Titan: A Toolset That Connects Software Architecture with Quality Analysis

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

Rick Kazman
University of Hawaii
Honolulu, HI, USA
kazman@hawaii.edu

ABSTRACT

In this tool demo, we will illustrate our tool—Titan—that supports a new architecture model: design rule spaces (DR-Spaces). We will show how Titan can capture both architecture and evolutionary structure and help to bridge the gap between architecture and defect prediction. We will demo how to use our toolset to capture hundreds of buggy files into just a few architecturally related groups, and to reveal architecture issues that contribute to the error-proneness and change-proneness of these groups. Our tool has been used to analyze dozens of large-scale industrial projects, and has demonstrated its ability to provide valuable direction on which parts of the architecture are problematic, and on why, when, and how to refactor. The video demo of Titan can be found at `https://art.cs.drexel.edu/~lx52/titan.mp4`

Categories and Subject Descriptors

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

Keywords

Software Architecture, Software maintenance, Software Quality

1. INTRODUCTION

In our recent work [12], we proposed a new architecture representation called the *Design Rule Space* (DRSpace) that bridges the gap between architecture and defect prediction. Based on design rule theory [1], a DRspace models software as design rules and modules that are decoupled by these design rules. We showed that a software system can be viewed as multiple overlapping DRSpaces, each reflecting a meaningful but different perspective of the software. We also showed that although a software project may have hundreds of buggy files, they are usually architecturally connected and can be captured by just a few DRSpaces.

In this demo, we will show how our toolset, called Titan, can be used to automatically extract the DRSpaces that capture most error-prone and change-prone files in a project. We will also show how to flexibly visualize software architecture from multiple perspectives using TitanGUI.

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2661677

2. BACKGROUND AND RELATED WORK

In this section, we introduce the background of Titan: Baldwin and Clark's [1] design rule theory and design structure matrix modeling, as well as Wong et al.'s [10,11] design rule hierarchy and modularity violation algorithms. We will also briefly compare Titan with related work.

Baldwin and Clark proposed that a modular structure is formed by inserting *design rules*—high-level architecture decisions—that decouple the rest of the system into independent modules. Design rules are typically manifested as interfaces or abstract classes. For example, the factory interface in an abstract factory pattern is a design rule that decouples clients from concrete factories. Most design patterns feature one or more interfaces that are instances of design rules.

Based on the concept of design rule, Wong et al. proposed a clustering algorithm called the *Design Rule Hierarchy* (DRH) [11] that automatically detects design rules and modules, which can be reflected in a hierarchical structure of a *Design Structure Matrix* (DSM) [1,11]. In a DRH-clustered DSM, design rules are arranged in the top layers followed by decoupled, mutually-independent, modules.

The essence of the design rule concept is that the modules decoupled by design rules should evolve independently as long as the design rules remain stable. If two modules change together when they are designed to be independent from each other, a *modularity violation* [10] has occurred. Wong et al [10] showed that modularity violations were often a signal of poor design that needed refactoring. For example, our industrial case study [7] showed that a modularity violation could be the result of shared secrets between files that should be better encapsulated.

Our recent work [12] proposed the concept of *Design Rule Space* (DRSpace) as a new software architecture model. A DRSpace is a set of files that are either structurally related or frequently changed together, as evidenced in a project's revision history. We proposed that a software architecture could be viewed as multiple overlapping DRSpaces. We also showed that most error-prone files were connected by just a few DRSpaces with architecture issues that were postulated to be the root causes of bugginess.

Our work bridges software architecture and bug prediction. In the former area, various clustering algorithms, such as ACDC [8] and LDA [2] were proposed to increase the comprehensibility of the architecture. But these algorithms don't reveal how modules affect software quality. Numerous approaches were proposed to predict bug location, using predictors extracted from revision history (history measures) [3], static code base (complexity measures) [4], or a combination of both [5]. However, the architecture connections among buggy files were never explored. Our tool also distinguishes itself from existing DSM tools, such as Lattix [6], by featuring the DRH algorithm, multiple DRSpace views, and automatic error-prone DRSpace extraction.

