-prone files can be captured by just a few design rule sub-spaces. Supported by our tool, Titan, we are able to flexibly visualize design rule spaces formed by different types of relationships, including evolutionary dependencies. This way, we are not only able to visualize which error-prone files belong to which design rule spaces, but also to visualize the structural problems that give insight into *why* these files are error prone. Design rule spaces provide valuable direction on which parts of the architecture are problematic, and on why, when, and how to refactor.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]

General Terms

Design

Keywords

Software Architecture, Software Quality, Architecture Recovery

1. INTRODUCTION

In this paper, we present *design rule spaces*, a new form of architecture representation that uniformly captures both architecture and evolution structures to bridge the gap between architecture and defect prediction. In the field of reverse engineering, numerous techniques have been proposed to recover a software architecture from source code, such as Bunch [13], ACDC [20], and LDA [10]. These approaches aim to improve the accuracy and comprehensibility of the recovered architecture. However, a means of leveraging the recovered architectural structure to inform software quality issues, such as the location of defects, has not been explored.

On the other hand, in the field of data mining, many approaches have been proposed to leverage co-change information in revision history to locate error-prone files, and construct defect predictors [7, 11, 12, 14]. Although our recent industrial case study [18] revealed that architecture problems are the root cause of a large number of defects, the relationships between file error-proneness and architectural properties have been largely ignored by the software community.

Our work is rooted in Baldwin and Clark's concept of *design rule* (DR) [2]–architectural interfaces that decouple the system into independent modules. We consider *design rules*, and *modules* framed by the design rules, as basic elements of software architecture, and propose a new concept—the *Design Rule Hierarchy* (DRH) [3, 23]—to capture their relationships.

After examining how files change together in revision histories [18, 22], we found that when a group of files frequently changes together, but they lack syntactical or architectural relationships, it often implies unencapsulated assumptions, implementation errors, or architectural problems. We term these *modularity violations.* In our prior studies, however, we had to manually examine hundreds of co-changed files and read thousands of developers' comments to verify if a modularity violation indeed indicated an issue.

The problem is that we lack a methodology to directly and effectively link software architecture with error-proneness: not all error-prone files have modularity violations, and vice versa. In this paper, we introduce the concept of *design rule space* (DRSpace) to bridge the gap between architecture and quality concerns. A DRSpace is a special graph whose vertices are a set of classes, with the following features:

1) Its edges can be one or more selected types of relationships between classes, including evolutionary coupling [9] derived from revision history. Some of these relationship types can be designated as *primary relations*, and the other types are *secondary relations.*

2) It must have one or more *leading classes*, which are usually the design rules of the space.

3) It has to be clustered into the form of a design rule hierarchy [3, 4, 23] based on its *primary relation.*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ICSE'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-5/14/05...$15.00
http://dx.doi.org/10.1145/2568225.2568241

In this paper, we show that analysis based on just a single relationship is not sufficient to capture the complexity of software systems. Instead, software architecture should be viewed as multiple overlapping DRSpaces.

An extended example that we present in Section 3 shows that each type of dependency relation, such as aggregation and inheritance, forms its own meaningful DRSpace. By choosing evolutionary coupling as a *secondary* relation within a DRSpace, the modularity violations within the space can be visualized. Moreover, design patterns used in the system also form unique DRSpaces that overlap with other DRSpaces formed by other patterns or relations.

As the first step toward evaluating the potential of DRSpaces, especially in terms of informing quality issues, we investigated the relation between DRSpaces and bug spaces—the spaces formed by error-prone files—in three large-scale open source projects: JBoss, Hadoop, and Eclipse JDT. Supported by our tool, Titan, we obtained three major results:

First, if the file leading a DRSpace is error-prone, then a significant portion of the files within the DRSpace are also error-prone. We call a DRSpace led by an error-prone file an *error-prone DRSpace*.

Second, although a project may have hundreds of error-prone files, these files are usually contained in a few error-prone DRSpaces. In all three projects, more than 50% of the error-prone files are captured by just 5 error-prone DRSpaces.

Third, all error-prone DRSpaces exhibit multiple structural and evolution issues, either violating commonly accepted design principles, or revealing exceptionally unstable architectural interfaces. Our result also shows that not all structural problems, such as cyclic dependencies, cause quality or maintainability issues. By choosing evolution coupling as the secondary relation within a DRSpace, we can visualize which structural problems are actually incurring high maintenance costs.

These results imply that, when investing the root cause of bugs, we should consider the DRSpace that these error-prone files belong to, because structural issues may contribute to their buggniness. We will present evidence that error-prone files influence the error-proneness of other files within the same DRSpace. Furthermore, the DRSpace can help in identifying the root cause of the bugginess (for example, cyclic dependencies), and thus indicate how such problems may be resolved.

The rest of the paper is structured as follows. Section 2 presents our prior work that this work is based upon. Section 3 introduces the concept of DRSpaces using an example. Section 4 introduces our tool, Titan. Section 5 presents our evaluation and results. Sections 6, 7, and 8 discuss the results, present related work, and provide our final conclusions.

2. BACKGROUND

Our work is rooted in the concept of a *design rule* (DR). DRSpaces, based on design rules, are presented in the form of a *design structure matrix* (DSM) [2], organized as a *design rule hierarchy* (DRH) [3, 4, 23]. A DRSpace can help to visualize *modularity violations* [22].

Design Rule (DR). Baldwin and Clark [2] proposed that a modular structure should be framed by *design rules*, the architectural decisions that decouple the rest of system into *modules*, so that modules can evolve independently from each other. In modern object-oriented software systems, design rules are usually implemented in the form of interfaces or abstract classes. For example, if a software system employs an observer pattern, the pattern should be led by an observer interface, which decouples subjects from concrete observers. If the interface is stable, changes to concrete observers and subjects should not influence each other. In this case, we consider the observer interface as a *design rule*, and the subjects and concrete observers form two independent *modules*.

Design Structure Matrix(DSM). These concepts can be visualized as a *design structure matrix* (DSM). A DSM is a square matrix with its rows and columns labeled by the same set of element names and/or numbers, in the same order. A cell along the diagonal represents self-dependency, and an non-empty off-diagonal cell captures some relation between the element on the row to the element on the column. For example, in Figure 1, the mark in cell, c:(r4,c1), indicates that `mij.ast.Number` (row 4) depends on `mij.ast.Node` (column 1). The shaded squares along the diagonal model sets of elements that are grouped together.

Design Rule Hierarchy (DRH). In our prior work, we proposed a concept called *design rule hierarchy* (DRH), a layered structure that reveals how design rules decouple the rest of the system into modules. A DSM with DRH structure has the following characteristics: 1) the elements in the lower layers of the hierarchy only depend on the elements within higher layers, i.e., design rules; 2) elements within the same layer are separated into mutually independent groups, that is, independent *modules*.

Figure 3 depicts a DRH DSM with two layers: l1:(rc1-5) and l2:(rc6-13). Within l2, there are 3 mutually independent modules: m1:(rc6-7), m2:(rc8-9), and m3:(rc10-13). In this case, the 5 elements in the first layer are the *design rules* of the second layer because they influence the second layer elements, but are not influenced by them.

In our recent work [4], we extended the original DRH algorithm to better support architecture recovery from source code, which we called the *architectural design rule hierarchy* (ArchDRH) algorithm. ArchDRH recognizes the existence of another special type of element commonly found in a software architecture: *control programs*. A control program, such as a class with a `main` function, usually depends on many other classes, but is not depended on by them. ArchDRH additionally separates these control elements into the bottom of a module, and supports recursively clustering the rest of the module into a DRH structure.

Figure 4 depicts an output of ArchDRH. The algorithm first identifies three layers: l1:(rc1-2), l2:(rc3-21), and l3:(rc22-32). Within l2, it then calculates two modules, m1:(3-10) and m2:(11-21), and then recursively calculates the DRH structure within each module. For example, m2 is further calculated into a two-layer hierarchy: l4:(rc11-20) and l5:(rc21-21). It first separates `mij.parse.Parser`, the control class of the module, into l5, and then aggregates the rest into one module m:(rc10-20) within l4. All the DSMs shown in this paper are calculated using this ArchDRH algorithm.

To simplify the presentation, we will call the structure processed and output by our recursive ArchDRH algorithm a design rule hierarchy (DRH).

