Bridging the Gap between Software Architecture and Maintenance Quality

A Dissertation

Submitted to the Faculty

of

Drexel University

by

Lu Xiao

in partial fulfillment of the

requirements for the degree

of

Doctoral of Philosophy in Computer Science

2016

Acknowledgement

On my Ph.D. journey, many wonderful people have helped and influenced me. I have been extremely fortunate to have them in my life and I am deeply grateful to them.

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Yuanfang Cai. She is a passionate researcher, a hard worker, and a role model. Without her encouragement, patience, and selfless support, I would not have been able to achieve my goals. Throughout my years as a Ph.D. student in the Computer Science Department at Drexel University, Dr. Cai has generously given me advice in research and in life, mentoring me to become an independent researcher as well as an academic writer and presenter. Her guidance will continue to influence me beyond my Ph.D. education. The greatest advice I received from her is, "Do what you love and love what you do." I hope to impart this wisdom to my future students.

I am also grateful to have had the honor of working closely with an outstanding scholar and mentor, Dr. Rick Kazman, who is also one of my committee members. I am truly grateful for his wisdom, kindness, and support. As a prestigious researcher, he has given me detailed guidance in my research. He has also patiently explained to me, for whom English is a second language, the meaning of idioms like "a needle in a haystack".

Likewise, thanks are due to my other committee members, Dr. Spiros Mancoridis, Dr. Jeff Popyack, and Dr. Colin S. Gordon. They have encouraged me to think more critically and outside the box. I appreciate their support and thoughtful discussion of my research.

I also owe my gratitude to Dr. Sunny Wong, my advisor's first Ph.D. student. This dissertation is enlightened by and built upon his work.

I would like to sincerely thank my Ph.D. buddies who have accompanied me on this journey. Thank you Ran Mo and Qiong Feng, for being excellent team-mates. Thank you Alex Duff, Bander Alsulami, Brandon Packard, and everyone in the department for your camaraderie and friendship. Thank you my dear friends, Linchuan Meng, Xiaoyu Chu, Lin Jiang and Handong Yang for bringing fun to my life in Philadelphia.

Thank you to my parents Duncai Xiao and Taozhi Liu. Thank you for your love, trust, and sacrifices in allowing me to pursue my career as a researcher in a faraway country. Thanks to my dear sister, Jing Xiao, for your love, support, and encouragement.

I also offer my deepest gratitude to my best friend and "long-suffering" husband, Zhan Zhang. His companionship through every second and minute, and through the ups and downs of this journey, has been invaluable to me. His abiding love and support have seen me through difficult times.

To all these wonderful people, and to the many others for whom space does not allow, I express my deepest appreciation to you all. Because of you, I could accomplish this challenging, yet rewarding journey.

Table of Contents

List of Tables

List of Figures

Abstract

Software architecture is generally recognized as the most critical determinant in achieving the functional and quality attribute requirements of a software system. Poor architecture can be the root cause of quality problems such as bug-proneness and related maintenance difficulties. Software practitioners need to identify architectural flaws and make informed decisions so that they can correct such flaws and fundamentally improve software quality. However, in the past there was no systematic way to model, analyze, and monitor the architecture of a software system with respect to addressing maintenance quality concerns. Consequently, there was a serious gap between software architecture and maintenance quality.

This dissertation offers a methodology to bridge the gap between software architecture and maintenance quality problems. Our proposed methodology consists of three parts: (1) a new architecture model, called the DRSpace model, which simultaneously captures the modular structure and maintenance "penalties" of a software system; (2) an Architecture Root detection algorithm that automatically identifies the most problematic design spaces, aggregating bug-prone files in a software system; and (3) a formal definition of Architectural Debt and an approach that automatically identifies such debts, and quantifies the "costs" and "interest rates" of such debts. Our studies have shown that this methodology has great potential in helping software practitioners identify and understand the architectural root causes of bug-proneness and related high maintenance costs. Ultimately this supports informed refactoring decisions to fundamentally improve software maintenance quality.

Part I

Introduction

Software architecture is critical throughout the entire life cycle of a software project. Bug-proneness and high maintenance costs, two major quality concerns, can be the results of poor architecture. To fundamentally reduce software bug-proneness and improve the maintainability, software practitioners should diagnose architectural flaws that permit the existence of the maintenance quality problems, and make informed refactoring decisions to correct such flaws. For example, if a set of files form a dependency cycle, whenever one of the files in the cycle is revised to fix bugs, it is highly likely that the change will propagate to other files in the cycle. This makes the bugs involving these files hard to eradicate and thus the related maintenance costs will keep increasing over time. Hence, to eradicate bugs and prevent the maintenance costs from keep growing, the developers should identify and cut the cycle.

However, to the best of our knowledge, in the past there was no systematic way to model, analyze, and monitor software architecture with respect to addressing maintenance quality concerns. Consequently, the relationship between software architecture and maintenance quality in terms of bug-proneness and high maintenance costs has not been adequately investigated. Many pertinent questions have not been fully answered. For example, what are the common architectural flaws that contribute to bug-proneness and high maintenance costs? How much effort, time and money have these flaws cost in a software project and how much more will they cost in the future? Whether, when, and where should the developer team invest in refactoring to fundamentally reduce the bug rates and increase the maintainability of a software system in the long run? A major challenge in answering these questions is the lack of a methodology that automatically and effectively analyzes software architecture with respect to addressing maintenance quality problems. Existing work (such as Schwanke et al. [2013]; Maranzano et al. [2005]; Kazman et al. [1994, 1999]) that analyzes software architecture to address quality concerns is largely labor-intensive, experience-based, and anecdotal. In a word, there has been a serious gap between software architecture and maintenance quality in terms of bug-proneness and high maintenance costs.

Thus, the goal of this dissertation is bridging this gap. To achieve this goal, we contribute a methodology, consisting of three parts:

- First, a novel architecture model, called the Design Rule Space (DRSpace) model, that captures the modular structure of software architecture and the maintenance "penalties" simultaneously.
- Second, an Architectural Root (ArchRoot) detection algorithm that automatically identifies the most problematic aspects (DRSpaces) of a system's architecture, which aggregates the bug-prone files.
- Third, an formal definition of Architectural Debt (ArchDebt), and an approach that identifies such "debts" and quantifies their "costs" and "interest rates" to support the refactoring decision-making.

DRSpace Modeling As the first step to bridging the gap, we first propose a new architecture insight, the DRSpace modeling, to simultaneously capture the modular structure of a software system's architecture and the maintenance "penalties" on the source files in the form of change-/bug-proneness.

Based upon the design rule theory of Baldwin and Clark [2000], we propose to represent a software system's architecture as multiple, overlapping Design Rule Spaces (DRSpaces). Each DRSpace is composed of certain leading files—the design rules of the DRSpace—and modules that depend on and decoupled by the leading files. Each DRSpace represents a cohesive aspect of the architecture. In addition, the evolutionary couplings among files are treated as a special form of architectural connections. We use the number of times two files are changed in the same commits in the revision history to reflect the stength of their evolutionary coupling. By simultaneously expressing the modular structure and the evolutionary couplings among files of a software system, the DRSpace modeling helps to reveal architectural flaws that 1) violate common design principles and 2) have incurred maintenance “penalties” in the form of high co-changes.

The DRSpace modeling provides a perspective for inspecting software architecture as multiple overlapping spaces, instead of as an expansive and complex view. In the study of 15 projects of various characteristics, we found that if the leading file of a DRSpace is bug-prone, a large number of the files in the DRSpace will also be bug-prone. Therefore, high-impact and bug-prone design rules should be given higher priority than average files in bug-fixing activities. We also observed various architectural flaws among bug-prone files, such as unstable key interfaces and modularity violations, among the bug-prone files. The developers should be aware of such flaws because they make bugs hard to eradicate. Our DRSpace modeling increases such awareness by providing a perspective for revealing these flaws.

ArchRoot Detection Given the new architecture insight, to further bridge the gap, we propose an algorithm to *automatically* identify a list of DRSpaces that concentrate the bug-prone files in a software system. We call these DRSpaces the Architectural Roots (ArchRoots) of bug-proneness. An architecture analyst should focus on the top few ArchRoots with the highest concentration of bug-prone files to facilitate the identification of architectural flaws.

The analysis of ArchRoots identified in 15 projects advanced our understanding of the impacts of software architecture on maintenance quality. We found that the top few (usually five) ArchRoots aggregate a significant percentage (up to 91%) of the top 30% most bug-prone files in these projects. This indicates that the bug-prone files seldom exist alone. Instead, they are likely to be architecturally connected in only a few groups (ArchRoots). In each project, we observed long-lasting ArchRoots with enduring impacts on the bug-proneness: their leading files kept aggregating a large number of bug-prone files over time. In addition, each root contains multiple, recurring architectural flaws, including unstable interfaces, modularity violations, cyclic dependencies, and unhealthy inheritance. We believe that these flaws can be the root causes of maintenance difficulties, because they can propagate changes among files and consequently keep incurring high maintenance costs over time. Software practitioners should identify and fix these flaws in order to fundamentally reduce the bug rates and increase the maintainability of a software system.

ArchDebt Quantification As long as the flawed architectural connections are not fixed, maintenance costs on files will keep increasing, just as penalties would keep accumulating until the debts were paid off. Hence, we formally define a special form of Technical Debt[1], an Architectural Debt (ArchDebt), as a group of files that keep incurring high maintenance costs due to their flawed architectural connections. The developer team can pay off such "debts" by refactoring, which is to fundamentally fix the high-maintenance architectural flaws. But this will likely delay the planned project progress. Or they can choose not to invest in refactoring immediately, but they are subject to the risk of future higher maintenance "penalties". Given this dilemma, software practitioners need to make informed decisions in terms of whether, when, and where to refactor. Hence, we propose an approach to automatically identify such "debts" and quantify their "costs" and "interest rates" to support informed refactoring decision-making.

We propose an approach to automatically identify which and how files are involved in ArchDebts. This approach identifies ArchDebts by matching four typical architectural flaw patterns. These patterns cover all possible combinations of structural

[1]Technical Debt is a metaphor used to refer to the long-term consequences of shortcuts taken in software development to achieve immediate goals. This concept was first proposed by Cunningham [1992].

dependencies and evolutionary couplings among files. To better model the evolutionary couplings among files, we develop a novel History Coupling Probability (HCP) matrix. The HCP matrix captures the probability of bug propagation from one file to another. We use the probabilities to replace the co-change numbers between files in the original DRSpace modeling. Each ArchDebt, matching one of the four patterns, is a potential refactoring opportunity. To further help software practitioners make informed refactoring decisions, we quantify the "cost" and the "interest rate" on each ArchDebt. Since the actual cost (in terms of time and money) of an ArchDebt can not be measured directly, we use the revised lines of code to fix the bugs involving the files in an ArchDebt to approximate its cost. We also monitor the evolution of each ArchDebt and model the growing trend of the maintenance cost over time to calculate the "interest rate". We use four types of regression models to describe four typical types of interest rates: the linear, logarithmic, exponential, and polynomial regression models represent stable, decreasing, increasing, and fluctuating interest rates, respectively.

In seven software projects with four to eight years of revision history, we identified true debts that generate and grow significant (up to the 85% of the total) maintenance penalties. Interestingly, the most high-impact and expensive type of ArchDebts involve groups of files without any direct structural dependencies but frequently changing together in revision history. This indicates the lack of sufficient design to encapsulate change-/bug-prone concepts shared among files in the projects. In addition, we monitored how an initially trivial architectural flaw evolved over time into a high-impact and expensive "debt" (shown by the example in section 10.4.2). We believe that our approach is able to identify architectural flaws in their early stages as refactoring opportunities. Ultimately, it helps software practitioners make informed refactoring decisions, based on the "cost" and "interest rate" of each "debt", rather than based on one's intuition or experience.

In summary, the contribution of this dissertation in bridging the gap between software architecture and maintenance quality is a systematic methodology to model, analyze, and monitor software architecture. To evaluate the potential of this methodology, we applied it on a variety of software projects, including dozens of open source projects and several commercial projects from Siemens, ABB, SoftServ, and Huawei. Given the different characteristics of these projects, our methodology has consistently shown great potential in identifying the architectural root causes of maintenance difficulties and in supporting informed refactoring decision-making to fundamentally improve maintenance quality. We strongly believe that our methodology will change the way software architecture is analyzed, monitored, and maintained for addressing maintenance quality concerns in practice.

The take-away messages of our studies include the following. First, when developers are trying to fix bugs, they should treat bug-prone files as connected groups, instead of isolated individuals, because the majority of bug-prone files are architecturally connected. Second, the flawed architectural connections among bug-prone files could be the root causes of bug-proneness and ever-accumulating maintenance "penalties". Developers should be aware of such flaws and their consequences when they are revising their codes, particularly if they decide not to immediately fix these flaws through refactoring. Finally, the flawed architectural connections among files, which are like debts, can generate and grow significant maintenance "penalties" in a software project. Software practitioners should identify and monitor such "debts" as early as possible. Ultimately they should make informed refactoring decisions to pay off the "debts" when necessary.