3. TITAN TOOLSET OVERVIEW

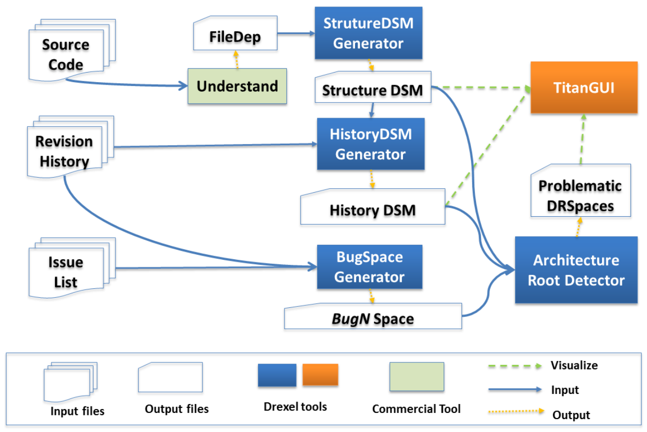


Figure 1: Titan Tool Chain Framework

Figure 1 depicts an overview of our 4 data processing components and 1 visualization component: TitanGUI.

StructureDSM Generator. This component takes the file dependency report generated by a reverse engineering tool, such as *Understand* [1] as input. The output is a structure DSM, in the form of a *.dsm* file, that represents the structural dependencies between files in a project.

HistoryDSM Generator. The input for this component includes a structure *.dsm* file and the revision history of a project, such as a SVN log. The user can specify a start and end date that designate the time span to consider in the generation. Evolutionary coupling is exported to a history dsm, also in the form of a *.dsm* file, that records the co-change frequencies between files in a project.

BugSpace Generator. This component uses revision history, e.g. a SVN log, a bug issue list, and a specified time period as input, and outputs a list of files that were changed multiple times to fix bugs in the time period, ranked by their bug change frequency, and recorded in a .csv file. We call the ranked buggy file list a *bug space*.

Architecture Root Detector. The inputs to this component include a structure DSM, a history DSM, a bug space, and an input parameter P representing the percentage of buggy files to be covered. The user can specify a severity threshold of the bug space, that is, the number of times a file is revised to fix bugs. The larger the number, the more error-prone the files are. If the severity threshold is specified to be N, then we call it a *BugN* space. This component computes the minimal number of DRSpaces needed to capture P% of the given *BugN* space. We call these DRSpaces the *architecture roots*, which are also recorded in .dsm files.

[1]http://www.scitools.com/

TitanGUI. TitanGUI is a component that takes .dsm files generated by the StructureDSM Generator, History DSM Generator, or the Architecture Root Detector as input. Using TitanGUI the user can manipulate, split, import, and export any parts of a DRSpace, save a specific clustering into a .clsx file, or export a DSM view into a spreadsheet.

In the next sections, we illustrate how to extract architecture roots, visualize software architecture as multiple DRSpaces, and inspect architecture issues using Titan.

4. EXTRACTING ARCHITECTURE ROOTS

The four data processing components of Titan share a graphical user interface called TitanDPC (Figure 2). Each component is operated from its own tab. This figure shows the tab for the Architecture Root Detector, where the user can input a structure DSM file, a history DSM file, a bug space file, a bug frequency threshold, and a cover percentage. The system will then generate a set of .dsm files in the specified output directory, each representing a root DRSpace with multiple buggy files. Their aggregation covers the given percentage of files within a *BugN* space.

Using this toolset, we have extracted architecture roots from dozens of open source and industrial projects. Our results consistently showed that a software project's hundreds of buggy files can be captured in just a few DRSpaces: architecture roots. We have reported [12] that for Hadoop, JBoss and Eclipse, more than half of the files in *Bug2*, *Bug5* and *Bug10* spaces were captured by just 5 DRSpaces. Furthermore, using TitanGUI, we can observe multiple architectural issues within each root DRSpace, such as cyclic dependencies, inappropriate inheritance and modularity violations. We believe those architecture issues contributed to the error-proneness of the files in the DRSpaces.