Modularity Violation. In our first work exploring the relationships between structure and history, we presented a tool called *Clio* that computes the discrepancies between how files *should* change together based on their modular structure, and how they *actually* changed together as revealed by version history, a concept called *modularity violation*. Our experiments with open source projects showed that Clio not only revealed many known structural problems, such as code clones or poor inheritance, it also detected large numbers of undefined couplings that were subsequently verified to be harmful.

In the work of Schwanke et. al. [18], we reported a case study of applying the modularity violation detection approach to an industrial project. In this study, we identified a project's architectural problems by analyzing and investigating the structural significance of the most complicated and error prone files. We also investigated how files evolved together without being structurally related. In this way, we identified large number of "shared secrets" (undocumented assumptions), implementation errors, and architecture violations. Many of these problems directly responsible for subsequent defects. The developers confirmed the architecture problems we identified, and the project manager then wrote a refactoring proposal based on the results of our study. The proposal was accepted by management and implemented.

In this paper we build, and improve, upon the above concepts, methodologies, and tools. In particular, our prior tool for identifying modularity violations, Clio, usually outputs hundreds of modularity violations. We had to spend a great deal of effort manually verifying which ones really indicated architecture problems, by reading developers' commit messages, reading the source code, or talking to developers directly. The DRSpace presented in the next section helps identify these problems much more efficiently.

3. DESIGN RULE SPACES

In this section, we use an example to illustrate the concept of a *design rule space* (DRSpace).

Definition. As already mentioned, a DRSpace is defined as a graph with the following characteristics:

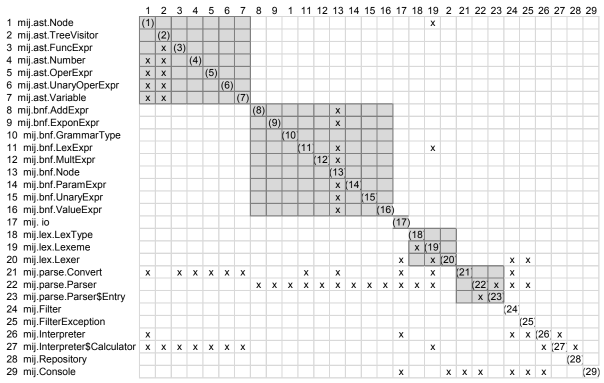
(1) A DRSpace is composed of a set of classes (files), and one or more selected types of relations between them. The major types of relations we explore in this paper include three major *structural* relations in object-oriented design: *inheritance/realization*, *aggregation*, and *dependency*, as well as one *evolutionary* relation: *evolutionary coupling*, derived from revision histories [9]. For example, if two files are committed together 10 times, as recorded in the version control system, we consider that they are evolutionarily coupled with a weight of 10, during the specified time period. Theoretically, a DRSpace can accommodate additional types of relations, such as run-time or data-flow relations, which we will explore in the future.

(2) The vertices (classes) of a DRSpace must be clustered into the form of design rule hierarchy (DRH) [3,4,23] based on one or more selected types of relations. We call these selected relations that form a DRH structure the *primary relations* of the DRSpace. Using our tool, Titan, the user can choose to include other types of relations in a DRSpace for analysis purposes, which we call the *secondary relations* of the DRSpace. For example, to visualize modularity violations, we first create a DRSpace with one or more of the three structural relations to show the designed modular structure, and then choose evolutionary coupling as the secondary relation to visualize where violations occur.

(3) A DRSpace must have one or more *leading classes*, that is, the de facto *design rules* of the space. If the DRSpace's DRH has more than one layer, then the classes within the first layer are the leading classes of the DRSpace. If a DRSpace only has one layer, then all the classes can be considered as leading classes. If a DRSpace, *ds*, has a leading class *c*, we also say that *ds* is led by *c*.

We call them leading classes to distinguish them from the original concept of *design rules*. The latter usually refers to architecture decisions of the overall system. A leading class of a DRSpace, by contrast, is only leading relative to a specific DRSpace, and may or may not be an architecturally important design rule. At one extreme, if a DRSpace only has one class, we still call this class the leading class of the space, but it cannot be a design rule because there are no other classes in its space and it doesn't decouple and frame modules.

Illustration. We now use an example to demonstrate that each type of relation, or group of types, can form a meaningful DRSpace. Using Titan, DRSpaces can be automatically calculated. All the DSMs shown in the rest of the paper are exported from Titan. For the sake of space, in these DSMs, *inheritance realization*, *aggregation*, *dependency*, and *nested* relations are marked in the cells using *ih*, *rl*, *ag*, *dp*, and *nt* respectively.



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

Figure 1: Package Clustering

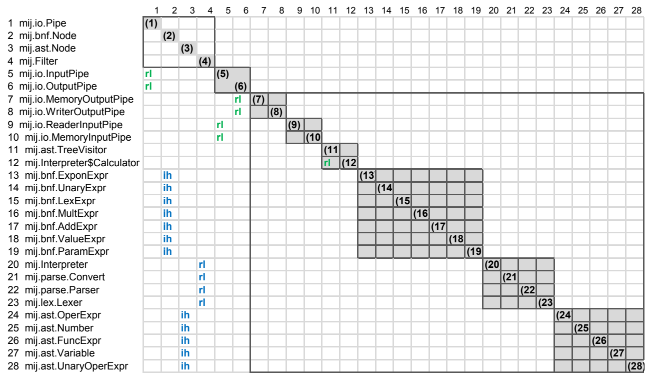
Figures 1 through 5 depict the different modular views of a simple calculator program, recovered from its Java source code. This program supports basic math calculations, such as addition, subtraction, and multiplication. The system is designed using an interpreter pattern with a parser and lexer. It also employs a visitor pattern so that different operations can be done in the same abstract syntax tree (AST). Different components of the system, such as lexer and parser, communicate with each other using a pipe and filter pattern. As a reference, we present Figure 1 to show its package structure, similar to the DSM created by most reverse engineering tools, such as Lattix [1].

Figure 1 shows that this system has five packages, depicted as shaded groups along the diagonal. This program has 36 classes. The DSM shows just 29 elements because the `io` package is collapsed. The cells of the DSM are marked with "x" because it uniformly models all types of relations as *dependency*.

[1]`http://www.lattix.com/`

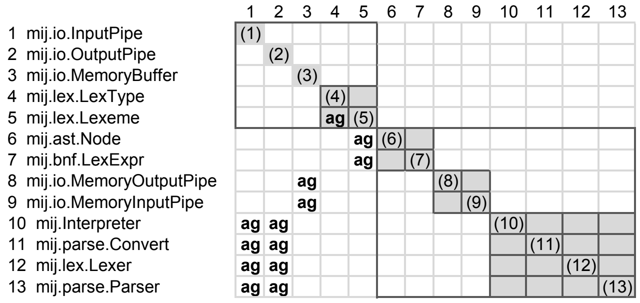
Now we show that the architecture of this system is composed of multi-layer DRSpaces.

1. *Inheritance DRSpace.* Figure 2 depicts the DRSpace that uses inheritance/realization relation as the primary relation. There are three layers in this DRSpace. The first layer l1:(rc1-4) contains four leading classes: `mij.io.Pipe`, `mij.Filter`, `mij.bnf.Node`, and `mij.ast.Node`. They are "leading" because they do not depend on any other class and many other classes depend on them. The second layer contains `mij.io.InputPipe` and `mij.io.OutputPipe`. Because they both realize the same parent class, our ArchDRH algorithm considers them as belonging to the same module. The classes within these two layers decouple the rest of system into 6 independent modules as shown in the third layer of the DSM: m1:(rc7-8), m2:(rc9-10), m3:(rc11-12), m4:(rc13-19), m5:(rc20-23), m6:(rc24-28). It is easy to see that each module has its own meaning. For example, m4 captures all the bnf classes in a module, m6 groups all the ast classes into a module, and m5 contains all the classes that are of type `Filter`. This obviously meaningful modular structure cannot been seen in other views generated by other clustering methods or using other types of relations.