The following of this dissertation is organized as follows. Part II presents the state of art in related fields. Part III introduces the background theories, concepts, and techniques of this dissertation. Part IV introduces and evaluates the three parts of our methodology: the DRSpace modeling, the ArchRoot detection algorithm, and the ArchDebt quantification approach. Part V summaries this dissertation and briefly discusses future directions.

Part II

Related Work

This dissertation is mainly related to the research in the fields of software architecture, bug prediction, and technical debt. This chapter briefly goes through each field and discusses how this dissertation distinguishes from and supplements to existing work.

In chapter 1, we introduce the research in software architecture, including architecture description languages, architecture reverse-engineering techniques, and architecture related analysis.

In chapter 2, we introduce a variety of techniques in bug prediction. Related work lies in three major categories: 1) bug prediction based on the source code complexity metrics, 2) bug prediction based on project revision history information, and 3) bug prediction based on the combination of the prior two types of data.

In chapter 3, we will introduce the state of art in Technical Debt (TD) research, including the origin of the concept, the most up-to-date definition and categorization of TD, as well as existing identification approaches.

1. Software Architecture

Software architecture is the high level structure of a software system, reflected in its components and their connections. A software system's architecture is probably the most critical determinant in attaining required functions as well as non-functional attributes throughout different phases of a project (Garlan et al. [2010]). Hence, the research in software architecture has drawn significant attention in the software engineering community. There are a large amount of literature devoting to the description, documentation, recovering, and analysis of software architecture. In this chapter, we will briefly go through these literature and discuss how this dissertation supplements to the state of art.

1.1 Software Architecture Description Language (ADL)

To communicate software architecture for addressing different concerns, stakeholders require a language that can be understood and documented. The work in ADLs provides tools for parsing, displaying, analyzing, or simulating architectural descriptions written in their associated languages. An ADL could be any form of formal representation to describe software architecture. It could be in either graphic or syntax representation. It describes software entities, such as processes, threads, data, and their interactions. Garlan [2003] summarized from related literature that ADLs provide notations and concrete syntax for modeling software architecture as components, connectors, and events.

There has been various types of ADLs. Each has different focus. For example, Terry et al. [1994] contributed a suit of supporting tools to help specify, design, validate, package, and deploy distributed intelligent control and management (DICAM) applications, in the domain of vehicle management systems. Newton and Browne [1992] developed a graphic parallel programming system that models parallel architectures. Palsberg et al. [1995] implemented a Demeter system that can be used to design and automatically generate adaptive programs specified by a so-called propagation pattern. Jahanian and Mok [1994] proposed a specification language for real-time systems called Modechart. Stephen H. Edwards and Weide [1994] proposed RESOLVE to describe a conceptual module as a RESEOLVE unit that specified an abstract component by defining its context and its interface structure and behavior for designing reusable components. Shaw et al. [1995] sketched and implemented a model called UniCon for defining architectures as different types of components and different ways these components can interact. Allen and Garlan [1994] proposed a model called Wright to distinguish between "implementation" and "interaction" relationships between modules. To help architecture understanding, Aldrich et al. [2002] contributed an extension to Java, called *ArchJava*, that aimed at keeping architecture and implementation consistent.

Clements [1996] reviewed and compared the pros and cons of different types of ADLs. They claimed that a glaring commonality among different ADLs was the lack of in-depth experience and real-world application. In addition, they claimed that none of the ADLS could capture the design rationale and/or evolution history of architecture.

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

1.2 Software Architecture Reverse Engineering

This dissertation is also related to software architecture recovery. In reality, the ground truth of a software system's architecture is usually missing due to the absence of an up-to-date and accurate document. Over the past decades there has been a considerable amount of research devoted to this problem. Researchers have been trying to provide more precise ways to recover a software system's architecture, while the development is going on. According to Kruchten [1995] and Bass et al. [2012], a software system has different architecture views from different perspectives. Hence, various reverse-engineering techniques have been developed to produce the high-level architecture with different foci.

Schwanke and Hanson [1994] proposed a tool to model the modularization of a software system as nearest-neighbor clustering and classification. The tool learns from human architects by performing back propagation on a neural network. Tzerpos and Holt [1997] presented the Orphan Adoption problem in architecture maintenance. The Orphan Adoption problem refers to the accommodation of newly introduced resource, such as variables or source files, to the existing architecture view. They proposed an algorithm to recover an up-to-date and accurate architecture view to reflect the newly added "orphans". Mancoridis et al. [1999] developed a clustering tool call Bunch to produce high-level system decomposition descriptions from the low-level structures present in the source code. Their approach was featured by the integration of designer knowledge about the system into an otherwise fully automatic clustering process. Tzerpos and Holt [2000] proposed a clustering algorithm helps to improve program comprehension based on subsystem patterns. Similar to the DRSpace modeling in this dissertation, Sangal et al. [2005] also utilized the DSM and hierarchical clustering based on the design rule theory to manage complex software architecture. Garcia et al. [2013] attempted to recover the ground-truth software architecture of four open source projects, using a combinations of techniques and resources, such as available documentation, structural dependencies between code-level entities, domain knowledge of the systems, and certification from the authority. They were able to recover the ground-truth architecture of software projects. They claimed that a single system can have multiple architecture views depending on the perspective the recovery of architecture was approached. Bavota et al. [2014] proposed an approach to recover the modular structure by analyzing underlying latent topics in source code and structural dependencies.

The above automated or semi-automated techniques recover the high-level architecture of a software system from the low-level implementation based on similarity, data sharing, domain knowledge, and call graphs. The goal of these techniques is to enhance the comprehensibility of a software system's architecture to help software developers to understand and finish their development tasks in hand.

In this dissertation, the DRSpace modeling utilizes reverse-engineering technique to recover the modular structure of software architecture. We first generate a call graph describing the static references among files. Then we use the Design Rule Hierarchy clustering algorithm proposed by Huynh et al. [2008a,b] (we will discuss this algorithm in greater detail in Part III) to represent software architecture as design rules and modules. The most distinctive aspects of our DRSpace modeling, from the architecture models created by existing reverse-engineering techniques, are that (1) we model software architecture as multiple overlapping spaces, instead of a general view generated by the above techniques; (2) the maintenance "penalties" in terms of change-/bug-proneness on software entities are directly captured in the DRSpace modeling, which are not considered in any of the above techniques.

In addition, the ultimate goal of this dissertation is not just to correctly recovering a software system' architecture views. Architecture recovering serves to facilitating maintenance quality related analysis. Our approach supports automatic and systematic analysis of software architectural flaws that are responsible for quality concerns. None of the recovering technique directly supports such analysis.

1.3 Software Architecture Analysis

One of the important goals of software architecture reverse-engineering is, of course, to support different analysis. Software architecture has been utilized in analysis for addressing concerns related to various aspects of software development, such as software evolution, development parallelism, and quality.

MacCormack et al. [2006] used Design Structure Matrix (DSM), which is also used in our work, to map dependencies between a software system's design elements and their relationship. They used this representation to compare the design of two large scale projects: Mozilla and Linux. They also tracked the evolution of these projects and identified "re-design" effort undertaken with the intention of improving the "modular" structure. Parnas [1972] discussed the modularization as a mechanism for improving flexibility and efficiency of the development of a software system. According to their discussion, software architecture should be designed as composing of independent modules to support parallel implementation, instead of as composing of sub-routines. Along this line, Wong et al. [2009] presented a software system's architecture as a Design Rule Hierarchy (DRH), which is composed of design rules and independent modules. The DRH reflects and supports the analysis of the parallelism in software development tasks. Robillard [2008] analyzed the topology of software dependencies to guide developers' navigation to find potentially relevant code to a change task. Kouroshfar et al. [2015] studied the role of software architecture in the evolution and quality of software. They found that changes made across different architectural modules are more likely to be error-inducing. But their work doesn't provide in-depth insight in terms of how and why such changes are error-inducing.

As shown above, software architecture has been used to aid analysis for addressing different concerns. However, none of the above work has directly linked software architectural with maintenance quality. No in-depth answers have been given to answer the question, whether and how software architecture is related to maintenance quality problems. Existing studies attempting to relate software architecture and quality concerns have either focused on questionnaires (Maranzano et al. [2005]) or scenarios (Kazman et al. [1994, 1999, 2001]). These methods are labor-intensive and their success depends heavily on the skill of the analysts. To the best of our knowledge, the methodology introduced in this dissertation is the first in *automatically* and *directly* linking software architecture and maintenance quality concerns.

2. Bug Prediction

Software bugs consume a considerable amount of efforts to discover, test and fix. To improve the efficiency of such activities, ample techniques have been proposed to predict the location of future bugs. Researchers build predicting models to locate bugs in a software system based on three categories of data: the complexity metrics, history bug fixing information, and a combination of the prior two types of data. In this chapter, we will briefly go through existing work in bug prediction field and discuss how our work distinguishes from and supplements to the bug prediction field.

2.1 Prediction based on Complexity Metrics

Different complexity metrics have been proposed to measure the structure of software architecture. The most basic complexity metrics include the number of files and the lines of code (LOC) in a software project. Researchers, such as McCabe [1976], Henry and Kafura [1981], Chidamber and Kemerer [1994], Basili et al. [1996], Bansiya and Davis [2002], and Ran Mo and Feng [2016], proposed a variety of complexity metrics based on different criterions. For example, McCabe [1976] proposed the Cyclomatic complexity to measure the number of linearly independent paths in a program's source code. Chidamber and Kemerer [1994] proposed a suite of metrics based on the object-oriented design principles. Most recently, Ran Mo and Feng [2016] proposed a Decoupling Level to measure how well software architecture is decoupled into independent modules.

Researchers in the bug prediction field have used different metrics to predict software entities/modules that are most likely to be bug-prone. Selby and Basili [1991] used the coupling and strength to identify error-prone system structure. Ohlsson and Alberg [1996], Nagappan et al. [2006], and Ohlsson and Alberg [1996] collected various complexity metrics, such as McCabe's cyclomatic complexity, fan-in and fan-out, to build their prediction models. Nagappan et al. [2006] investigated eighteen complexity metrics as bug predictors, in a case study of five major Microsoft software components. These studies have shown that different complexity metrics work best for different projects. Bansiya and Davis [2002] and Briand et al. [2000] assessed the relationship between various object-oriented design metrics, such as coupling and cohesion, with the quality attributes, such as reusability, flexibility, understandability, etc.. Zimmermann and Nagappan [2008] built their prediction models based on measures derived from the network analysis of the dependency graph. These studies have shown the usefulness of various complexity metrics in predicting bug-proneness. Menzies et al. [2007] argued that how complexity attributes are used to build predictors is much more important than which particular attributes are used.

2.2 Prediction based on Project History

A project's revision repositories, including the version control system and the bug tracking database, are also valuable sources for bug prediction. Researchers extracted a variety of history predictors, such as artifact ownership and number of changes made to file, for bug-proneness.

T.L. Graves and Siy [2000] proposed a weighted time damp model, which computes the fault potential of modules in software system, based on the changes made to each module in revision history. They explored the extent to which measurements based on the change history were more useful in predicting fault rates than complexity metrics. They found, for example, the number of times code had been changes was better indication of bug-proneness than its length. Jones et al. [2002] visualized the test information to assist fault localization. Ostrand et al. [2004] built a negative binomial regression model using information from previous releases, such as file age and file change, to predict the number of faults with each file in a large industrial inventory system. Nagappan and Ball [2005] used relative code churn measures to predict system defect density. Kim et al. [2007] built a model, called FixCache, using cached bug information to predict future bug locations with high accuracy. Moha et al. [2008] introduced an approach to automate the generation of design defect detection algorithm from the existing textual descriptions of defects. Eaddy et al. [2008] found that crosscutting concerns can cause defects. D'Ambros et al. [2009] analyzed the relationship between change coupling, in terms of co-changes among software artifacts, and software defects. Hassan [2009] argued that a complex code change process negative effects on the system. They claimed that complexity of code changes is better predictors of fault compared to other well-known historical predictors, such as prior modifications and prior faults. More recently, Posnett et al. [2013] predict bug-proneness based on both the developer focus and artifact ownership. They found that developers in charge of greater number of files were more likely to introduce defects than developers in charge of fewer number of files. In the meanwhile, files revised and reviewed by more developers were less likely to contain defects than other files.