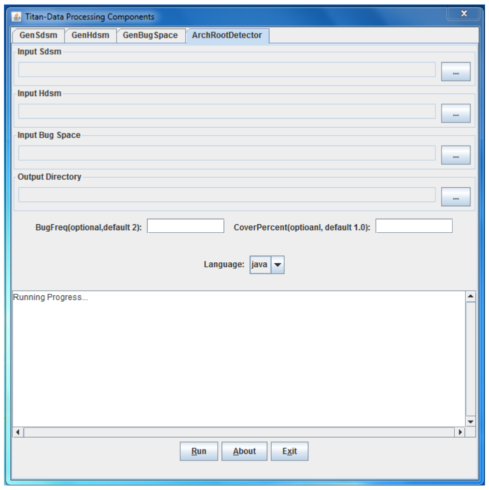
More interestingly, in our most recent study of 10 open source projects and 1 industrial project, we found that, during the evolution of a software project, the number of buggy roots needed to cover the majority of a bug space remains constant over time despite the drastic increase in the size of bug space. In other words: buggy roots grow over time. This observation implies that these roots are the cores of bugginess that connect more and more buggy files. Thus developers need to fix the architectural problems in the buggy roots to avoid the growth of bugginess.

5. EXPLORING DRSPACES

In this section, we introduce how to use TitanGUI to explore software architecture as multiple DRSpaces and to examine architecture roots. Figure 3 is a snapshot of TitanGUI, showing 4 sections: a top toolbar, a control panel on the left, a tree view, and a DSM view.

The toolbar on the top contains action menus for *.dsm* file manipulation (open, save, etc), clustering a DSM based on package structure or ArchDRH structure, splitting a DRSpace based on selected classes, or extracting a subsystem based on a module selected from the tree view panel.

The control panel on the left allows the user to customize the DSM view. When a structure DSM is loaded, the dependency types associated with it will be automatically displayed as a group of check boxes. In the snapshot, we can see that the Apache Camel DSM has 12 types of structural depen-



Input Sdsm

Input Hdsm

Input Bug Space

Output Directory

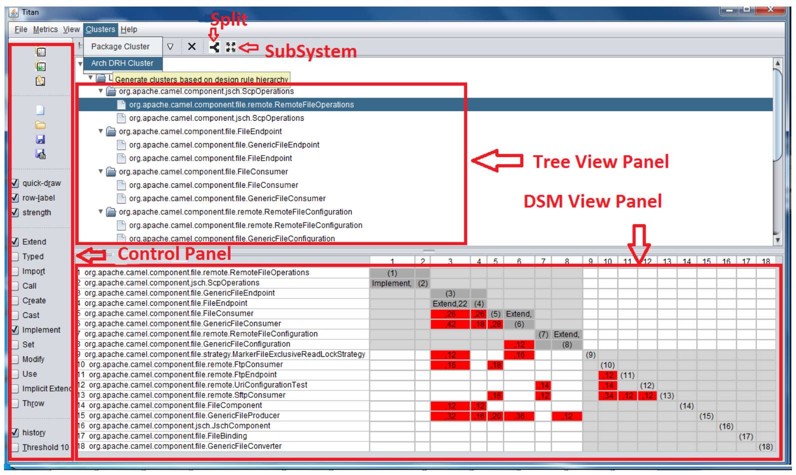
BugFreq(optional,default 2):

CoverPercent(optioanl, default 1.0):

Language: java

Running Progress...

Figure 2: TitanDPC (Data Processing Components)



Generate clusters based on design rule hierarchy

- org.apache.camel.component.jsch.ScpOperations
  - org.apache.camel.component.file.remote.RemoteFileOperations
  - org.apache.camel.component.jsch.ScpOperations
- org.apache.camel.component.file.FileEndpoint
  - org.apache.camel.component.file.GenericFileEndpoint
  - org.apache.camel.component.file.FileEndpoint
- org.apache.camel.component.file.FileConsumer
  - org.apache.camel.component.file.FileConsumer
  - org.apache.camel.component.file.GenericFileConsumer
- org.apache.camel.component.file.remote.RemoteFileConfiguration
  - org.apache.camel.component.file.remote.RemoteFileConfiguration
  - org.apache.camel.component.file.GenericFileConfiguration