	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
1 mij.io.Pipe	(1)																											
2 mij.bnf.Node		(2)																										
3 mij.ast.Node			(3)																									
4 mij.Filter				(4)																								
5 mij.io.InputPipe	rl				(5)																							
6 mij.io.OutputPipe	rl					(6)																						
7 mij.io.MemoryOutputPipe						rl	(7)																					
8 mij.io.WriterOutputPipe						rl		(8)																				
9 mij.io.ReaderInputPipe					rl				(9)																			
10 mij.io.MemoryInputPipe					rl					(10																		
11 mij.ast.TreeVisitor											(11																	
12 mij.Interpreter$Calculator											rl	(12																
13 mij.bnf.ExponExpr		ih											(13															
14 mij.bnf.UnaryExpr		ih												(14														
15 mij.bnf.LexExpr		ih													(15													
16 mij.bnf.MultExpr		ih														(16												
17 mij.bnf.AddExpr		ih															(17											
18 mij.bnf.ValueExpr		ih																(18										
19 mij.bnf.ParamExpr		ih																	(19									
20 mij.Interpreter				rl																(20								
21 mij.parse.Convert				rl																	(21							
22 mij.parse.Parser				rl																		(22						
23 mij.lex.Lexer				rl																			(23					
24 mij.ast.OperExpr			ih																					(24				
25 mij.ast.Number			ih																						(25			
26 mij.ast.FuncExpr			ih																							(26		
27 mij.ast.Variable			ih																								(27	
28 mij.ast.UnaryOperExpr			ih																									(28)

Figure 2: Inheritance DR Space

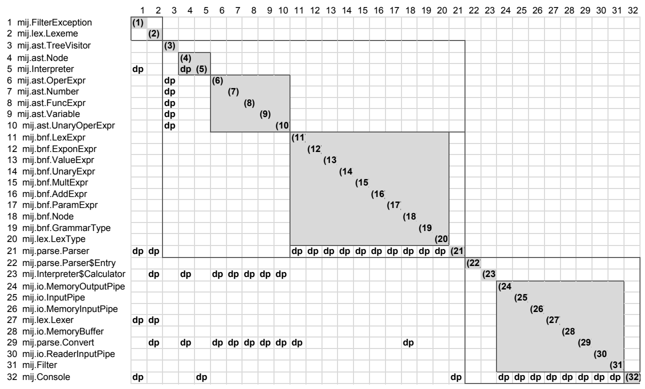


	1	2	3	4	5	6	7	8	9	10	11	12	13
1 mij.io.InputPipe	(1)												
2 mij.io.OutputPipe		(2)											
3 mij.io.MemoryBuffer			(3)										
4 mij.lex.LexType				(4)									
5 mij.lex.Lexeme				ag	(5)								
6 mij.ast.Node					ag	(6)							
7 mij.bnf.LexExpr					ag		(7)						
8 mij.io.MemoryOutputPipe			ag					(8)					
9 mij.io.MemoryInputPipe			ag						(9)				
10 mij.Interpreter	ag	ag								(10)			
11 mij.parse.Convert	ag	ag									(11)		
12 mij.lex.Lexer	ag	ag										(12)	
13 mij.parse.Parser	ag	ag											(13)

Figure 3: Aggregation DR Space

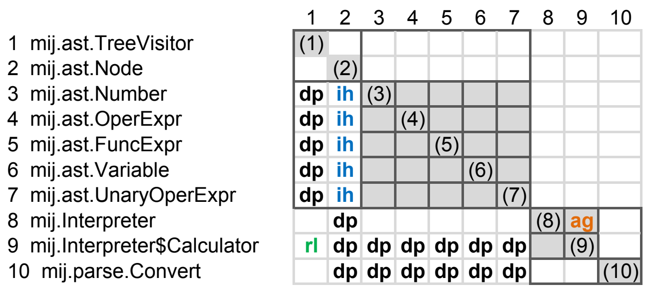
2. *Aggregation DRSpace.* Figure 3 depicts the DRSpace in which the primary relation is aggregation. There are two layers in this DRSpace. The first layer, l1:(rc1-5), contains four modules of leading classes, and the second layer contains three meaningful modules. For example, m1:(rc8-9) is a `MemoryBuffer` module that contains two classes using it; m2:(rc10-13) groups major components such as parser and lexer together because they all communicate through pipes, and thus aggregate, mij.io.InputPipe and mij.io.OutputPipe.

3. *Dependency DRSpace.* Figure 4 depicts the DRSpace with dependency as the primary relation. Completely different from the other two DRSpaces, this DRSpace shows how classes work together to accomplish a function. For example, m:(rc11-20) shows which classes the parser needs in order to accomplish the parsing function.



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

Figure 4: Dependency DR Space



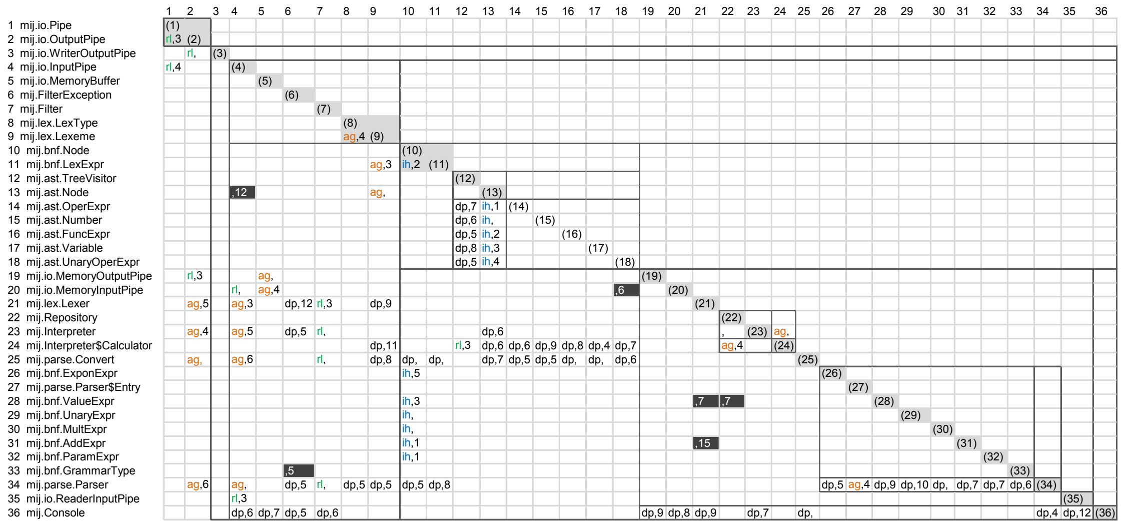
	1	2	3	4	5	6	7	8	9	10
1 mij.ast.TreeVisitor	(1)									
2 mij.ast.Node		(2)								
3 mij.ast.Number	dp	ih	(3)							
4 mij.ast.OperExpr	dp	ih		(4)						
5 mij.ast.FuncExpr	dp	ih			(5)					
6 mij.ast.Variable	dp	ih				(6)				
7 mij.ast.UnaryOperExpr	dp	ih					(7)			
8 mij.Interpreter		dp						(8)	ag	
9 mij.Interpreter$Calculator	rl	dp	dp	dp	dp	dp	dp		(9)	
10 mij.parse.Convert		dp	dp	dp	dp	dp	dp			(10)

Figure 5: Visitor DR Space

4. *Pattern DRSpace.* Figure 5 depicts a DRSpace led by `mij.ast.TreeVisitor`. As we can see, this DRSpace captures the overall structure of the classes that participate in the visitor pattern. The key design rules of this pattern include `mij.ast.TreeVisitor`, acting as the role of visitor interface, and `mij.ast.Node`, acting as the element interface. The classes in the module m:(rc3-7) contains all the concrete elements of the pattern. These classes are all subclasses (the "ih" relation) of `mij.ast.Node`, which fills the element role in the visitor pattern. They all accept the visitor interface, and pass themselves to the visitor interface (the "dp" relation), as required by the pattern. The `Calculator` class takes the concrete visitor role through the realization ("rl") relation to `mij.ast.Treevisitor`.

5. *Hybrid DRSpace.* Figure 6 depicts a DRSpace in which the DRH is produced using all three types of structural relations as primary ones. As we can see, all the interesting and meaningful modular structures that can be observed from previous DRSpaces are all mixed up, and become less obvious. The DRH now has many more nested layers.