2.3 Prediction based on Combined Information

Some researchers build their prediction modules based on combined history and complexity metric predictors. For example, Fenton and Ohlsson [2000] tested a ranges of predictors based on revision history and complexity metrics. They found that modules with most of the pre-release faults also contained most of the faults discovered in operation. They also reported that there was no found evidence to support previous claims that complexity metrics are good fault predictors. Ostrand et al. [2005] successfully predicted 80% of the faults using file size and file change information for two large industrial systems. Cataldo et al. [2009] examined the impact of syntactic, logical, and work dependencies (in terms of workflow dependencies and coordination requirements) on the failure proneness of a software system. The results of their study suggested that re-architecting held promise for reducing defects.

In summary, the goal of bug prediction is to efficiently and economically predict bug locations to facilitate testing and bug fixing activities. It has been shown, and as we have discussed above, that history bug information could be a good source for bug prediction, meaning files that are buggy in the past tend to remain buggy in the future. In the one hand, this is good news for developers because history bug fixing information can provide insight in where to find bugs in the future. On the other hand, however, this indicates that bugs are seldom entirely fixed by the developers, otherwise, how could past bug information be used to predict future bugs? Based on the reasoning, a more important question that has not been addressed by bug prediction is why bugs are hard to eradicate in software systems. Or in other words, what are the root causes that contribute to the bug-proneness?

Leszak et al. [2000] attempted to answer this question by manually inspecting the bug tickets in the bug tracking database, and categorizing bugs into different types. In this dissertation, we attempt to approach this problem from the perspective of software architecture. The underlining assumption is that poor software architecture design could make bugs hard to eradicate. We argue that, instead of predicting future bug locations, software practitioners should fix the root causes of bug-proneness to fundamentally reduce bug-fixing effort.

3. Technical Debt

3.1 Technical Debt Origin

Since Cunningham [1992] first coined the term up in 1992, *Technical Debt (TD)* has been used to describe the consequences of shot-cuts taken in software projects to achieve intermediate goals. During the past decade, *TD* has drawn increasing attention in the software engineering community (Brown et al. [2010]; Kruchten et al. [2012]; Shull et al. [2013]; Falessi et al. [2014]; Seaman et al. [2015]).

3.2 Technical Debt Definition and Categorization

Li et al. [2015] conducted a mapping study on different categories of *TD* based on related literature published between 1992 and 2013. They classified ten coarse-grained *TD* types according to the phases of the software development life-cycle, such as requirements, architectural, and code. They found that Code *TD* is the most well studied type, and Architectural *TD* has also received significant attention. They further categorized Architectural *TD* into seven sub-categories, including architectural smells (Mo et al. [2013]), architectural anti-patterns (Griffith and Izurieta [2014] and Peters [2014]), complex architectural behavioral dependencies (Brondum and Zhu [2012]), violations of good architectural practices (Curtis et al. [2012]), architectural compliance issues (Kazman and Carriere [1999]), system-level structural quality issues, and all others. *TD* can compromise both functional and quality requirements, such as performance, security, usability, and modifiability.

Alves et al. [2014] organized 13 types of *TD* and their key indicators, including Architectural *TD*. They described Architectural *TD* as “problems encountered in software architecture”, and referred to issues in software architecture, structure dependencies/analysis, and modularity violations as indicators of Architectural *TD*. Their work focused on building the ontology of *TD* rather than focusing on resolving a specific type of *TD*.

3.3 Technical Debt Identification

Maldonado and Shihab [2015] proposed an approach to identify different types of "self-admitted" *TD* in software projects, by reviewing the comments left by developers. They identified five types of self-admitted *TD*: design, requirement, defect, test, and documentation *TD*. According to their study, the most common types of *TD* are design and requirement. But as the name "self-admitted" suggests, the *TD* identified in their work was limited to ones that the developers are aware of. There are forms of *TD* introduced unwittingly by developers.

Martini and Bosch [2015] conceptualized two patterns of Architectural *TD*: contagious debt and vicious circle. Contagious debt leads to ripple effects in projects. Vicious circle refers to a more severe contagious debt where the ripple effects form a loop. Their work has two limitations. First, it intensively relies on interviewing developers to identify these problems. As stated above, it is possible the developers are not aware of all the *TD* existing in their project. Furthermore, this approach is labor-intensive and relies highly on the expertise of the analyst. Second, this only identifies two anti-patterns, and these overlap with each other.

Given the substantial research literature, it is surprising that definitions of the types of *TD* are still largely informal. In fact, the identification of *TD* relies heavily on interviews or reviewing developers' revision comments, and these are only problems that the developers are aware of. Many questions in *TD* research remain open. For example, how to precisely define the forms of *TD*, how to automatically identify these forms of *TD*, and how to measure *TD*: its costs and interest rates.

Part III

Background Concepts and Techniques

This part introduces concepts and techniques that serve as the foundation of this dissertation. The content is organized as follows.

In chapter 4, we introduce the design rule theory proposed by Baldwin and Clark [2000]. According to this theory, any complex system can be interpreted as design rules—the high level design decision, and modules that are decoupled by the design rules. Our DRSpace modeling embraces this theory.

Chapter 5 introduces the Design Structure Matrix (DSM) representation proposed by Baldwin and Clark [2000]. DSM is a compact matrix representation for modeling and visualizing system entities and their relationships. In this dissertation, we use the DSM to represent and visualize the DRSpace modeling.

Chapter 6 introduces the Design Rule Hierarchy (DRH) algorithm proposed by Huynh et al. [2008a,b]. DRH is a DSM-based clustering algorithm that automatically captures the design rules and modules of a system. The algorithm will arrange the design rules and modules in a hierarchical manner, such that the dependencies among entities will form a lower triangle in the DSM representation. We use the DRH algorithm to calculate the hierarchy of design rules and modules in each DRSpace.

Chapter 7 discusses modularity violation proposed by Wong et al. [2011]. Modularity violation refers to the phenomenon that structurally independent modules keep changing together in revision history. In this dissertation, the ArchDebt quantification approach quantifies the maintenance consequences brought by such violations. In addition, the DRSpace modeling can directly visualize modularity violations and their maintenance consequences.

4. Design Rule Theory

The design rule theory, proposed by Baldwin and Clark [2000], allows for both independence of structure and integration of function in large and complex systems. Design rules are important design decisions that decouple the other design parameters in a system into mutually independent modules. Design rules facilitate interdependence within and independence across modules. The design and implementation of a module can be held independent from another module, as long as established design rules are obeyed. Baldwin and Clark [2000] claim that any complex modern system can be interpreted as consisting of design rules and modules.

In this dissertation, we advocate Baldwin and Clark [2000]'s design rule theory. We believe, the architecture of any complex modern software system can also be interpreted as design rules and modules. In modern programming languages following object-oriented philosophy, like Java, the interfaces or abstract classes usually play the role of design rules. For example, in the abstract factory pattern[1] (Gamma et al. [1994]; Meyer [1988]), the abstract factory interface is the design rule, which decouples concrete factories and the clients of the factories into independent modules. As long as the abstract factory interface remains stable, the implementation of a concrete factory should be independent from the implementation of other concrete factories and from that of the clients.

The DRSpace modeling is based on the design rule theory. We propose to represent a software system's architecture as multiple overlapping design rule spaces. Each space is composited of certain design rules, and modules that are decoupled by the design rules. Each space represents a cohesive aspect of the architecture.

[1]Abstract factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

5. Design Structure Matrix (DSM)

Design Structure Matrix (DSM) is a powerful tool for presenting the architecture of a software system. The design decisions and their interdependences in a complex system can be compactly mapped into a square matrix (the DSM). The rows and columns can represent the source files, arranged in the same order. The off-diagonal cells show the dependency from the file on the row to the file on the column. The diagonal cells imply self-dependencies of each file. Since self-dependencies are not imperative for architecture analysis, the diagonal cells are marked by the order numbers of files instead for the sake of readability.

For example, Figure 5.1 is the DSM of a simple Java program , reversed-engineered from the source code. In the DSM, the left-most column shows the list of source files. The top-most row shows the order numbers of these files as arranged on the left-most column. The mark on cell [r6:c7] indicates the file on row 6 (*mij.ast.FuncExpr*) depends on the file on column 7 (*mij.ast.Node*). We use rectangles within the matrix to show groups of files clustered together based on a certain criteria. In this particular case, files are group together according to their directory structure. A DSM can also be clustered in other methods, such as the design rule hierarchy algorithm, which will be introduced in Chapter 6.

In this dissertation, we leverage the DSM representation to model and visualize software architecture. Each DRSpace can be represented and viewed as a separate DSM. In order to facilitate the diagnosing of architectural flaws that violate common design principles and indeed incurred maintenance consequences, we develop two types of DSM. The first is a structure DSM, which models the structural dependencies among files. In the cells of a DSM, we distinguish various types of structural dependencies, such as “inherit” and “depend”. The second type is a history DSM,

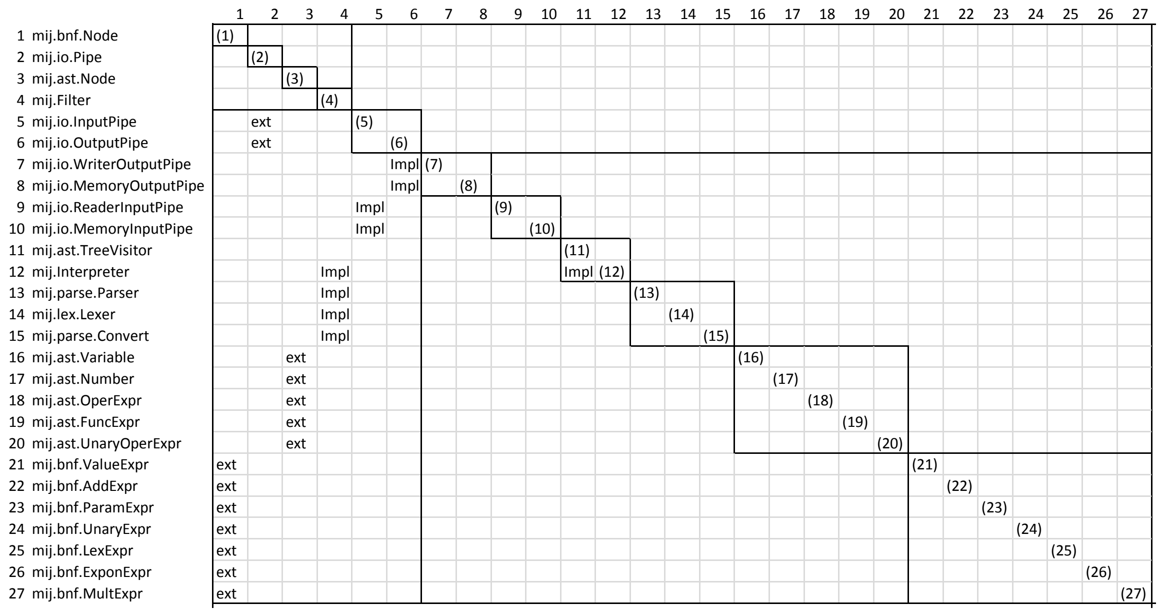
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
1 mij.Console	(1)	x	x	x			x											x				x	x	x	x	x	x	x			x	x	x	x
2 mij.FilterException		(2)																																
3 mij.Filter		x	(3)																															
4 mij.Interpreter		x	x	(4)	x	x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

Figure 5.1: MIJ DSM in Package Cluster

which models how files change together in revision history. Each cell represents the number of times the file on the row and the file on the column change together. We can stack a structural DSM and an evolutionary DSM together to visualize the modular structure and the change-proneness of files simultaneously. We will illustrate this in greater detail in Section 8.2.

6. Design Rule Hierarchy (DRH)

Huynh et al. [2008a,b] proposed a DSM based clustering algorithm, call the *Design Rule Hierarchy (DRH)* algorithm. The algorithm automatically distinguishes the architecture roles of source files as design rules and modules in a DSM. A DSM clustered by the DRH algorithm has 3 key features: (1) the design rules and modules are arranged in hierarchical levels, with design rules on the upper levels, while the modules decoupled by the design rules on lower levels; (2) the modules in lower levels depend on the modules in higher levels, but not vice versa; (3) modules in the same level are mutually independent from each other.



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

Figure 6.1: MIJ Inherit DRSPace

Figure 6.1 shows a DSM (formed by "extend" and "implement" relationships) with 3 levels. The first level (row 1 to row 4) contains 4 files that are the super classes or interfaces recognized as the design rules. The second level (row 5 to row 6) contains 2 files that “extend” *mij.io.Pipe* on the first level. The third level (row 7 to row 27) contains 21 files that extend/implement files in the first 2 levels. Level 3 is decoupled into 6 mutually independent modules, each module follows a specific design rule in the top 2 levels.

In this dissertation, we apply the DRH algorithm on each DRSpace to automatically capture the leading files and modules of each DRSpace.