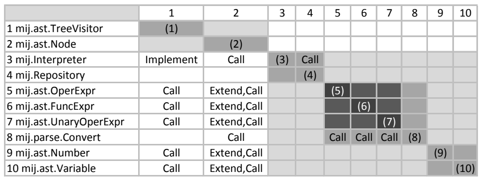
[✓] quick-draw
[✓] row-label
[✓] strength

[✓] Extend
[ ] Typed
[ ] Import
[ ] Call
[ ] Create
[ ] Cast
[✓] Implement
[ ] Set
[ ] Modify
[ ] Use
[ ] Implicit Extend
[ ] Throw

[✓] history
[ ] Threshold 10

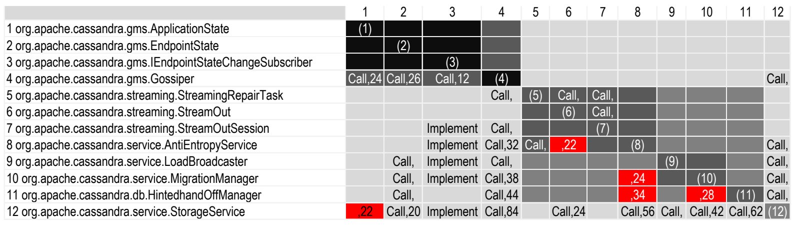
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1 org.apache.camel.component.file.remote.RemoteFileOperations	(1)																	
2 org.apache.camel.component.jsch.ScpOperations	Implement,	(2)																
3 org.apache.camel.component.file.GenericFileEndpoint			(3)															
4 org.apache.camel.component.file.FileEndpoint			Extend,22	(4)														
5 org.apache.camel.component.file.FileConsumer			,26	,26	(5)	Extend,												
6 org.apache.camel.component.file.GenericFileConsumer			,42	,18	,28	(6)												
7 org.apache.camel.component.file.remote.RemoteFileConfiguration							(7)	Extend,										
8 org.apache.camel.component.file.GenericFileConfiguration						,12		(8)										
9 org.apache.camel.component.file.strategy.MarkerFileExclusiveReadLockStrategy			,12			,16			(9)									
10 org.apache.camel.component.file.remote.FtpConsumer			,16		,18					(10)								
11 org.apache.camel.component.file.remote.FtpEndpoint										,12	(11)							
12 org.apache.camel.component.file.remote.UriConfigurationTest							,14			,14		(12)						
13 org.apache.camel.component.file.remote.SftpConsumer					,16		,12			,34	,12	,12	(13)					
14 org.apache.camel.component.file.FileComponent			,12	,12										(14)				
15 org.apache.camel.component.file.GenericFileProducer			,32	,16	,20	,36		,12							(15)			
16 org.apache.camel.component.jsch.JschComponent																(16)		
17 org.apache.camel.component.file.FileBinding																	(17)	
18 org.apache.camel.component.file.GenericFileConverter																		(18)

Figure 3: TitanGUI



	1	2	3	4	5	6	7	8	9	10
1 mij.ast.TreeVisitor	(1)									
2 mij.ast.Node		(2)								
3 mij.Interpreter	Implement	Call	(3)	Call						
4 mij.Repository				(4)						
5 mij.ast.OperExpr	Call	Extend,Call			(5)					
6 mij.ast.FuncExpr	Call	Extend,Call				(6)				
7 mij.ast.UnaryOperExpr	Call	Extend,Call					(7)			
8 mij.parse.Convert		Call			Call	Call	Call	(8)		
9 mij.ast.Number	Call	Extend,Call							(9)	
10 mij.ast.Variable	Call	Extend,Call								(10)

Figure 4: Design Pattern View (Visitor Pattern)