In this DRSpace, we also choose evolutionary coupling as the secondary relation. For example, cell c:(r13, c4) has number 12, meaning that mij.ast.Node and mij.io.InputPipe changed together 12 times in the revision history. This cell has dark background and white font to indicate that there



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
1 mij.io.Pipe	(1)																																			
2 mij.io.OutputPipe	rl,3	(2)																																		
3 mij.io.WriterOutputPipe		rl,	(3)																																	
4 mij.io.InputPipe	rl,4			(4)																																
5 mij.io.MemoryBuffer					(5)																															
6 mij.FilterException						(6)																														
7 mij.Filter							(7)																													
8 mij.lex.LexType								(8)																												
9 mij.lex.Lexeme								ag,4	(9)																											
10 mij.bnf.Node										(10)																										
11 mij.bnf.LexExpr									ag,3	ih,2	(11)																									
12 mij.ast.TreeVisitor												(12)																								
13 mij.ast.Node				,12					ag,				(13)																							
14 mij.ast.OperExpr												dp,7	ih,1	(14)																						
15 mij.ast.Number												dp,6	ih,		(15)																					
16 mij.ast.FuncExpr												dp,5	ih,2			(16)																				
17 mij.ast.Variable												dp,8	ih,3				(17)																			
18 mij.ast.UnaryOperExpr												dp,5	ih,4					(18)																		
19 mij.io.MemoryOutputPipe		rl,3			ag,														(19)																	
20 mij.io.MemoryInputPipe				rl,	ag,4													,6		(20)																
21 mij.lex.Lexer		ag,5		ag,3		dp,12	rl,3		dp,9												(21)															
22 mij.Repository																						(22)														
23 mij.Interpreter		ag,4		ag,5		dp,5	rl,						dp,6									,	(23)	ag,												
24 mij.Interpreter$Calculator									dp,11			rl,3	dp,6	dp,6	dp,9	dp,8	dp,4	dp,7				ag,4		(24)												
25 mij.parse.Convert		ag,		ag,6			rl,		dp,8	dp,	dp,		dp,7	dp,5	dp,5	dp,	dp,	dp,6							(25)											
26 mij.bnf.ExponExpr										ih,5																(26)										
27 mij.parse.Parser$Entry																											(27)									
28 mij.bnf.ValueExpr										ih,3											,7	,7						(28)								
29 mij.bnf.UnaryExpr										ih,																			(29)							
30 mij.bnf.MultExpr										ih,																				(30)						
31 mij.bnf.AddExpr										ih,1											,15										(31)					
32 mij.bnf.ParamExpr										ih,1																						(32)				
33 mij.bnf.GrammarType						,5																											(33)			
34 mij.parse.Parser		ag,6		ag,		dp,5	rl,	dp,5	dp,5	dp,5	dp,8															dp,5	ag,4	dp,9	dp,10	dp,	dp,7	dp,7	dp,6	(34)		
35 mij.io.ReaderInputPipe				rl,3																															(35)	
36 mij.Console				dp,6	dp,7	dp,5	dp,6												dp,9	dp,8	dp,9		dp,7		dp,									dp,4	dp,12	(36)

Figure 6: DR Space with History

are no structural relations between these classes. The content in cell c:(r23, c2) is "ag,4", meaning that mij.Interpreter aggregates mij.io.OutputPipe, and they changed together 4 times in the revision history. As an illustrative example, the history of this system is faked. In real systems, as shown in Section 5, the dark cells indicate modularity violations.

In summary, it is clear that the architecture of this small system can be viewed as a set of multi-layer DRSpaces. Each DRSpace reflects a unique aspect of the architecture that cannot be captured using any other types of relations or clustering methods.

4. TOOL SUPPORT

In this section, we briefly introduce our tool, Titan, that supports the creation and visualization of DRSpaces. All the figures and tables in the paper were created using data exported from Titan.

Titan accepts DSM files, with extension .dsm, and clustering files, with extension .clsx, as input. A .dsm file captures pair-wise relations among classes. For a "structure DSM", the number in a cell is used to represent different types of relations. So far our tool processes inheritance, realization, dependency, nested, and evolutionary coupling relations. For a "history DSM", the number in a cell represents the number of times the two classes changed together (where "changed together" means that both classes were involved in the same commit), which is called *co-change frequency.* One .dsm file can be associated with multiple clustering files, each representing a different way the DSM can be clustered.

Figure 7 shows a snapshot of Titan's GUI. Similar to commercial tools with DSM-based user interfaces, Titan has a tree structure view (the top right part) and a DSM view (the lower right part).

The Tree View. When a structure DSM file is first opened, the tree view renders classes randomly. After the user loads a clustering file, the tree view is redrawn to reflect the given structure. Using the tree view, the user can expand, collapse, group, and ungroup classes, and the DSM view will be updated when the user clicks the redraw button.

The user can also cluster the DSM using an algorithm by choosing the Clusters menu item. As shown in the figure, currently Titan supports the following clustering methods:

1. Package Cluster. The DSM will be clustered based on the project's package and namespace structure, as supported by other commercial tools.

2. ArchDRH Cluster. This is the clustering method we employed to generate DRSpaces in this paper. The algorithm is described in our prior work [4].

3. ArchDRH+ACDC Cluster and ArchDRH+Bunch Cluster. As introduced in [4], each inner module of a DSM first framed using ArchDRH can be further clustered using other algorithms. We do not further discuss these functions here.

The user can also view partial DSMs in two ways. If a tree node (folder) is selected, the SubSystem button will be activated. Clicking it creates a new GUI instance showing just the subspace within the chosen folder.

If a DSM is clustered using ArchDRH, and at least one tree leaf (class) is selected, the Split button will be activated. Clicking it creates a new window that contains only the classes in the DRSpace led by the selected class(es). All the DRSpaces in this paper are generated this way.

The window created by clicking the Split or SubSystem button is exactly the same as the original GUI so that the user can treat the subspace as an entirely independent design space, which can be further manipulated or split.

The DSM View. In the DSM view, each group of classes are colored using a dark background. A nested group always has a darker background than the outside group. The diagonal line is labeled with the index of the class. The relation displayed in the cells can be controlled using the check-boxes located at the left lower corner of the GUI.

The user can check and uncheck any listed relation, or any combination of them, to control the display. Once the relation types are selected, clicking the clustering menu item will cluster the DSM using the selected relations as primary relations. That is how we generated the aggregation, inheritance, and dependency DRSpaces, for example.

To show the evolution coupling together with structure relations, the user first loads a history DSM, and then checks the history checkbox. The cells of the DSM will then display how many times each pair of classes have changed together in the history. For example, the DSM in Figure 7 displays aggregation, nesting, and history relations. The cell c:(r8,c2) has: "aggregate,11", meaning that `JDBCStopCommand` aggregates `JDBCEntityBridge`, and they changed together 11 times.

If two classes do not have any structural relation but still changed together, the cell will have a red background. For example, cell c:(r2,c1) shows that although `JDBCEntityBridge` and `RelationDataManager` have no structural relation, they changed together 35 times.

The user can control the threshold of the co-change frequency to be displayed by checking the Threshold box and filling a number in a pop-up window. In the DSM of Figure 7, the threshold is set to 10, so that only cells with co-change frequency of 11 or more are displayed.

To summarize the key differences between Titan and other commercial DSM tools: Titan allows the user to choose any combination of relation types, and to cluster the DSM based on the selected primary relation(s) only. Moreover, it supports the display of evolutionary coupling data, together with structure relations, so that their discrepancies can be visualized.

5. EVALUATION

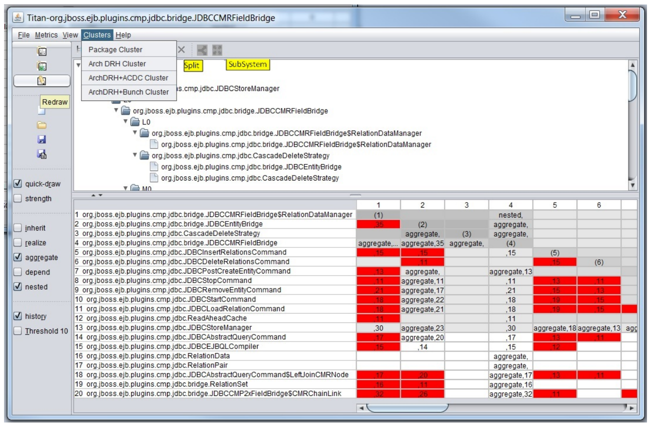
As our first evaluation of the usefulness of DRSpaces, we determine whether they can provide insights on bug location. We explore the following research questions:

RQ1: Is it true that if a design rule is error-prone, then the files contained in its DRSpace are also error-prone?