7. Modularity Violation

Wong et al. [2011] use the term *modularity violation* to refer to the phenomenon in software projects that, a set of files *should*, according to their modular structure, evolve independently, but they *actually* are highly coupled with each other in revision history. Wong et al. [2011] implemented a tool, called Clio, which can detect *modularity violations* in software projects from the source code and revision history. In the experiments on three open source projects, Clio identified large numbers of *modularity violations*, which were verified to bring maintenance consequences, such as errors, modularity decay, or even expensive refactorings.

We conducted a case study (Schwanke et al. [2013]) on an industry agile project. In this study, we identified and verified many cases of *modularity violations* using Clio. We found that *Modularity violations* usually suggest "shared secrets" (undocumented assumptions) among files that require better encapsulation. For example, we found that a set of files in the project shared an assumption of using the same time unit without encapsulating this concept explicitly. Whenever the time unit used in one file changed, the other files had to accommodate accordingly.

The DRSpace modeling, utilizing the DSM representation, can help to reveal and visualize modularity violations in a software system. In a DSM, if a cell contains evolutionary coupling without any structural dependencies, it indicates a modularity violation. Aided by the visualization tool, Titan-GUI (which will be introduced in Section 9.3), software practitioners can investigate modularity violations conveniently. In addition, in the ArchDebt quantification approach, we quantify the maintenance "costs" and model the "interest rate" of each instance of modularity violation. We found that modularity violations are actually the most high-impact and expensive type of flaws.

Part IV

Our Methodology

In this part, we will introduce our methodology in great detail.

Chapter 8 introduces the DRSpace modeling, and shows its usefulness in helping software practitioners to understand the architectural root causes of quality concerns, relating to bug-proneness and high maintenance costs.

Chapter 9 presents the Architectural Root detection algorithm based upon the DRSpace modeling. We applied the ArchRoot detection algorithm on 15 software projects. The analysis of the ArchRoots identified in these projects advanced the understanding of the architectural root causes of bug-proneness.

Chapter 10 formally defines a particular form of Technical Debt—Architectural Debt, based on the observations made using the DRSpace modeling and ArchRoot detection algorithm. In this chapter, we also provide an approach to automatically identify, quantify and model the maintenance consequences of ArchDebts.

8. Design Rule Space (DRSpace) Model

In a case study on a commercial agile project (Schwanke et al. [2013]), we found that software architectural flaws, such as unstable key interfaces and important undocumented assumptions, could cause maintenance difficulties. However, like other similar studies (Maranzano et al. [2005]; Kazman et al. [1994, 1999]), the diagnosing of poor architectural decisions contributing to the maintenance quality problems was largely labor-intensive and experience-based. To the best of our knowledge, there was no systematic way to analyze the architectural root causes of bug-proneness and related high maintenance costs. In a word, there remains a gap between software architecture and maintenance quality.

To bridge this gap, a model that expresses relevant architecture information for diagnosing quality problems is vital. Hence, in this chapter, we propose a new architecture insight, called the Design Rule Space (DRSpace) modeling, to capture the modular structure and the change-/bug-proneness of files simultaneously. The DRSpace modeling is based upon the design rule theory proposed by Baldwin and Clark [2000]. They claim that any complex modern system can be interpreted as design rules—the high level decisions, and modules—the implementations of concrete tasks. As long as the design rules are well established and rigorously obeyed, modules can be implemented and maintained independently.

Based on the design rule theory, we propose to represent the modular structure of software architecture as multiple overlapping Design Rule Spaces (DRSpaces). Each DRSpace, composed of certain key design rules and modules following the design rules, reflects a cohesive aspect of software architecture. For example, a complex system can apply multiple design patterns[1]. Each design pattern can, and should,

[1]Design patterns are well accepted design solutions for common problems(Gamma et al. [1994];

be expressed by a separate DRSpace. The key interfaces of a pattern are the design rules, and the concrete implementations of the design rules are the modules. We will illustrate the DRSpace modeling using a simple Java system as an example in section 8.2.

In the DRSpace modeling, the evolutionary couplings among files are expressed as a special form of architectural connections. The number of times two files are changed together in revision history denotes the strength of their evolutionary coupling. The more frequently two files are changed together, the more coupled they are. We visualize the evolutionary couplings among files and the modular structure of a DRSpace simultaneously. This helps to diagnose architectural flaws that 1) violate common design principles and 2) have actually brought maintenance "penalties" (in the form of high co-changes). For example, the evolutionary coupling between two structurally independent modules indicates a shared implicit assumption among them. High coupling between a key interface and its dependent modules suggests an unstable key interface.

Our supporting tool, Titan-GUI, visualizes each DRSpace in the form of a Design Structure Matrix (DSM). The rows and columns of a DSM represent files in a DRSpace. Each cell can flexibly display various relationships between the file on the row and the file on the column, including different types of structural dependencies and/or the evolutionary coupling. Visualizing the evolutionary coupling and the modular structure simultaneously can help to reveal architectural flaws among the bug-prone files. In section 8.4.2, we will show examples of architectural flaws identified in an open source project.

As the first step to bridging the gap between software architecture and maintenance quality, the DRSpace modeling has shown great potential in facilitating the

Freeman et al. [2004])

understanding of the architectural root causes of bug-proneness. We found that, if a design rule is bug-prone, files depending on it are also likely to be bug-prone. Thus, bug-prone and high-impact key design rules should be granted higher priority in bug fixing activities. The bug-prone files in a DRSpace are architecturally related to each other through the dependencies to the common design rule. In addition, with the help of the DRSpace modeling, we revealed multiple architectural flaws among the bug-prone files in 15 open source projects, such as unstable interface and cyclic dependencies. The developers should be aware of such architectural flaws when they fix bugs. They should even consider fixing these flaws first to prevent error propagation in the bug fixing activities.

The following of this chapter is organized as follows. Section 8.1 defines what is a DRSpace. Section 8.2 illustrates the overlapping DRSpaces of a small Java project. Section 8.3 presents the supporting tools for building and viewing the DRSpace modeling. Section 8.4 shows the usefulness of the DRSpace modeling for understanding the architectural root causes of bug-proneness in 15 open source projects. Section 8.5 discuss the limitations and threats to validity of our DRSpace modeling. Section 8.6 summarizes this chapter.

8.1 DRSpace Definition

Design Rule Space (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. It distinguishes different types of structural connections among files, such as “inherit”, “realize”, “aggregate” and “depend”. “Inherit” and “realize” are basic polymorphism concepts in object-oriented programming languages, like Java. “Extend” (i.e., “inherit”) refers to a child class

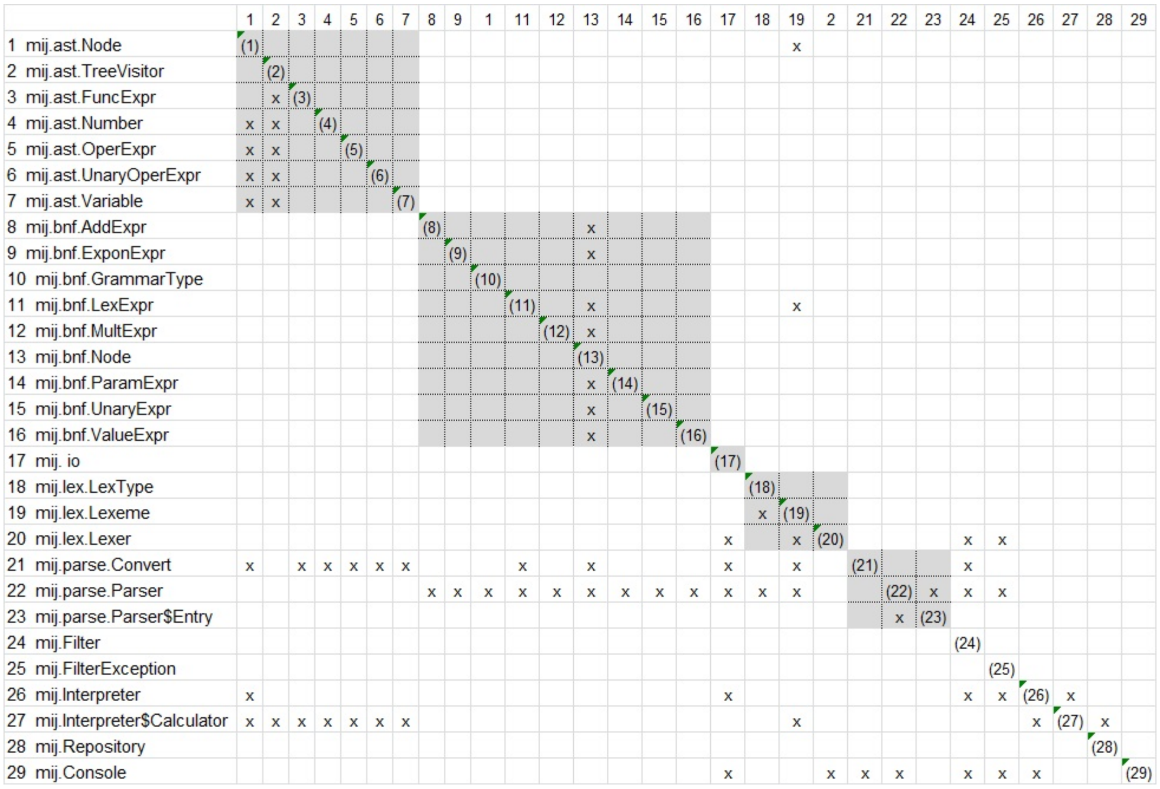
inheriting from a parent class. "Implement" (i.e., "realize") refers to a concrete class realizing an interface. All other general references, such as method calls, between files are uniformly recognized as "depend". It models evolutionary couplings among files as a special form of architecture connection. The evolutionary couplings are extracted from the revision history of a project. If two files change in the same commits, we consider the two files to be evolutionarily coupled. The number of times they change together in the same commits represents the weight of their evolutionary coupling. For example, if two files change together in 10 commits, the weight of their evolutionary coupling is 10.

2. The vertices (classes) of a DRSpace must be clustered into the DRH form (introduced in chapter 6) 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 always contains one or more *leading files*, which are the *design rules* of the space. All other files in the space directly or indirectly depend on the *leading files*. We use the term *leading files* instead of *design rules* because the latter usually refer to architecturally important decisions for the whole system, while the former are only the *design rules* of a particular *DRSpace*. We use a DSM (introduced in Chapter 5) to represent a DRSpace. And we apply the DRH algorithm (introduced in Chapter 6) to automatically capture the *leading files* and modules in a *DRSpace*. Each module is a cluster of structurally coupled

files that directly or indirectly depend on the *leading files.* The DRH algorithm also arranges the *leading files* and *modules* of a DRSpace in a hierarchy in the DSM: the *leading files* are on the top level, while the modules are in the lower levels. If a DRSpace *DRS* has leading *LD*, we also say that *DRS* is led by *LD*.

8.2 DRSpace Model Illustration



	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)

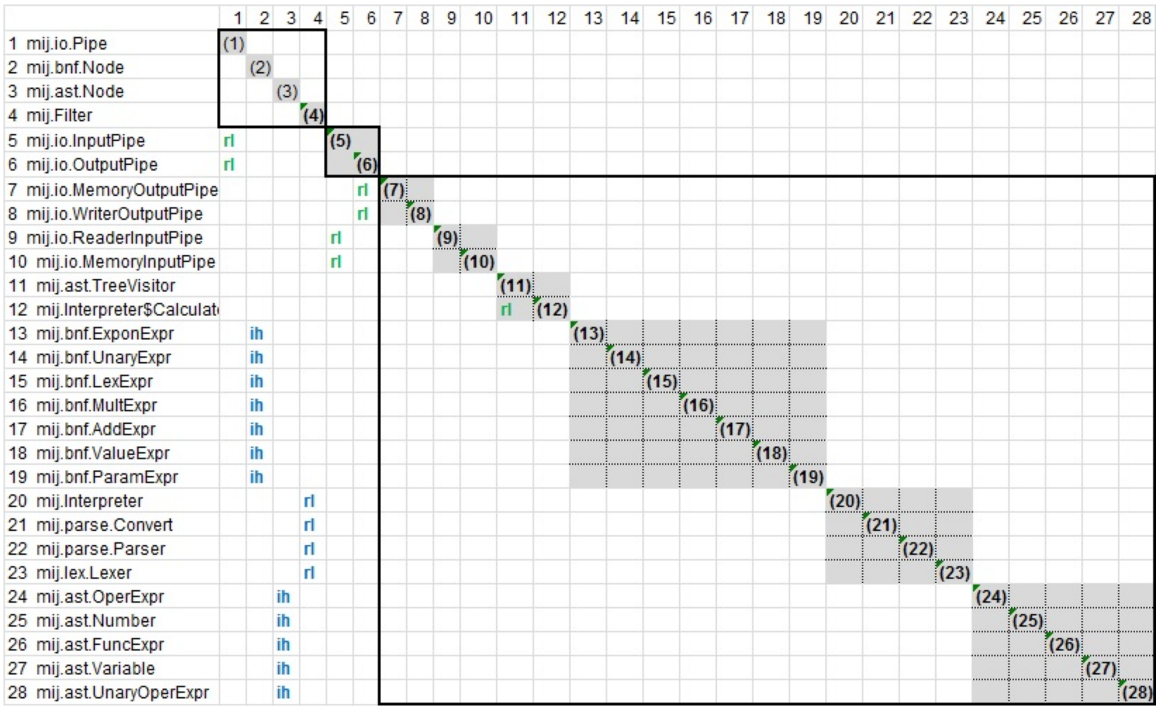
The rows and columns represent files, arranged in the same order. The “x” mark in each cell represents a structural dependency from the file on the row to the file on the column. Each inner rectangle represents a package.