	1	2	3	4	5	6	7	8	9	10	11	12
1 org.apache.cassandra.gms.ApplicationState	(1)											
2 org.apache.cassandra.gms.EndpointState		(2)										
3 org.apache.cassandra.gms.IEndpointStateChangeSubscriber			(3)									
4 org.apache.cassandra.gms.Gossiper	Call,24	Call,26	Call,12	(4)								Call,
5 org.apache.cassandra.streaming.StreamingRepairTask				Call,	(5)	Call,	Call,					
6 org.apache.cassandra.streaming.StreamOut						(6)	Call,					
7 org.apache.cassandra.streaming.StreamOutSession			Implement	Call,			(7)					
8 org.apache.cassandra.service.AntiEntropyService			Implement	Call,32	Call,	,22		(8)				Call,
9 org.apache.cassandra.service.LoadBroadcaster		Call,	Implement	Call,					(9)			Call,
10 org.apache.cassandra.service.MigrationManager		Call,	Implement	Call,38				,24		(10)		Call,
11 org.apache.cassandra.db.HintedhandOffManager		Call,		Call,44				,34		,28	(11)	Call,
12 org.apache.cassandra.service.StorageService	,22	Call,20	Implement	Call,84		Call,24		Call,56	Call,	Call,42	Call,62	(12)

Figure 5: Architecture Root of Cassandra

dencies. Only the checked dependency types are displayed in the DSM view. It also contains a "history" check box. The user can also check the "threshold" box to input a change frequency threshold. Any history coupling below the specified threshold will not be shown in DSM view.

When a DSM is opened, the tree view renders classes randomly. After a clustering file is loaded or a clustering method is selected, the tree view is redrawn to reflect the new structure. Using the tree view, the user can collapse/expand, group/ungroup classes, pick classes to split and select a module to extract a sub system. The DSM view can be updated by clicking the redraw button. If the user selects the "package cluster" menu, the tree view will reflect the project's namespace structure. When the "ArchDRH cluster" menu is selected, the tree view will reflect a DRH structure.

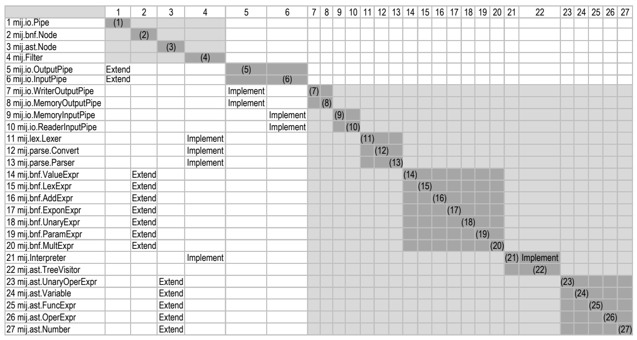
The DSM view presents a square matrix where the columns and rows are labeled with the same set of files in the same order. In the DSM view, each set of classes are colored using a dark background. A nested set has a darker background than the outside group. The diagonal line is labeled with the index of the class. Each cell displays selected dependency types between the file on the row and the file on the column.

5.1 Software Architecture as Multiple, Overlapping DRSpaces

A DRSpace is displayed whenever one or more types of dependencies are selected and the "ArchDRH clustering" menu item is clicked. We use a simple Java calculator program (*mij*) as a running example to show how to explore the different DRSpaces within the same system. The *mij* system was implemented using 36 Java classes, including those implementing Visitor, Pipe-and-Filer, and Interpreter patterns. We now show that the architecture of *mij* consists of multiple DRSpaces, each having its own meaningful modular structure. At the minimum, its architecture consists of a *polymorphism* DRSpace and three design pattern DRSpaces.

A polymorphism DRSpace can be generated by checking *implement* and *extend* types only, and clicking *ArchDRH Cluster*. Figure 6 shows the resulting DSM view of TitanGUI. In this DRSpace, the base classes of `Pipe`, `Filter`, `bnf.Node`, and `ast.Node` are located in the top layer of the design rule hierarchy. The child classes of `io.Pipe`, `OutputPipe` and `InputPipe`, form the second layer design rules which further decouple the rest of the files into 6 mutually independent, meaningful modules. For example, `WriterOutput` and `MemoryOutputPipe` form a output pipe module; the 7 *bnf* classes form a *bnf* node module, and the 4 *ast* classes form an *ast* node module.