If the answer is yes, it means that (1) these error-prone files within the same DRSpace should be considered together because they are structurally related, even though these files may not depend on each other directly; (2) these design rules should be given higher priority in terms of bug fixing (and, potentially, refactoring) given their significant impact.

RQ2: Are most error-prone files concentrate in a few DR spaces?

If the answer is yes, this implies that even if a system has hundreds of error-prone files, we should be able to understand their relationships by just looking into a few DRSpaces. Furthermore, this implies that these error-prone files, or error-prone file groups, are not isolated—they are structurally related—and thus should be analyzed and fixed as a group.



```
                    s.cmp.jdbc.JDBCStoreManager
    ▼ org.jboss.ejb.plugins.cmp.jdbc.bridge.JDBCCMRFieldBridge
      ▼ L0
        ▼ org.jboss.ejb.plugins.cmp.jdbc.bridge.JDBCCMRFieldBridge$RelationDataManager
            org.jboss.ejb.plugins.cmp.jdbc.bridge.JDBCCMRFieldBridge$RelationDataManager
        ▼ org.jboss.ejb.plugins.cmp.jdbc.CascadeDeleteStrategy
            org.jboss.ejb.plugins.cmp.jdbc.bridge.JDBCEntityBridge
            org.jboss.ejb.plugins.cmp.jdbc.CascadeDeleteStrategy
      ▼ M0
```

	1	2	3	4	5	6	
1 org.jboss.ejb.plugins.cmp.jdbc.bridge.JDBCCMRFieldBridge$RelationDataManager	(1)			nested,			
2 org.jboss.ejb.plugins.cmp.jdbc.bridge.JDBCEntityBridge	,35	(2)		aggregate,			
3 org.jboss.ejb.plugins.cmp.jdbc.CascadeDeleteStrategy		aggregate,	(3)	aggregate,			
4 org.jboss.ejb.plugins.cmp.jdbc.bridge.JDBCCMRFieldBridge	aggregate,...	aggregate,35	aggregate,	(4)			
5 org.jboss.ejb.plugins.cmp.jdbc.JDBCInsertRelationsCommand	,15	,15		,15	(5)		
6 org.jboss.ejb.plugins.cmp.jdbc.JDBCDeleteRelationsCommand		,11			,15	(6)	
7 org.jboss.ejb.plugins.cmp.jdbc.JDBCPostCreateEntityCommand	,13	aggregate,		aggregate,13			
8 org.jboss.ejb.plugins.cmp.jdbc.JDBCStopCommand	,11	aggregate,11		,11	,13	,11	
9 org.jboss.ejb.plugins.cmp.jdbc.JDBCRemoveEntityCommand	,21	aggregate,17		,21	,15	,13	
10 org.jboss.ejb.plugins.cmp.jdbc.JDBCStartCommand	,18	aggregate,22		,18	,19	,15	
11 org.jboss.ejb.plugins.cmp.jdbc.JDBCLoadRelationCommand	,18	aggregate,21		,18	,19	,15	🟥
12 org.jboss.ejb.plugins.cmp.jdbc.ReadAheadCache	,11			,11			
13 org.jboss.ejb.plugins.cmp.jdbc.JDBCStoreManager	,30	aggregate,23		,30	aggregate,18	aggregate,13	agg
14 org.jboss.ejb.plugins.cmp.jdbc.JDBCAbstractQueryCommand	,17	aggregate,20		,17	,13	,11	
15 org.jboss.ejb.plugins.cmp.jdbc.JDBCEJBQLCompiler	,15	,14		,15	,12		
16 org.jboss.ejb.plugins.cmp.jdbc.RelationData				aggregate,			
17 org.jboss.ejb.plugins.cmp.jdbc.RelationPair				aggregate,			
18 org.jboss.ejb.plugins.cmp.jdbc.JDBCAbstractQueryCommand$LeftJoinCMRNode	,17	,20		aggregate,17	,13	,11	
19 org.jboss.ejb.plugins.cmp.jdbc.bridge.RelationSet	,16	,11		aggregate,16			
20 org.jboss.ejb.plugins.cmp.jdbc.bridge.JDBCCMP2xFieldBridge$CMRChainLink	,32	,26		aggregate,32	,11		🟥

Figure 7: The Graphical User Interface of Titan

RQ3: By combining information about evolution and structure coupling, can we get more insight into architectural problems? Can this help us find not just the locations of errors, but also the reasons for them?

So far, the prevailing bug-location research focuses on *where* the bugs are, rather than *why* these locations are error-prone. Although error-proneness can be caused by many reasons, our recent work has shown that structural errors can be an important cause of bugginess. Here we explore whether the combination of different types of DRSpaces can shed light on the structural problems among these error-prone files.

5.1 Subjects

We choose three large-scale open source projects as our evaluation subjects: JBoss[2]–a Java application server, Hadoop Common[3]–the common utilities supporting a suite of distributed computation tools, and Eclipse Java Development Tools (JDT)[4]–a core AST analysis toolkit in the Eclipse IDE. For each project, we choose one target release to analyze its DRSpaces. We made sure, however, that there were at least 10 releases *before* the target release so that we could produce history DSMs and identify error-prone files. The target project releases, evolution history time span, and the number of releases before the target can be found in Table 1.

Similar to our prior work [22], we generated history DSMs using revision and issue tracking histories. Using Hadoop as an example, we investigated its SVN repository to extract transactions. In Table 1, we present data regarding the number of files (*#Files*), releases (*#Rel.*), transactions (*#Trans.*), and issues (*#Issues*) we studied. We removed commits with only one file or more than 30 files, because they either do not contribute to evolution coupling or they introduce substantial "noise" in the data, such as bulk changes to files to update license information.

Table 1: Subject System Information

Subject	History	#Files	#Rel.	#Trans.	#Issues
JBoss3.2.4	Apr 00-Jun 04	762	11	6458	550
Hadoop0.15	Feb 06-Oct 07	692	15	3001	490
Eclipse3.0.2	May 01-Mar 05	3704	10	27806	3458

5.2 DRSpace Error-Proneness

To answer the first research question, for each target release, we first ranked all the files by the number of times they were involved in bug fixes. As others have observed in bug prediction research, the more often a file is involved in a bug fix, the more error-prone it is. For each of the 30 most error-prone files in each project, we used Titan to determine its DRSpace. If the size of a DRSpace is small, it means that this file does not have high impact. We then rank the size of the DRSpaces of the 30 most error-prone files, and we only consider DRSpaces with at least 10 files, which we call *Top DRSpaces*.

Table 2 summarizes the status of Top DRSpaces for each project. For example, in JBoss, 9 out of the 30 most error-prone files *lead* a DRSpace with at least 10 files. In both

[2]`http://www.jboss.org`

[3]`http://hadoop.apache.org/`

[4]`http://www.eclipse.org/jdt/`

Table 2: Top DRSpaces

#TopDRS	Avg. in Bug2		Avg. in Bug5		Avg. in Bug10	
	dsb	bsc(#)	dsb	bsc(#)	dsb	bsc(#)
JBoss(9)	62%	10%(206)	50%	13%(129)	18%	30%(23)
Hadoop(11)	67%	11%(187)	53%	14%(106)	47%	18%(68)
Eclipse(11)	31%	7%(291)	15%	9%(98)	9%	13%(36)

Hadoop and Eclipse, this number is 11. Because there is no obvious threshold on the number of bug fixes to determine if a file is error-prone, for each project we use the following conventions:

- We define a *bug space, BugN,* as the set of files with at least *N* bugs. The size of a bug space is the number of files within it. In this research we chose values of 2, 5, and 10 for *N*, resulting in three bug spaces: Bug2, Bug5, and Bug10.
- We next define *design space bugginess, dsb*: If a DRSpace has m files, and n of them are within a bug space, we define n/m as the *design space bugginess* of the DRSpace, represented as *dsb* in the tables.
- And we define *bug space coverage, bsc*: If a DRSpace has n files in BugX, then the bug space coverage of the DRSpace with respect to BugN is $n/size(BugN)$.

For example, in JBoss (Table 3), there are a total of 206 files with 2 or more bug fixes (Bug2), 129 files with 5 or more bug fixes (Bug5), and 23 files with 10 or more bug fixes (Bug10).