Figure 8.1: MIJ DSM in Package Cluster

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

Figure 8.1 shows the traditional DSM (used in most existing tools, such as Latix [2]) of the MIJ program, reverse-engineered from the source code. This DSM only models a uniformed dependency type, presented by “x”, among files. The first column is the list of Java files in MIJ, arranged first by the directory hierarchy, and then in alphabetical order. Each non-empty cell in the matrix indicates a dependency from the file on the row to the file on the column. For example, there is a “x” in cell[r3:c2], indicating the file on row 3 (*mij.ast.FuncExpr*) depends on the file on column 2 (*mij.ast.TreeVisitor*). It is clustered by the file directory structure. Each inner rectangle in the matrix represents a package in MIJ. For example, files in the *ast* (abstract syntax tree) directory, from file 1 (*mij.ast.Node*) to file 7 (*mij.ast.Variable*), are grouped in a rectangle.

In a way of contrast, the complexity of software architecture cannot be well represented using just a single view like Figure 8.1. Next, we will show the uniqueness of our DRSpace model in representing software architecture as multiple overlapping spaces. Groups of files addressing different concerns should be viewed separately. For instance, each (set of) dependency type(s) can be used as the primary relation(s) to form a separate DRSpace.



	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$Calculat											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)

"rl" stands for realize. "ih" stands for inherit. The inner rectangles are calculated by the DRH algorithm (introduced in chapter 6) based on the displayed relationships to show the modular structure of this Polymorphism space.

Figure 8.2: MIJ Polymorphism DRSpace

8.2.1 Polymorphism DRSpace

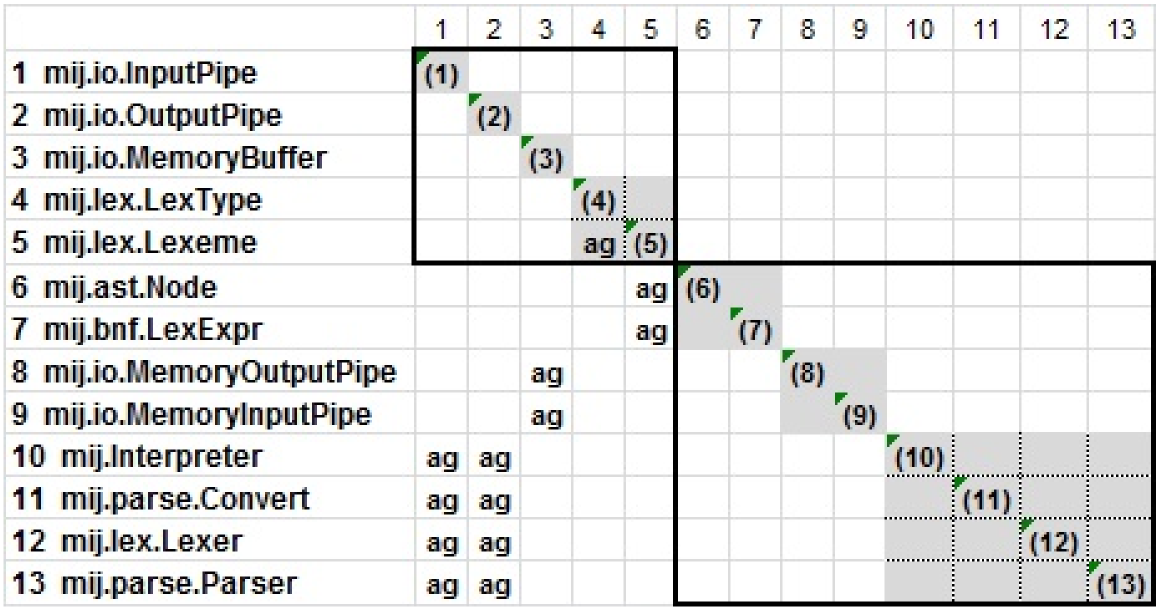
Figure 8.2 shows a *DRSpacce* formed by dependency types "ih" (inherit) and "rl" (realize) as the primary relations. This *DRSpace* depicts the modular structure of the inheritance tree in MIJ. The DRH algorithm (introduced in chapter 6) automatically captures such structure in a hierarchy. The design rules and modules are reflected by the layered inner-rectangles in the DSM in Figure 8.2.

The leading files (which are also the design rules) of this space are clustered into the first layer, containing four files, including *mij.io.Pipe*, *mij.bnf.Node*, *mij.ast.Node*, and *mij.Filter*. All other files in this space are clustered into modules based on their structural dependencies to the leading files. Groups of files extending or im-

[2]http://lattix.com/

plementing different base classes or interfaces are decoupled into mutually independent modules (shown as rectangles in the lower parts of the DSM). For instance, file *mij.ast.OperExpr* (row 24) to file *mij.ast.UnaryOperExpr* (row 28) form the concrete *ast* (abstract syntax tree) module, because they all extend base class *mij.ast.Node*. File *mij.bnf.ExponExpr* (row 13) to file *mij.bnf.ParamExpr* (row 19) form the concrete *bnf* (Bbckus-naur form) module, because they all extend base class *mij.bnf.Node*.

8.2.2 Aggregation DRSpace



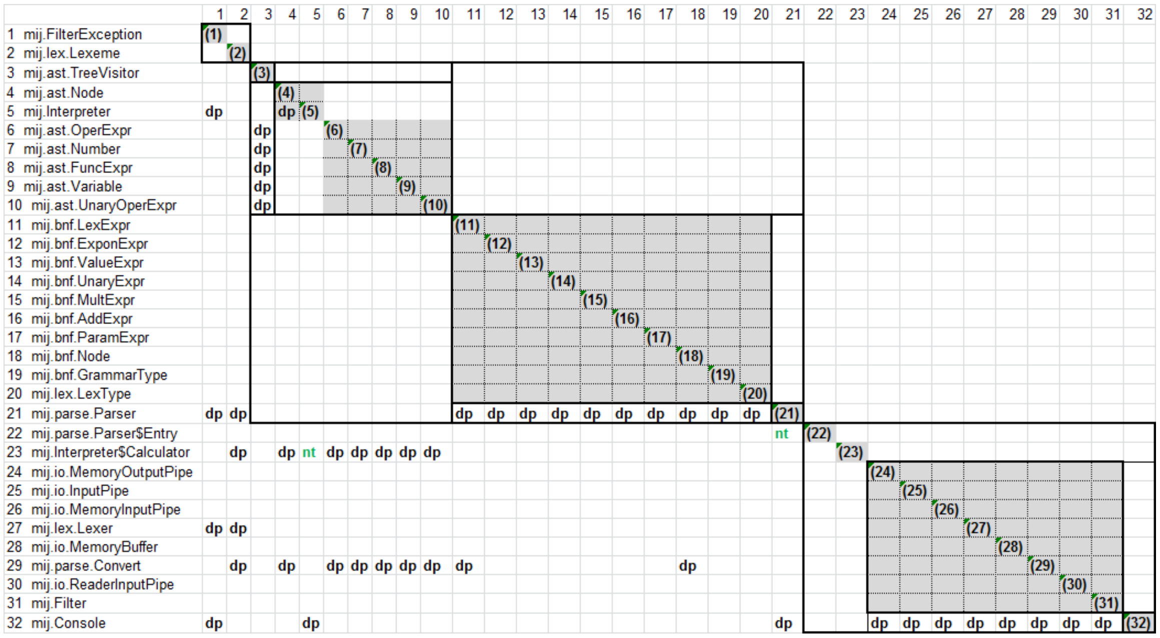
	1	2	3	4	5	6	7	8	9	10	11	12	13
1 mij.io.InputPipe	(1)												
2 mij.io.OutputPipe		(2)											
3 mij.io.MemoryBuffer			(3)										
4 mij.lex.LexType				(4)									
5 mij.lex.Lexeme				ag	(5)								
6 mij.ast.Node					ag	(6)							
7 mij.bnf.LexExpr					ag		(7)						
8 mij.io.MemoryOutputPipe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

"ag" stands for a class aggregating another class as an attribute. The inner rectangles are calculated by the DRH algorithm (introduced in chapter 6) to capture the modular structure formed by "aggregation" relationship.

Figure 8.3: MIJ Aggregation DRSpace

Similarly, Figure 8.3 depicts the DRSpace in which the primary relation is aggregation (shown as "ag" in the DSM). Aggregation relationship refers to a class aggregating another class as its attribute. By applying the DRH algorithm (introduced in chapter 6), there are two layers in this DRSpace. The first layer, row 1 to row 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*.

8.2.3 Dependency DRSpace



	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$Entry																					nt	(22)										
23 mij.Interpreter$Calculator		dp		dp	nt	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)

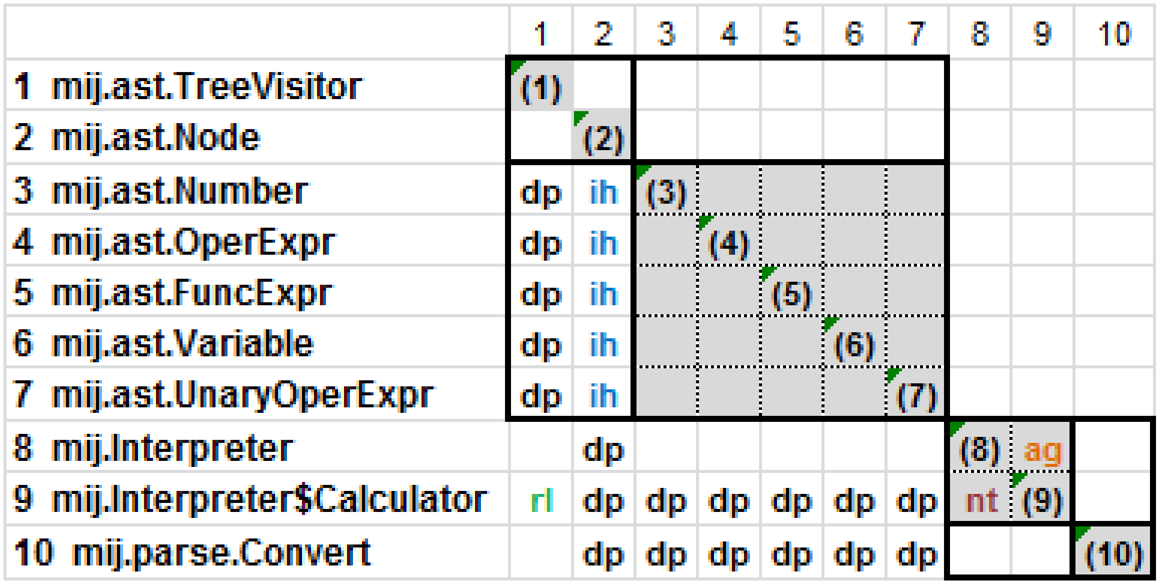
"dp" stands for all general types of references, such as function calls. The inner nested rectangles are calculated by the DRH algorithm (introduced in chapter 6) to capture the modular structure formed by the "dp" relationship.

Figure 8.4: MIJ Depend DRSpace

Figure 8.4 depicts the DRSpace with general structural dependencies, such as function calls, as the primary relations. 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.

8.2.4 Pattern DRSpace



	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	nt	(9)	
10 mij.parse.Convert		dp	dp	dp	dp	dp	dp			(10)

"rl" stands for realize. "ih" stands for inherit. "dp" stands for all other general references. "ag" stands for a class aggregating another class. "nt" means that a class nest another class.

Figure 8.5: MIJ Visitor Pattern DRSpace

Figure 8.5 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*.

8.2.5 Hybrid DRSpace

Figure 8.6 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.