We use the *mij* visitor pattern as an example of how to display a design pattern DRSpace using TitanGUI. We first run ArchDRH clustering on the entire system. After that, we select `ast.Node` and `ast.TreeVisitor` on the *Tree View Panel* and then click the *Split* button on the top toolbar. As a result, a new instance of Titan GUI will be launched containing only the files participating in the visitor pattern. Figure 4 is the DSM panel snapshot of the this newly generated visitor pattern DRSpace. The design rules of the pattern, the key interfaces *TreeVisitor* and *ast.Node*, are in the first layer, which decouples the rest of the pattern to 3



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

Figure 6: Polymorphism View

mutually independent modules.

5.2 Architecture Root Exploration

After the architecture roots are identified by the Architecture Root Detector, TitanGUI can visualize their architecture issues. To do this, the following steps are followed: 1) the history and structure .dsm files of the architecture root are loaded using the file menu; 2) the preferred structure dependency types are selected on the left control panel; 3) ArchDRH clustering is run, based on the selected dependencies; 3) the "history" box and an appropriate history threshold are selected on the left control panel.

Figure 5 presents the DRSpace of an architecture root extracted from the Apache Cassandra project. Its DRSpace with structure dependencies reveals five cyclic calls in it, three of which are between `StorageService` and its sibling classes. This is a sign of poor architecture design. After checking the "history" checkbox, a number is displayed after the structural dependency in each cell, indicating the frequency of co-changes between the file on the row and the file on the column. In this DRSpace, evolutionary couplings less than 10 are filtered out. We can see that these architecture issues are incurring high maintenance costs in the form of frequent co-changes: *StorageService* changes with *Gossiper*, *AntiEntropyService*, *MigrationManager* and *HintedHandOff-Manager* (which are participants in 4 call cycles) 84, 56, 42 and 62 times respectively.

The red cell in the DSM view indicates *modularity violations* [9], that is, modules that are structurally independent but change frequently together. Figure 5 shows that this architecture root suffers from modularity violations. We hypothesize that the developers won't be able to reduce the maintenance cost on those files unless they treat them as a group and fix the structural problems.

6. CONCLUSION

In this paper, we presented, Titan, a novel tool that bridges the gap between software architecture and quality. We show that Titan can simultaneously display structure and evolutionary information, highlight modularity violations, and aid in exploring software architecture from multiple perspectives, such as the design space formed by each design pattern. We also show that Titan can cluster the files connected by problematic architecture structures into a few architecture root DRSpaces, and provide insights about how to refactor these structures.

We have applied the Titan toolset in dozens of open source and industrial projects. In the future, we plan to further explore how to automatically extract architecture issues, design patterns, and anti-patterns. The academic version of the Titan toolset can be downloaded at `https://art.cs.drexel.edu:4000/`. Please contact Yuanfang Cai at yfcai@cs.drexel.edu or Lu Xiao at lx52@drexel.edu to request a pin number needed for downloading the toolset.

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.

7. REFERENCES

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

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

[3] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proc. 27th ICSE*, pages 284–292, May 2005.

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

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

[6] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proc. 20th OOPSLA*, pages 167–176, Oct. 2005.

[7] R. Schwanke, L. Xiao, and Y. Cai. Measuring architecture quality by structure plus history analysis. In *Proc. 35rd ICSE*, pages 891–900, May 2013.

[8] V. Tzerpos and R. C. Holt. ACDC: An algorithm for comprehension-driven clustering. In *Proc. 7th WCRE*, pages 258–267, Nov. 2000.

[9] S. Wong and Y. Cai. Improving the efficiency of dependency analysis in logical models. In *Proc. 24th ASE*, pages 173–184, Nov. 2009.

[10] S. Wong, Y. Cai, M. Kim, and M. Dalton. Detecting software modularity violations. In *Proc. 33rd ICSE*, pages 411–420, May 2011.

[11] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi. Design rule hierarchies and parallelism in software development tasks. In *Proc. 24th ASE*, pages 197–208, Nov. 2009.

[12] L. Xiao, Y. Cai, and R. Kazman. Design rule spaces: A new form of architecture insight. In *Proc. 36rd ICSE*, pages 967–977.