Table 3: JBoss's Top DRSpaces

#DRS	#Bug (Rank)	#Bug2: 206		#Bug5: 129		#Bug10: 23	
		$\#(dsb)$	bsc	$\#(dsb)$	bsc	$\#(dsb)$	bsc
dr1:18	27(2nd)	10(56%)	5%	7(39%)	5%	4(22%)	17%
dr2:56	21(4th)	32(57%)	16%	26(46%)	20%	10(18%)	43%
dr3:28	18(7th)	22(79%)	11%	18(64%)	14%	5(18%)	22%
dr4:43	14(12th)	27(63%)	13%	20(47%)	16%	5(12%)	22%
dr5:76	11(21st)	43(57%)	21%	35(46%)	27%	12(16%)	52%

Consider design rule dr2, `org.jboss.ejb.Container`, whose data is shown in the second row of Table 3. It is the 4th most error-prone file in JBoss and leads a DRSpace with 56 files. Of these 56 files, 32 of them have more than 2 bug fixes. Thus the *bsc* of dr2 in JBoss is 16% (32/206) with respect to Bug2. Similarly we calculate the *bsc* for Bug5 as 20% (26 out of 129), and for Bug10 as 43% (10 out of 23). Furthermore, the *dsb* of dr2 is 57% with respect to Bug2 (32 out of the 56 files within dr2 are in Bug2), 46% with respect to Bug5 (26 of the 56 files are in Bug5) and 18% with respect to Bug10 (10 of the 56 files are in Bug10).

Table 3 lists the first 5 (out of 9) most error-prone DRSpaces in JBoss. Consider dr5, `BeanMetaData`; it has the largest DRSpace with 76 files. Within these 76 files, 43 (57%) have more than 2 bug fixes, 35 (46%) have more than 5 bug fixes, and 12 (16%) have more than 10 bug fixes. These 12 files cover more than 50% of the all the files in Bug10. This result shows that not only is `BeanMetaData` itself bug-prone—it has 11 bug fixes, and is ranked the 21st overall in terms of error-proneness—but a substantial part of the DRSpace it is leading is also error-prone. The other 4 DRSpaces show similar results: their *dsb* values for Bug2, Bug5, and Bug10 range from 56%-79%, 39%-64%, and 12%-22% respectively.

Table 2 shows the average *dsb* and *bsc* values for each project. The first line of the table shows that in JBoss, there are 9 DRSpaces led by the most error-prone files. On average, within each DRSpace, 62% of the files have more than 2 bug fixes, 50% of them have more than 5 bug fixes, and 18% have more than 10 bug fixes. Although the *dsb* decreases with higher threshold of bugginess, the *bsc* increases. For example, the average bug space coverage of a JBoss DRSpace in Bug10 is 30%, meaning that on average, each top DRSpace in JBoss contains about one-third of the most error-prone files (with 10 or more bug fixes each).

Interestingly, Table 2 also shows that the *bsc* and *dsb* for Eclipse DRSpaces are much lower than the other two projects. For example, its design space bugginess for Bug5 (15%) is only about one-third of the other two projects (50% and 53%). To explore why Eclipse is special, we calculate the *dependency density* of each top DRSpace of each project. The dependency density is the number of dependencies within a DRSpace divided by the square of its size. The higher the density, the more tightly coupled the files within the DRSpace. The result shows that the average density for JBoss and Hadoop DRSpaces are 12% and 15% respectively, while the density for Eclipse is only 7%. Now the results become intuitive: the more highly coupled the files within a DRSpace, the more that the DRSpace can be influenced by error-prone design rules and neighbor files.

In summary, these results show that if a file is error-prone and leading a highly coupled DRSpace, then a significant portion of the DRSpace is also error-prone. We thus call a DRSpace led by an error-prone file an *error-prone DRSpace*.

5.3 Error-Prone DRSpace Coverage

Now we investigate the second research question. A project may have hundreds of error-prone files. Can they be captured by a much smaller number of DRSpaces led by error-prone design rules? We explore this problem by answering two complementary questions: 1) How many DRSpaces are needed to maximally cover Bug2, Bug5, and Bug10? 2) How large a bug space can the top 10 largest DRSpaces cover?

To answer these questions, we ranked all the DRSpaces with at least 10 files led by error-prone files based on their non-overlapping bug space coverages. The results, summarized in Table 4, answer the first question. Take JBoss for example: the first 15 DRSpaces cover 66% of the Bug2 space; the first 9 DRSpaces cover 57% of the Bug5 space, and the first 3 DRSpaces cover 78% of Bug10 space. We never reach 100% coverage because we are only considering DRSpaces with at least 10 files; the other error-prone files are distributed in smaller DRSpaces.

Table 4: Minimal Error Space Coverage

Proj	Bug2		Bug5		Bug10	
	#drs	bs%(#)	#drs	bs%(#)	#	bs%(#)
JBoss	15	66%(206)	9	57%(129)	3	78%(23)
Hadoop	23	77%(187)	18	82%(106)	12	88%(68)
Eclipse	38	90%(291)	13	92%(98)	6	92%(36)

To answer the second question, we list the *bsc* of the first 5 and 10 DRSpaces in Table 5. This table shows that the top 5

DRSpaces of any of the three projects, within any of Bug2, Bug5, or Bug10, can capture more than half of the error-prone files within each bug space. The top 10 DRSpaces can cover from 57% to 92% of a bug space. Interestingly, by looking at just the top 5 DRSpaces we do nearly as well: we can cover from 52% to 89% of a bug space. In summary, the answer to the second research question is yes: indeed, most error-prone files are concentrated in just a few DRSpaces.

Table 5: Top Space Bug Coverage

Proj	Bug2		Bug5		Bug10	
	Top 5	Top 10	Top 5	Top 10	Top 5	Top 10
JBoss	57%	64%	52%	57%	78%	78%
Hadoop	59%	67%	68%	75%	76%	85%
Eclipse	71%	78%	83%	89%	89%	92%

5.4 The Structure of Error-Prone Spaces

The results reported in the previous sections imply that large numbers of error-prone files belong to the same few DRSpaces. The question is whether these DRSpaces can provide insights into the *reasons* why these files are error-prone. For example, Figure 8 depicts the DRSpace led by `JDBCCMRFieldBridge`. We obtained this DRSpace by first clustering the overall DSM using ArchDRH. We then chose this file in Titan, and clicked the "Split" button.

This file has 27 bug fixes, and is ranked as the 2nd most error-prone in the project. This Figure shows its DRSpace with only the aggregation relation, together with evolutionary coupling. A pair of files that have no structural relation but only evolutionary coupling are shown as dark background cells with white font. The threshold of evolutionary coupling is set to 10.

First, the large number of dark cells indicates that there are many modularity violations. These violations can be separated into two categories: 1) the files whose names include `Command` are always changed together; 2) the `RelationDataManager` always changed together with these `Command` files. Although it is not possible to know why this file is error-prone, this DSM reveals obvious structural issues that violate well-known design principles.

First, consider l:(rc5-11), a layer containing all the command classes, and the `JDBCStoreManager` class. The latter aggregates all the 7 command classes, and is aggregated by 4 of them. This cyclical aggregation relation obviously violates good design principles. It seems that the developers intended to apply a command pattern, but the DSM does not reveal a valid command pattern structure where the client (which in this case seems to be `JDBCStoreManager`), should only depend on an abstract command interface, rather than on concrete commands. This DSM also shows another aggregation cycle in rc(1-4).

Second, it is not obvious why concrete commands always change together with `RelationDataManager` even though there is no structural relation between them. It seems that `RelationDataManager` shares some "secrets" with these command classes.

We studied all the error-prone DRSpaces of each project, and observed that they often exhibit the following problems:

1. Aggregation/dependency cycles: once we select just aggregation or dependency as the primary relation, we found that many error-prone DRSpaces exhibit multiple aggregation or dependency cycles. For example, in the DRSpace led by `metadata.JDBCEntityMetaData`, there are 4 aggregation cycles. In one of them, `JDBCEntityBridge` (71st most error-prone) and `JDBCCMRFieldBridge`(2nd most error-prone) aggregate each other!