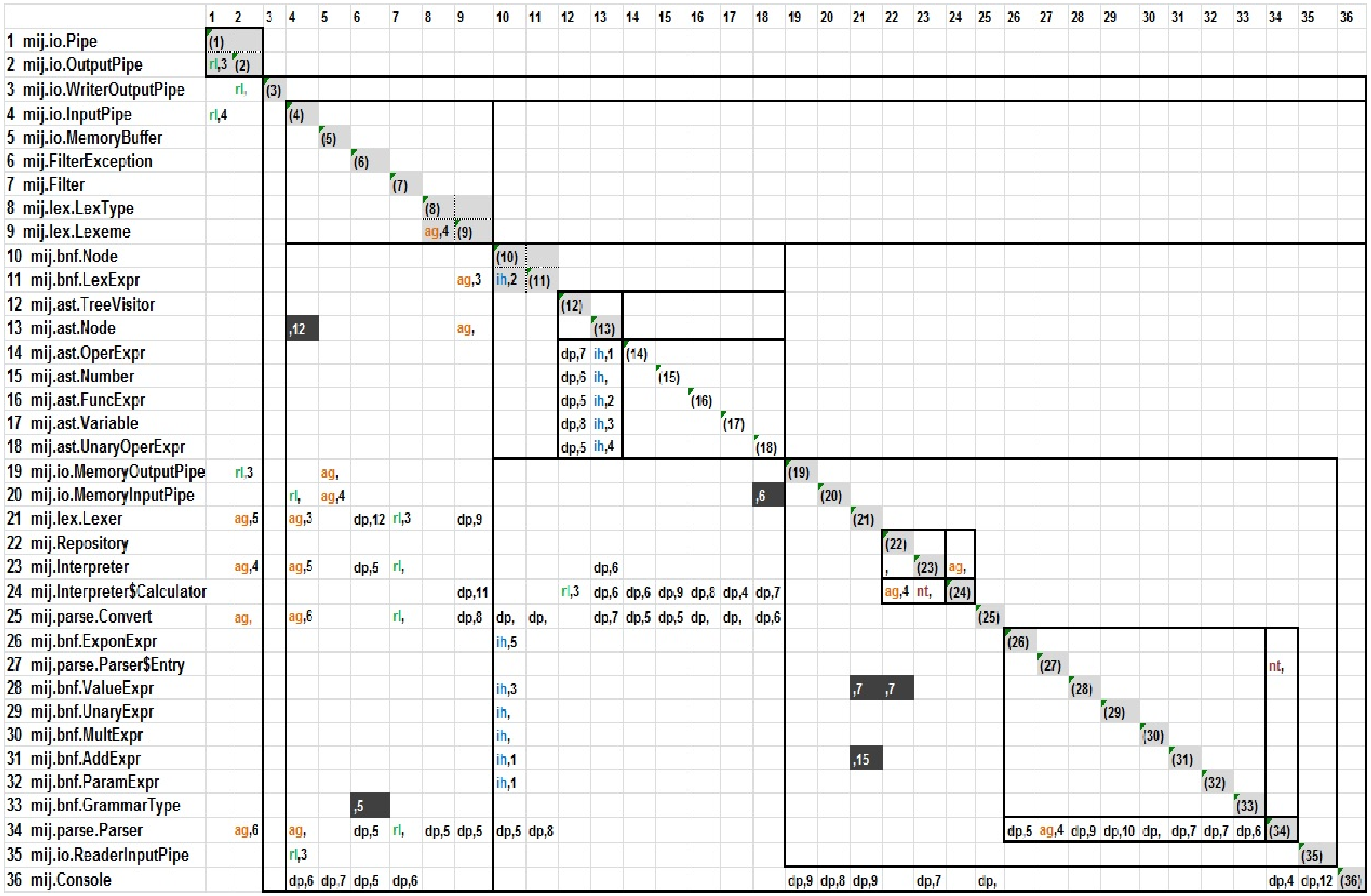
In this DRSpace, we also choose evolutionary coupling as the secondary relation. For example, cell c:(r13, c4) has number 12, meaning that *mij.ast.Node* and *mij.io.InputPipe* changed together 12 times in the revision history. This cell has dark background and white font to indicate that there are no 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 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.

8.3 Tool Support

In this section, we briefly introduce our tool, Titan (Xiao et al. [2014b]), that supports the creation and visualization of DRSpaces. All the figures in this dissertation were created from data exported from Titan.

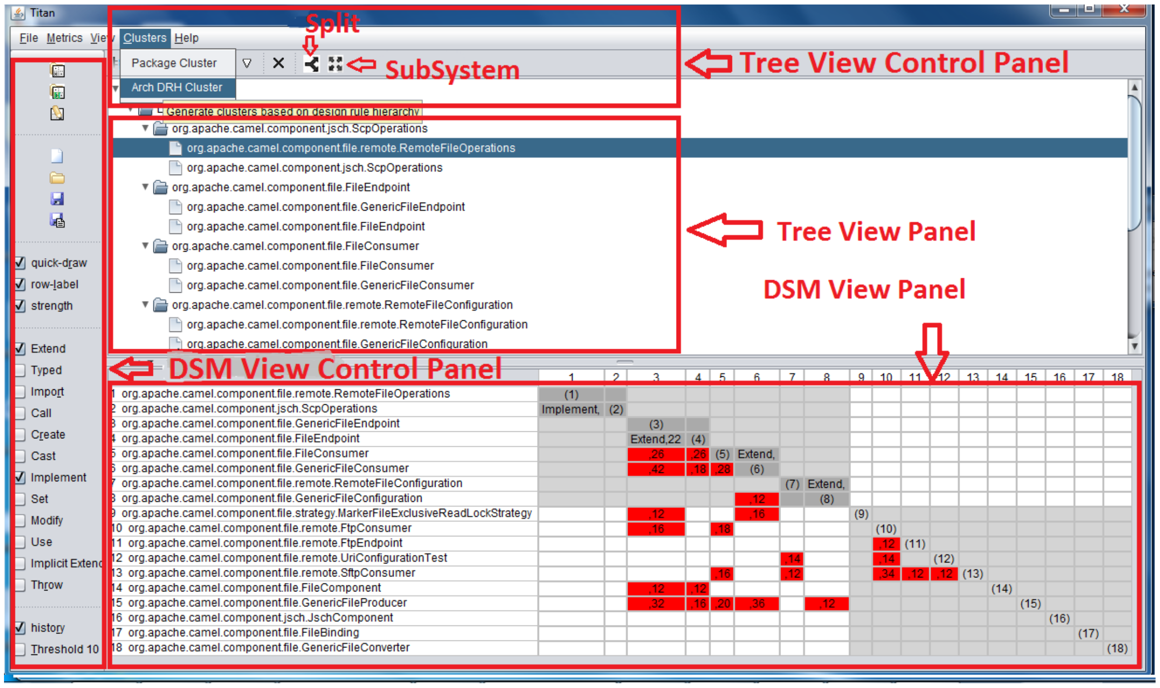
Titan accepts DSM files, with extension .dsm. There are two types of .dsm files: the structure DSM and the history DSM. The structure DSM is calculated from the file



	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$Calculator									dp,11			rl,3	dp,6	dp,6	dp,9	dp,8	dp,4	dp,7				ag,4	nt,	(24)												
25 mij.parse.Convert		ag,		ag,6			rl,		dp,8	dp,	dp,		dp,7	dp,5	dp,5	dp,	dp,	dp,6							(25)											
26 mij.bnf.ExponExpr										ih,5																(26)										
27 mij.parse.Parser$Entry																											(27)							nt,		
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)

“rl” stands for realize. “ih” stands for inherit. “ag” stands for aggregate. “dp” stands for all other types of structural dependencies.

Figure 8.6: MIJ Hybrid DRSpace



- 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

	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 8.7: The DRSpace Viewer - Titan Graphic User Interface

dependency report generated by a reverse engineering tool, such as *Understand* [3]. For a structure DSM, the value in a cell is used to represent different types of relations. So far our tool processes inheritance, realization, dependency, aggregation, and nested.

The history DSM is extracted from the revision history of a project, such as a SVN log. For a history DSM, the number in a cell represents the number of time the two classes changed together (where "changed together" means that both classes were involved in the same commit), which is called *co-change frequency*.

Titan also accepts clustering files, with extension .clsx, as another input. The .clsx files are generated by the DRH clustering algorithm. It describes the nested hierarchy of design rules and modules in the XML format. One .dsm file can be associated with multiple clustering files, each representing a different way the DSM can be clustered.

Figure 8.7 shows a snapshot of Titan's GUI. Similar to commercial tools with

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

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. The leaf nodes represent files (classes) in a DRSpace, while the parent nodes represent clusters of files in the given (or generated) clsx file.

The Tree View Control Using the tree view control panel, 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. DRH Cluster. This is the clustering method we employed to generate DRSpaces in this paper. We have discussed this clustering algorithm in Chapter 6 in Part II (Background). This algorithm automatically captures the design rules and modules in a hierarchy in the DSM representation. The design rules will be ranked on the top of the DSM, while the models depends on the design rules will be ranked on the lower parts of the DSM.

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 representing only 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 splitted.

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 cells show the relations among files, including different types of structure dependencies, and evolutionary couplings. If two classes do not have any structural relation but still changed together, the cell will have a red background. For example, cell c:(r5,c3) shows that although *FileConsumer* and *GenericFileEndpoint* have no structural relation, they changed together 26 times.

The DSM View Control 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 8.7 displays aggregation, nesting, and history relations. The cell c:(r4,c3) has: "extend,22", meaning that *FileEndpoint* extends *GenericFileEndpoint*, and they changed together 22 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 8.7, 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 evolution coupling together with structure relations so that their discrepancies can be visualized.

8.4 Usefulness of DRSpace Modeling

We have shown (Xiao et al. [2014a]), based on the study of three large scale open source projects, that the DRSpace model is useful for understanding the structural relations among bug-prone files, as well as revealing the problematic relations that contribute to the bug-proneness. If a design rule is bug-prone, the majority of files contained in the space led by it are also bug-prone. We call such a DRSpace a bug-prone DRSpace. The bug-prone files in such a space are architecturally connected with each other, directly or indirectly, because they all depend on the same design rule. Therefore, when developers are fixing bugs on these files, they should consider them as a connected group, instead of as isolated individuals. In addition to that, bug-prone design rules, leading large numbers of bug-prone files, should be given higher priority in bug-fixing activities.

In this dissertation, we chose another 15 Apache open source projects to further validate the usefulness of the DRSpace model. Those projects differ in size, application domain, length of history, and other characteristics. They are: Avro [4]

[4] https://avro.apache.org/

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

Summary information for these projects is given in Table 8.1. The first column shows the project names. The second column shows the length of each project's history covered by our study, denoted by a start time-stamp to an end time-stamp, and the number of months within the history (in parentheses). The third column shows the number of releases we selected in each project. Each project has many snapshot-

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

Table 8.1: Summary of Evaluation Projects

Subject	Length of history (#Mon)	#Versions	#Commits	#Committers	#Issues	#Files
Avro	8/2009 to 1/2014 (53)	12	1115	17	734	156 to 426
Camel	7/2008 to 7/2014 (72)	12	14563	106	2790	1838 to 9866
Cassandra	9/2009 to 11/2014 (62)	10	14673	122	4731	311 to 1337
CXF	12/2007 to 5/2014 (77)	13	8937	46	3854	2861 to 5509
Derby	10/2007 to 8/2014 (83)	13	4275	23	2726	2388 to 2776
Hadoop	8/2009 to 8/2014 (60)	9	8253	75	5443	1307 to 5488
HBase	12/2009 to 5/2014 (53)	9	6718	37	6280	560 to 2055
Mahout	10/2008 to 2/2014 (64)	9	3113	22	658	455 to 1262
MINA	10/2005 to 10/2009 (49)	8	1760	19	467	219 to 550
OpenJPA	2/2007 to 4/2013 (74)	11	6098	25	1572	1266 to 4314
PDFBox	8/2009 to 9/2014 (62)	12	2005	16	1857	447 to 791
Pig	3/2008 to 1/2012 (47)	10	1668	19	2579	302 to 1195
Tika	6/2008 to 1/2015 (80)	15	2412	17	714	131 to 550
Wicket	6/2007 to 1/2015 (92)	15	8309	65	3557	1879 to 3081
ZooKeeper	4/2008 to 11/2012 (55)	10	1012	10	1154	151 to 382

s/releases during the history covered by our study. We carefully selected "stable" releases, such that the interval between any two consecutive releases is roughly 4 to 6 months. The column "#Commits" is the number of revisions made to each project. The column "#Committers" is the number of developers who were submitting changes to each project. Both "#Commits" and "#Committers" are extracted from the version control systems of the projects: either from SubVersion [19] or Git [20]. The column "#Issues" is the number of bug reports in each project, which is extracted from the JIRA [21] bug-tracking database of each project. The last column shows the size of each project, measured by the number of files in the first release and the last release.