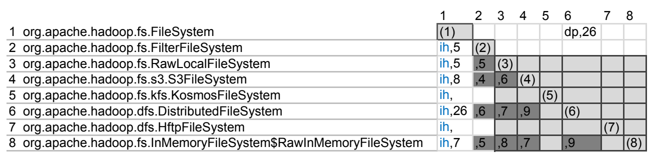
Different from other tools that can detect cyclic relations, our tools show the *penalty* of such relations. For example, our tools show that `JDBCEntityBridge` and `JDBCCMRFieldBridge` changed together 35 times. Furthermore, it should be noted that not all cyclical relations are harmful. `CascadeDeleteStrategy` and `JDBCCMRFieldBridge` also aggregate each other, but they never changed together, and `CascadeDeleteStrategy` has no bug fixes. A tool that simply identifies cyclic relations can not distinguish between harmful and harmless cases.

As another example, `FSNamesystem` has 190 bug fixes and is ranked as the number 1 most error-prone file in Hadoop. From its DRSpace with 17 files, we can see that `FSNamesystem` is involved in a dependency cycle with 11 files, and an aggregation cycle with 7 elements.

2. Problematic inheritance hierarchy. Inheritance issues manifest themselves in different ways, including parent and children frequently changing together, a client inheriting a parent class while aggregating its child, a parent depending on one of its children, etc.

Figure 9 shows the inheritance DRSpace of `FileSystem` (ranked the 13th most error-prone). We obtained this space by clustering the DRSpace using inheritance as the primary relation and then selecting dependency as the secondary relation when we saw that `FileSystem` depends on one of its children, `DistributedFileSystem`. After choosing evolutionary coupling as another secondary relation, we can see that these two files changed together 26 times, while the other elements within the same space changed together 5 to 10 times.

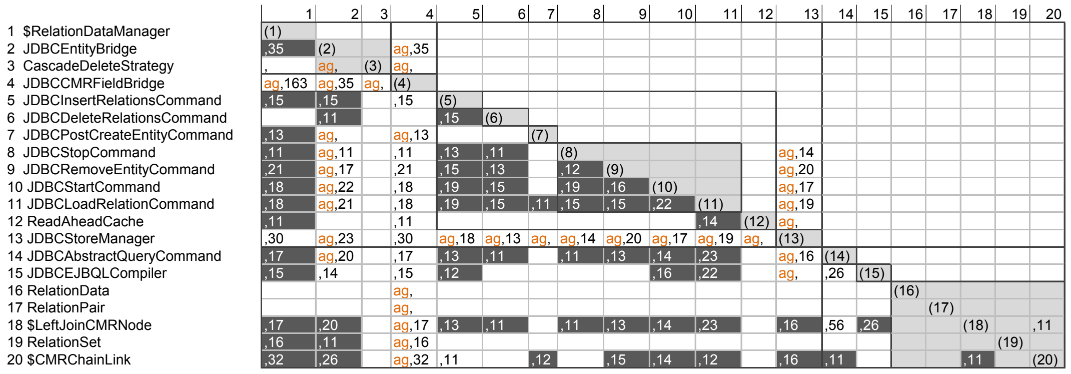
Figure 9 also depicts an example where problematic co-changes may *not* be modularity violations: Titan did not mark the relation between `DistributedFileSystem` and `FileSystem` as a violation because they do have structural relations. However, the fact that they changed together unusually frequently and that they have both inheritance and dependency relations, indicates that there is something wrong.



		1	2	3	4	5	6	7	8
1	org.apache.hadoop.fs.FileSystem	(1)					dp,26		
2	org.apache.hadoop.fs.FilterFileSystem	ih,5	(2)						
3	org.apache.hadoop.fs.RawLocalFileSystem	ih,5	,5	(3)					
4	org.apache.hadoop.fs.s3.S3FileSystem	ih,8	,4	,6	(4)				
5	org.apache.hadoop.fs.kfs.KosmosFileSystem	ih,				(5)			
6	org.apache.hadoop.dfs.DistributedFileSystem	ih,26	,6	,7	,9		(6)		
7	org.apache.hadoop.dfs.HftpFileSystem	ih,						(7)	
8	org.apache.hadoop.fs.InMemoryFileSystem$RawInMemoryFileSystem	ih,7	,5	,8	,7		,9		(8)

Figure 9: Hadoop FileSystem Inherit DRSpace

As another example, JobTracker in Hadoop is ranked most error-prone with 165 bug fixes. In its inheritance DRSpace, after choosing dependency and aggregation as secondary relations, we saw that JobTracker depends on conf. Configuration, and aggregates mapred.JobConf, which, in turn, is a child of conf.Configuration. Both of mapred.JobConf and conf.Configuration are highly buggy, ranking 21st and 26th respectively. They both lead large error-prone DRSpaces, with 76 and 54 files respectively. Since JobTracker either depends on or aggregates them, it is not surprising that it is the most error-prone file of the entire project.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1 $RelationDataManager	(1)																			
2 JDBCEntityBridge	,35	(2)		ag,35																
3 CascadeDeleteStrategy	,	ag,	(3)	ag,																
4 JDBCCMRFieldBridge	ag,163	ag,35	ag,	(4)																
5 JDBCInsertRelationsCommand	,15	,15		,15	(5)															
6 JDBCDeleteRelationsCommand		,11			,15	(6)														
7 JDBCPostCreateEntityCommand	,13	ag,		ag,13			(7)													
8 JDBCStopCommand	,11	ag,11		,11	,13	,11		(8)					ag,14							
9 JDBCRemoveEntityCommand	,21	ag,17		,21	,15	,13		,12	(9)				ag,20							
10 JDBCStartCommand	,18	ag,22		,18	,19	,15		,19	,16	(10)			ag,17							
11 JDBCLoadRelationCommand	,18	ag,21		,18	,19	,15	,11	,15	,15	,22	(11)		ag,19							
12 ReadAheadCache	,11			,11							,14	(12)	ag,							
13 JDBCStoreManager	,30	ag,23		,30	ag,18	ag,13	ag,	ag,14	ag,20	ag,17	ag,19	ag,	(13)							
14 JDBCAbstractQueryCommand	,17	ag,20		,17	,13	,11		,11	,13	,14	,23		ag,16	(14)						
15 JDBCEJBQLCompiler	,15	,14		,15	,12					,16	,22		ag,	,26	(15)					
16 RelationData				ag,												(16)				
17 RelationPair				ag,													(17)			
18 $LeftJoinCMRNode	,17	,20		ag,17	,13	,11		,11	,13	,14	,23		,16	,56	,26			(18)		,11
19 RelationSet	,16	,11		ag,16															(19)	
20 $CMRChainLink	,32	,26		ag,32	,11		,12		,15	,14	,12		,16	,11				,11		(20)

Figure 8: JBoss JDBCCMRFieldBridge DRSpace

Although we can not enumerate all possible problems in all the error-prone DRSpaces (there are many possible combinations of relation types and DRSpaces), we observe that indeed each error-prone DRSpace has more than one type of structural issue. Again, we consider them as problematic because they violate common design principles, and the files involved in these structural problems are both highly error- and change-prone.

Shared Secrets. By displaying evolutionary coupling together with structural DRSpaces, we were able to identify large numbers of co-changes among files with neither structural relations nor obvious structural problems. We hypothesize that this indicates that there are "shared secrets" that cannot be captured by structural information alone. In our future work, we will investigate this hypothesis.

5.5 Result Summary

Now we can positively answer the three research questions posed at the beginning of the section.

RQ1: Indeed, if a file is error-prone itself, and leading a non-trivial and highly coupled DRSpace, then a significant number of the files within its DRSpace are also error-prone.

RQ2: Although each project may have hundreds of error-prone files, these files are often captured by just a few DRSpaces. In all projects studied with all three types of bug spaces, the 5 largest DRSpaces always captured more than half of the files in the bug space.

RQ3: By examining DRSpaces with different types of primary and secondary relations, we found that all error-prone DRSpaces also have more than one structural problem that violate commonly accepted design principles. The most prominent problems include large dependency or aggregation cycles, problematic inheritance hierarchies, the aggregation or inheritance of highly error-prone files, and the existence of potential shared secrets.

6. DISCUSSION

In this section, we discuss the threats to validity of our evaluation as well as our planned future work.

Threats to Validity. Our evaluation is subject to *internal* threats to validity. We chose several thresholds purely based on our observations. For example, we only consider a DRSpace with at least 10 files because we observed that the smaller the DRSpace led by an error-prone file, the larger the percentage of files within the DRSpace that are also error-prone. Consider DRSpaces with 10 or fewer files only, the average design space bugginess of these small DRSpaces in JBoss and Hadoop are 68% and 73% respectively, where the smaller the size, we see the highest percentages. This is intuitive because in smaller DRSpaces, the files are more closely related and thus impacted by each other, and their structural problems are relatively easy to directly identify. The larger the DRSpace, the more indirect the relationships among the files (in general), and thus their structural problems will be more complex and subtle.