8.4.1 Bug-prone DRSpaces

We have reinforced our findings in these 15 open source projects as shown in Table 8.2. In order to analyze the structural relations among bug-prone files, we first

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

rank all the files by the frequency they are revised to fix bugs. The more frequently a file is involved in bug fixes, the more bug-prone it is. We designate three bug-prone file sets, which we call bug spaces, according to the bug-fixing frequencies:

1. Bug10%—the top 10% most frequently revised files to fix bugs;
2. Bug30%—the top 30% most frequently revised files to fix bugs;
3. Bug100%—all the files ever revised to fix bugs.

Then we measure how bug-prone a DRSpace is by computing its intersections with the three bug spaces. We use the following two parameters to masure the intersection between a DRSpace, DRS, and a bug space, Bug_X:

1. **Design Space Bugginess (*dsb*)**: the percentage of files in DRS that are also in Bug_X. It is calculated as:

$$dsb = \frac{|DRS \cap Bug_X|}{|DRS|} \tag{8.1}$$

   , where $|DRS|$ is the number of files in the DRS, and $|DRS \cap Bug_X|$ is the number of files in the intersection between DRS and Bug_X.

2. **Bug Space Coverage (*bsc*)**: the percentage of files in Bug_X that are also contained in DRS. It is calculated as:

$$bsc = \frac{|Bug_X \cap DRS|}{|Bug_X|} \tag{8.2}$$

These two values, together measure how bug-prone a DRSpace is. The higher both values are, the more bug-prone the DRSpace is.

Table 8.2 lists one DRSpace from each project with a bug-prone leading file. The leading files of these spaces are the top 1% to the top 22% most frequently revised files to fix bugs. These DRSpaces contain from 34 (ZooKeeper) to 856 (Camel) files.

From Table 8.2, we can summarize that, **when the leading file is bug-prone, a significant portion of files in its space are also bug-prone.** In these DRSpaces, from 28% (Hadoop) to 76% (Cassandra) of files were revised to fix bugs. In addition, from 17% (Avro and Hadoop) to 65% (PDFBox) of files are in $Bug_{30\%}$ (the top 30% most frequently revised to fix bugs). Furthermore, from 8% (Avro) to 38% (Mahout) of files are from $Bug_{10\%}$.

We can also summarize that, **when the leading file is bug-prone, its space aggregates a non-trivial, if not significant, portion of the bug-prone files in a project.** From 5% (CXF and Hadoop) to 65% (Avro) of files in the $Bug_{100\%}$ spaces are aggregated in a DRSpace with a bug-prone leading file. Even though the *bsc* is much lower in some projects (for example, only 5% for $Bug_{100\%}$ in CXF and Hadoop), each DRSpace still aggregates a non-trivial number of bug-prone files (for example, 32 in Hadoop and 80 in CXF), because the $Bug_{100\%}$ spaces can contain hundreds and even thousands of files. Similarly, from 9% (Hadoop) to 72% (Avro) of files in $Bug_{30\%}$ and from 13% (CXF) to 80% (Avro) of files in $Bug_{10\%}$ are aggregated in just a single DRSpace with a bug-prone leading file.

For a particular note, there exist super large spaces, led by high-impact and bug-prone design rules. These spaces aggregate a large percentage of bug-prone files. For example, the DRSpace led by *ObjectHelper* (top 1% most bug-prone) from Camel, containing a total of 856 files, aggregates 64% of files in $Bug_{10\%}$. And 54% of files in the space were revised to fix bugs, meaning, every other file in the space was involved with bugs. We observed a similar case for the DRSpace led by *SQLState* (top 1% most bug-prone) from Derby, which contains 658 files, and with both high *bsc* and high *dsb*.

Table 8.2: DRSpaces with a Bug-prone Leading File

Project (Release#)	Leading File (# Files in DRS)	Leading File Bug Info		$Bug_{100\%}$		$Bug_{30\%}$		$Bug_{10\%}$	
		#Bug Fixes	Bug Rank	bsc	dsb	bsc	dsb	bsc	dsb
Avro (1.7.4)	Schema (197)	18	5%	65%	46%	72%	17%	80%	8%
Camel (2.12.4)	ObjectHelper (856)	31	1%	30%	54%	49%	29%	64%	14%
Cassandra (1.2.5)	CFMetaData (165)	86	2%	19%	76%	36%	48%	43%	29%
CXF (3.0.0)	JAXRSUtils (106)	41	1%	5%	75%	10%	46%	13%	32%
Derby (10.11.1.1)	SQLState (658)	43	1%	48%	67%	62%	30%	75%	15%
Hadoop (2.5.0)	SecurityUtil (114)	16	6%	5%	28%	9%	17%	18%	11%
HBase (0.98.2)	ProtobufUtil (171)	60	1%	9%	65%	15%	47%	25%	23%
Mahout (0.9)	AbstractJob (85)	8	4%	8%	68%	13%	47%	21%	38%
MINA (1.1.7)	ByteBuffer (84)	10	22%	33%	42%	40%	23%	50%	11%
OpenJPA (2.2.2)	DBDictionary (329)	77	1%	7%	50%	14%	32%	29%	18%
PDFBox (1.8.7)	PDResources (54)	25	4%	9%	74%	13%	65%	16%	30%
Pig (0.9.0)	PigContext (193)	7	15%	14%	49%	28%	26%	33%	13%
Tika (1.7)	XHTMLContentHandler (108)	15	5%	28%	55%	44%	33%	55%	16%
Wicket (6.19.0)	RequestCycle (204)	26	1%	9%	60%	17%	35%	25%	24%
ZooKeeper (3.4.5)	Leader (34)	23	8%	18%	71%	31%	47%	43%	26%

bsc is the percentage of all the bug-prone files that are also contained in a DRSpace.

dsb is the percentage of files in a DRSpace that are also bug-prone.

Based on these observations, we believe that a bug-prone and high-impact design rule should be grant higher priority in bug-fixing activities, because a large number of files aggregated in the DRSpace led by it are also likely to be bug-prone. These bug-prone files are structurally related to each other, directly or indirectly, given their structural dependencies to the same design rule. Therefore, when developers are trying to fix bugs, these files should be treated as a connected group, instead of as isolated individuals.

8.4.2 Problematic Relations

Our Titan-GUI provides insights into what are the problematic relations among files that contribute to a bug-prone DRSpace. Figure 8.8 is part of the DRSpace led by *ObjectHelper* from Camel, generated using Titan with all the types of structural dependencies as the primary relations and the evolutionary coupling as the secondary relation. We can see from column "#b (# of bug fixes)" and "br (bug-prone ranking)" that files displayed rank from the top 1% to the top 18% most bug-prone in

Camel. All the files in the space have direct structural dependencies on the leading file *ObjectHelper*, except *OnCompletionDefinition* which indirectly depends on it.

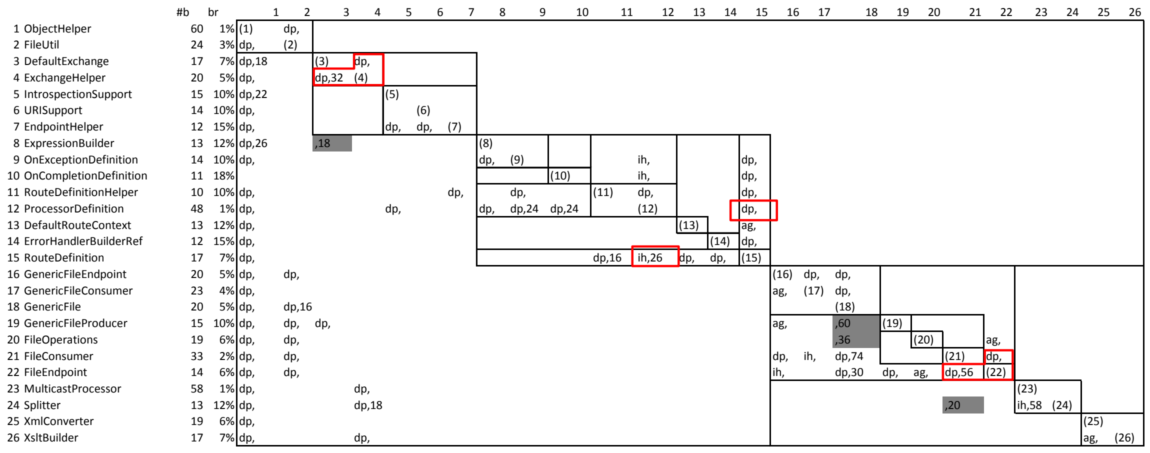
With the help of Titan, we are able to identify multiple problematic relations among these bug-prone files:

1. Dependency cycles. We found many bug-prone files exhibiting cyclic dependencies among them. For example, in Figure 8.8, *ExchangeHelper* (row4) and *DefaultExchange* (row3) structurally depend on each other. Different from other tools that detect cyclic dependencies, Titan also visualize the maintenance *penalty* on such relation. Titan shows that *ExchangeHelper* (row4) and *DefaultExchange* (row3) changed together 32 times in revision history. Similarly, *FileEndpoint* (row 22) and *FileConsumer* (row 21) also form a dependency cycle, and they changed together 56 times. For a particular note, not all dependency cycles are harmful. For example, *RouteDefinition* (row 15) and *ErrorHandlerBuilderRef* (row 14) also depend on each other, but they don't introduce any maintenance *penalty* in terms of evolutionary co-changes. By using evolutionary coupling among files as the secondary relation, our Titan tool can distinguish harmful and harmless cases.

2. Unhealthy inheritance. According to dependency inversion principle (abstractions should not depend on the implementation details), structural dependency from a parent class to its child manifest a problematic inheritance relation. For example, in Figure 8.8, *RouteDefinition* (row 15) "inherit" *ProcessorDefinition* (row 12), and in the meanwhile, *ProcessorDefinition* depends on its child class *RouteDefinition*. Given the problematic inheritance relation, these two files changed together 26 times in revision history.

3. Shared Secrets. Using evolutionary coupling as the secondary relation, Titan

automatically highlights cells in the matrix that contain evolutionary couplings without any direct structural dependencies, using a dark background color. For example, cell[r19,c18] says ",60", meaning *GenericFileProducer* and *GenericFile* changed together 60 times without any direct structural dependencies. Wong et al. [2011] first defined such phenomenon as modularity violation. In our prior case study of an agile comercial project (Schwanke et al. [2013]), we have showed that modularity violations usually suggests "shared secrets" among files that need to be better encapsulated.



	#b	br	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
1 ObjectHelper	60	1%	(1)	dp,																								
2 FileUtil	24	3%	dp,	(2)																								
3 DefaultExchange	17	7%	dp,18		(3)	dp,																						
4 ExchangeHelper	20	5%	dp,		dp,32	(4)																						
5 IntrospectionSupport	15	10%	dp,22				(5)																					
6 URISupport	14	10%	dp,					(6)																				
7 EndpointHelper	12	15%	dp,				dp,	dp,	(7)																			
8 ExpressionBuilder	13	12%	dp,26		,18					(8)																		
9 OnExceptionDefinition	14	10%	dp,							dp,	(9)			ih,			dp,											
10 OnCompletionDefinition	11	18%										(10)		ih,			dp,											
11 RouteDefinitionHelper	10	10%	dp,						dp,		dp,		(11)	dp,			dp,											
12 ProcessorDefinition	48	1%	dp,				dp,			dp,	dp,24	dp,24		(12)			dp,											
13 DefaultRouteContext	13	12%	dp,												(13)		ag,											
14 ErrorHandlerBuilderRef	12	15%	dp,													(14)	dp,											
15 RouteDefinition	17	7%	dp,										dp,16	ih,26	dp,	dp,	(15)											
16 GenericFileEndpoint	20	5%	dp,	dp,														(16)	dp,	dp,								
17 GenericFileConsumer	23	4%	dp,															ag,	(17)	dp,								
18 GenericFile	20	5%	dp,	dp,16																(18)								
19 GenericFileProducer	15	10%	dp,	dp,	dp,													ag,		,60	(19)							
20 FileOperations	19	6%	dp,	dp,																,36		(20)		ag,				
21 FileConsumer	33	2%	dp,	dp,														dp,	ih,	dp,74			(21)	dp,				
22 FileEndpoint	14	6%	dp,	dp,														ih,		dp,30	dp,	ag,	dp,56	(22)				
23 MulticastProcessor	58	1%	dp,			dp,																			(23)			
24 Splitter	13	12%	dp,			dp,18																	,20		ih,58	(24)		
25 XmlConverter	19	6%	dp,																								(25)	
26 XsltBuilder	17	7%	dp,			dp,																					ag,	(26)

Figure 8.8: Hadoop FileSystem Inherit DRSpace

We can't enumerate all possible problematic relations in the bug-prone DRSpaces because there are different combinations of leading files, primary and secondary relations to form a DRSpace. But, actually, we observed multiple problematic relations in each bug-prone DRSpace with a bug-prone leading file. We recognize them as problematic not only because they violate common design principles, but also that the involved files are both change- and bug-prone. Mo et al. [2015] reported that these problematic relations, such as cyclic dependencies, modularity violation, and unhealthy inheritance, have positive correlation with reduced quality and increased bug rates. Files that are involved in more problematic relations are more bug-prone compared to average files.