Other important thresholds we choose include the sizes of bug spaces, We choose 2, 5, and 10 based on the observation that when the threshold reaches 10, the number of DRspaces with more than 10 files is already very small. Both Hadoop and Eclipse have 11 such DRSpaces, and JBoss only has 7. As a result, we believe that 10 is a reasonable threshold, and that Bug2, Bug5, Bug10 reflect DRSpaces with low, medium and high error-proneness across all three projects.

We used the threshold of 30 most error-prone files because we observed that, in all three projects, the bug ranking of files leading the largest DRSpaces are within this scope. For example, in Hadoop, the file `BeanMetaData` is the 21st error-prone, and leads the largest DRSpace of size 76. Our results may, however, be impacted if we choose different thresholds for different projects.

In the data reported thus far, we used all the history for each project to calculate evolutionary coupling and bug proneness. Our prior work [21] showed that recent history has a different impact than more distant history. To determine the impact of history we recalculated all the data reported here based on just the most recent 5 releases of each project. This analysis showed that the top DRSpaces and bug ranking order of their leading files are somewhat different, but the general conclusions are exactly the same: a significant part of the DRSpace led by an error-prone file is also error-prone, and a small number of DRSpaces can cover most of the bug spaces in consideration.

Our evaluation is also subject to several *external* threats. First, as with other history-based bug prediction work, we link a bug with a file by searching developers' commit messages when they submit changes to a file, trying to find bug IDs associated with the commit. However, as prior work [1] has pointed out, since there is no guarantee that developers always report which commits are fixing which bugs, the bug space we considered may be biased. The second threat comes from the subject projects we chose. We only studied 3 open source projects, all of which are written in Java.

The results could be different for closed-source industrial projects and for projects implemented using other object-oriented programming languages.

Future Work. We envision that DRSpaces can be useful in many ways. First, we have observed the potential of using DRSpaces to identify design patterns. We observe that a design pattern typically leads its own design space. The visitor pattern in Figure 5 is an example. We plan to investigate how to leverage DRSpaces for design pattern *detection*, and compare the results with other existing design pattern detection techniques.

Our prior work [18, 22] has shown that modularity violations usually reveal architecture problems or shared secrets. While the work reported in this paper investigated the relation between DRSpaces and error-proneness, one interesting future research direction is to study the relation between DRSpaces and change-proneness, and architecture violation proneness. We also plan to investigate the possibility of leveraging DRSpaces to detect architecture violations, design pattern degradation, and implementation errors that violate the designed modular structure, using both open source and real industrial projects.

7. RELATED WORK

Our work is related to considerable prior research.

Design Structure Matrix Tools. There are several commercial DSM tools available, such as Lattix, Structure 101,[5] and NDepend.[6] These tools also reverse engineer DSMs from source code. Sangal et al. [17] reported a study of using Lattix to identify dependency violations. However, this required manually specifying which classes should not depend on which other classes.

Unlike these DSM tools, Titan integrates structural relations and evolutionary coupling, and supports ArchDRH clustering to reveal multiple DRSpaces, by flexibly choosing any combination of primary and secondary relations.

Metrics and Defects. Using dependency structure to locate software defects has been widely studied, as exemplified by the work of Selby and Basili [19]. Researchers have proposed various metrics to predict failure proneness from coupling measures, such as the work of Chidamber and Kemerer [6]. The relation between evolutionary coupling [9] and error-proneness has also been widely studied. For example, Cataldo et al. [5] reported a strong correlation between change coupling density and failure proneness. Fluri et al. [8] also reported that a great deal of evolutionary coupling is not captured by structural dependencies. Ostrand et. al. [16] reported that file size was the most significant factor influencing the number of faults. Nagappan et. al. [15] demonstrated that complexity metrics can be successful bug predictors. They also claimed there was no single best set of metrics for all projects. In their investigation of network measures such as closeness, Zimmermann and Nagappan [24] reported that network measures were capable of predicting twice as many defects as complexity measures.

By contrast, our study revealed that most error-prone files can be captured by a few DRSpaces led by error-prone design rules, and how these error-prone files influence each other structurally, suggesting how these files should be changed.

Architecture Recovery. Several algorithms have been proposed to recover software modular structure from source code. The most representative ones include Bunch [13], ACDC [20], and LDA [10]. Bunch is a coupling-and-cohesion based clustering method proposed by Mitchell and Mancoridis [13]. It groups highly interdependent nodes of a dependency graph into one subsystem and separates independent nodes into different subsystems, to maximize cohesion and minimize coupling within a subsystem. Algorithm for Comprehension-driven Clustering (ACDC) is a pattern-driven algorithm developed by Tzerpos and Holt [20]. It first constructs a system skeleton by identifying frequently recurring patterns, and then uses an orphan adoption algorithm to cluster leftover files. Gethers and Poshyvanyk [10] claimed that structure-based clustering algorithms cannot capture conceptual dependency, and proposed a new coupling metric called *Relational Topic based Coupling (RTC)* to capture latent topics and relationships among source code, leveraging LDA, a probabilistic topic model.

These algorithms, however, only process one type of relation. By contrast, our DRSpaces revealed multi-layer, multi-type modular structures, formed by the flexible combination of primary and secondary relations. While their purpose is to help developers understand the structure more easily, our goals are to use DRSpaces to inform the root causes of defects.

8. CONCLUSIONS

In this paper, we have introduced *design rule spaces*, a new form of architecture representation that uniformly captures both architecture and evolution relations using design structure matrices. We proposed that software architectures should be viewed and analyzed as multi-layered overlapping DRSpaces, because each DRSpace, formed using different types of primary and secondary relations, exhibits meaningful and useful modular structures. Each of these structures promotes and supports a different kind of analysis.

As the first attempt to bridge the gap between architecture and defect prediction, we studied the relationships between DRSpaces and bug spaces in three large-scale open source projects. The results showed that error-prone files usually lead error-prone DRSpaces in which most of the files are also error-prone, and that a few error-prone DRSpaces can capture a large portion of the project's error-prone files. Most interestingly, by viewing different DRSpaces of the same architecture, formed and complemented by different types of relations, we were able to identify a large number of structural and evolutionary problems that may contribute to the root cause—the structural cause—of bugginess. This analysis can aid the architect in determining when and how these error-prone files should be fixed.

We envision that DRSpaces have the potential to change how software architecture is viewed, modeled, and analyzed today, and to bridge the gap between architecture and defect prediction by not only locating error-prone files, but also providing suggestions for corrective maintenance.

9. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation of the US under grants CCF-0916891, CCF-1065189, CCF-1116980 and DUE-0837665.

[5]`http://www.headwaysoftware.com/products/structure101/`
[6]`http://www.ndepend.com/`

10. REFERENCES

[1] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The missing links: Bugs and bug-fix commits. In *Proc. 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Nov. 2010.

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

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

[4] Y. Cai, H. Wong, 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*, pages 133–142, June 2013.

[5] 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*, 35(6):864–878, July 2009.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[7] M. D'Ambros, M. Lanza, and R. Robbes. On the relationship between change coupling and software defects. In *Proc. 16th Working Conference on Reverse Engineering*, pages 135–144, Oct. 2009.

[8] B. Fluri, H. C. Gall, and M. Pinzger. Fine-grained analysis of change couplings. In *Proc. 5th IEEE International Workshop on Source Code Analysis and Manipulation*, pages 66–74, Sept. 2005.

[9] 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*, pages 190–197, Nov. 1998.

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

[11] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[12] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proc. 31rd International Conference on Software Engineering*, pages 78–88, May 2009.

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

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

[15] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. pages 452–461, 2006.

[16] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[17] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proc. 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 167–176, Oct. 2005.

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

[19] R. W. Selby and V. R. Basili. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering*, 17(2):141–152, Feb. 1991.

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

[21] S. Wong and Y. Cai. Generalizing evolutionary coupling with stochastic dependencies. In *Proc. 26rd IEEE/ACM International Conference on Automated Software Engineering*, pages 293–302, Nov. 2011.

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

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

[24] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proc. 30th International Conference on Software Engineering*, pages 531–540, May 2008.