8.5 Limitations and Threats

Our evaluation for the usefulness of the DRSpace modeling is subject to *internal* threats to validity. First, to evaluate how bug-prone a particular DRSpace is, we calculated its ***dsb*** and ***bsc*** with regards to three levels of bug spaces: $Bug_{10\%}$, $Bug_{30\%}$, and $Bug_{100\%}$. We chose these three levels purely based upon experience and intuition. We consider files from $\mathrm{Bug}_{10\%}$ as very bug-prone, files from $\mathrm{Bug}_{30\%}$ as average bug-prone, while files from $\mathrm{Bug}_{100\%}$ as the least bug-prone. Due to individual difference of the studied subject, these three bug spaces may imply different bug-proneness levels within the subject's context (such as project problem domain and project management conventions). Thus, the selections of bug spaces potentially poses a threat to validity for the evaluation of the bug-proneness level of a DRSpaces. However, in general, we believe the choice of these three levels of bug spaces is reasonable. In the future, we plan to thoroughly test more sample levels of bug spaces for each individual subject.

Second, in the data reported thus far, we used all the history for each project to calculate evolutionary coupling and bug proneness. A prior work by Wong and Cai [2011] 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 five 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.

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 Bachmann et al. [2010] 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 15 open source projects, all of which are written in Java. The results could be different for projects implemented using other object-oriented programming languages. We plan to address this by investigating a more diversified set of projects in the future.

8.6 Summary

In this chapter, we introduced the DRSpace modeling, a new form of architecture representation that simultaneously captures the modular structure and the evolution coupling among files. We proposed that software architectures should be viewed and analyzed as multiple overlapping DRSpaces, because each DRSpace, formed using different types of primary and secondary relations, exhibits a meaningful and useful aspect of software architecture. Each of these structures promotes and supports a different kind of analysis. As the first step to bridging the gap between architecture and maintenance quality, the DRSpace modeling provides a perspective for inspecting the maintenance quality of software architecture as separate design spaces.

The study on 15 projects showed that, if a design rule is bug-prone, files in the DRSpace led by it are also likely to be bug-prone. In addition, by viewing the structural and evolutionary relations simultaneously, DRSpace modeling helps to reveal flawed structural relations, such as modularity violation, unstable interface, cyclic dependencies. We have shown that these flawed relations not only violate common design principles, but also have actually brought maintenance “penalties”.

Based on these observations, we suggest that the developer team should give higher priority to high-impact and bug-prone design rules, compared to average files. They should also be aware of the flawed architectural connections among bug-prone files. The DRSpace modeling, we envision, has the potential to increase such awareness.

9. Architectural Root (ArchRoot) Detection

Given all possible combinations of primary and secondary relations, there can be a large number of DRSpaces for describing the architecture of a software system. It is overwhelming, and more likely impossible, to inspect each and every DRSpace to look for architectural flaws that contribute to maintenance difficulties in a project. To reap the largest benefits in the bug fixing activities or even refactoring, the developer team should focus on the most bug-prone DRSpaces. The question is which DRSpaces are bearing the highest concentration of bug-prone files?

In this chapter, to answer this question, we propose an ArchRoot detection algorithm to automatically locate DRSpaces with the highest concentration of bug-prone files. First, we automatically extract the bug-prone files from the revision history and the bug tracking database of a software project. We call these bug-prone files a bug space. Then we reverse engineer the source code of a software system to generate a comprehensive set of DRSpaces. These DRSpaces are generated using each and every file in the project as a leading file (if there are 100 files in a project, in this way we generate 100 DRSpaces). The algorithm calculates the intersection between each DRSpace and the input bug space to identify a list of DRSpaces which, together, aggregate the files from the input bug space. We call these DRSpaces the Architectural Roots (ArchRoots) of bug-proneness. Each ArchRoot, which is also a DRSpace, can be represented by a DSM. It thus can be visualized and explored using Titan-GUI.

We applied the ArchRoot detection algorithm on 15 software projects. The analysis of the identified ArchRoots advanced our understanding of the impacts of software architecture on the maintenance quality. First of all, the bug-prone files seldom exist alone. Instead, a significant percentage (up to 91%) of the most (the top 30%) bug-prone files are architecturally connected in the top five ArchRoots. Some long-lived

ArchRoots survive multiple releases of a project. Their leading files keep aggregating a large number of bug-prone files over time. These ArchRoots are more bug-prone compared to random groups of files with equal sizes. With the help of Titan-GUI, we identified multiple, recurring architectural flaws in the long-lived ArchRoots, such as modularity violation, unstable interfaces, cyclic dependencies, and unhealthy inheritance. These flaws can keep propagating changes among files and make bugs hard to eradicate.

Based on the observations, we believe that the ArchRoots in a project tend to have persistent and significant impacts on the maintenance quality. The flawed architectural connections contained in the ArchRoots can be the root causes of bug-proneness and related high maintenance costs. We envision that the developers are unlikely to make a single file bug-free, without first fixing the flaws in an ArchRoot. Just like debts keep accumulating interest, these flaws, if not fixed, will keep incurring high maintenance costs. Hence, to improve maintenance quality in the long run, the developer team should consider refactoring to pay off the “debts”.

The following of this chapter is organized as follows. Section 9.1 defines Architectural Root. Section 9.2 introduces the ArchRoot detection algorithm. Section 9.4 shows the usefulness of ArchRoot detection algorithm in uncovering architectural root cusses contribute to the bug-proneness. Section 9.5 briefly discusses the limitations and threats to validity of the ArchRoot detection algorithm. Section 9.6 summarizes this chapter.

9.1 Architectural Roots of Bugginess

If a file is revised to fix bugs, we consider it as bug-prone. The more frequently it is revised to fix bugs, the more bug-prone it is. If a file is more bug-prone than 90% of all bug-prone files, we consider it the top 10% most bug-prone. We define a bug space as a set of bug-prone files. If a bug space is consisted of the top X% most bug-prone files, we call it $Bug_{X\%}$.

Suppose there are N files in a bug space $Bug_{X\%}$. Given a DRSpace DRS, if n files contained in it are also in $Bug_{X\%}$, we claim that DRS covers n/N of $Bug_{X\%}$ in the project. We have observed that a bug space can usually contain hundreds or even thousands of files, therefore it can intersect with multiple DRSpaces. Given a bug space $Bug_{X\%}$, we can calculate a ***minimal*** set of DRSpaces which cover all the files in $Bug_{X\%}$. These DRSpaces, together, capture all the architectural connections aggregating files in $Bug_{X\%}$. These connections, especially the problematic ones, could be the root causes of bug-proneness. Intuitively, we call this minimal set of DRSpaces the ***architectural roots (ArchRoots)*** of $Bug_{X\%}$.

In order to calculate the ArcRoots of $Bug_{X\%}$, we define the following two parameters to describe the intersection between a set of DRSpaces, S=$\{S_1, S_2, ..., S_n\}$, and $Bug_{X\%}$:

1. ***Coverage***: the *coverage* of S=$\{S_1, S_2, ..., S_n\}$ on $Bug_{X\%}$ can be calculated as:

$$coverage_{S,Bug_{100\%}} = \frac{|(S_1 \cup S_2 ... \cup S_n) \cap Bug_{X\%}|}{|Bug_{X\%}|} \tag{9.1}$$

   $|(S_1 \cup S_2 ... \cup S_n) \cap Bug_{X\%}|$ is the number of files in the intersection between S and $Bug_{X\%}$. $|Bug_{X\%}|$ is the number of files in $Bug_{X\%}$.

2. ***LOC Normalized Coverage (LocCoverage)***. Ostrand et al. [2004] found that files with large lines of code (loc) are more likely to be bug-prone compared to small files. As a result, a set of *DRSpaces* with a high *coverage* can simply because they contain a set of very large files. In order to reduce the interference of *loc*, we define *LocCoverage* as the coverage of a set of *DRSpaces* normalized by the *loc* of each file. The *LocCoverage* of S=$\{S_1, S_2, ..., S_n\}$ on $Bug_{X\%}$ can be

calculated by the following steps:

(a) For each file f in a project, we compute its *loc* normalized bug-proneness:

$$f_{weight} = \frac{f_{Bug\_Fix\_Frequency}}{f_{loc}} \tag{9.2}$$

$f_{Bug\_Fix\_Frequency}$ is the number of bug fixes involving f.

f_{loc} is the lines of code in f.

f_{weight} represents the bug-proneness of f normalized by its lines of code.

(b) We compute the sum of the *loc* normalized bug-proneness on all the files in $Bug_{X\%}$:

$$W_{Bug_{X\%}} = \sum_{\forall\ f \in Bug_{X\%}} f_{weight} \tag{9.3}$$

(c) We compute the sum of the *loc* normalized bug-proneness on the files in the intersection between S=$\{S_1, S_2, ..., S_n\}$ and $Bug_{X\%}$:

$$W_{S \cap Bug_{X\%}} = \sum_{\forall\ f \in (S_1 \cup S_2 ... \cup S_n) \cap Bug_{X\%}} f_{weight} \tag{9.4}$$

(d) The ***LocCoverage*** of S=$\{S_1, S_2, ..., S_n\}$ on $Bug_{X\%}$ is computed using the equation 9.4 divided by the equation 9.3 :

$$LocCoverage_{S,Bug_{X\%}} = \frac{W_{S \cap Bug_{X\%}}}{W_{Bug_{X\%}}} \tag{9.5}$$

The ArchRoots of $Bug_{X\%}$ satisfied the following conditions: (1) $Coverage_{ArchRoots,Bug_{X\%}}$ equals to 100%; (2) ArchRoots contain a minimal number of DRSpaces; (3) the top few ArchRoots have a maximal possible *coverage* on $Bug_{X\%}$.

9.2 Detection Algorithm

As the first step to detecting the most problematic DRSpaces, we just bluntly assume that each file in a project could be a high-impact design rule, potentially connecting a large number of bug-prone files. Thus, we use each and every file in a project as the leading file to generate a DRSpace. In each DRSpace, we treat all the types of structural dependencies as the primary relations and the evolutionary couplings as the secondary relation. This set of DRSpaces comprehensively capture all the files and their connections in a project.

ALGORITHM 1: genDRSpaces (DSM)

```
1: $DRSpaceSet \leftarrow \emptyset$
2: $DrhCluster \leftarrow$ DRHClustering($DSM$)
3: for each $file$ in $DSM$ do
4:     $newSpace \leftarrow \emptyset$
5:     $LeadingFile \leftarrow file$
6:     $newSpace.add(LeadingFile)$
7:     for each $Module$ in $DrhCluster$ do
8:         if $Module$ has structural dependences on $LeadingFile$ then
9:             $newSpace.add(Module)$
10:        end if
11:    end for
12:    $DRSpaceSet$.add($newSpace$)
13: end for
14: return $DRSpaceSet$
```

Algorithm 1 shows the procedure for generating such a DRSpace set of a project. The input is the DSM of the project, reversed engineered from the source code. The DRH algorithm (introduced in chapter 6) in line 2 clusters the input DSM into modules based on the structural dependencies among files. Each module is a group of tightly coupled files with strong structural dependencies. From line 3 to line 13, using each file in the input DSM as a leading file, the algorithm generates a DRSpace composed of the leading file and all the modules that depend on the leading file. The new DRSpace is added to the return value, *DRSpaceSet*, which is a comprehensive set of DRSpaces covering all the files and their connections in a project.

Then, which DRSpaces, from *DRSpaceSet*, together have the highest concentration of the bug-prone files. To answer this question, we propose an *ArchRoot* detection algorithm to identify a list of DRSpaces $AR = [R_1, R_2, ..., R_m]$, covering all the files and connections in a given bug space, $Bug_{X\%}$. The algorithm calculates and ranks the intersection between each DRSpace from the comprehensive set and $Bug_{X\%}$. The top few DRSpaces together have a maximal $Coverage$ on $Bug_{X\%}$. The *ArchRoot* detection algorithm is efficient and greedy in identify a list of bug-prone DRSpaces. Although it doesn't identity the *minimal* list of DRSpaces that concentrates bug-prone files, our study (we will discuss in detail later) shows that this algorithm is very helpful in identifying problematic DRSpaces that worth attention.

ALGORITHM 2: ArchRootDetection ($DRSpaceSet$,$Bug_{X\%}$)

```
1: $AR \leftarrow \emptyset$;
2: $BugSpace2Cover \leftarrow Bug_{X\%}$;
3: while $BugSpace2Cover \neq \emptyset$ do
4:     $MaxCoverageSpace \leftarrow$ SelectMaxCoverange($DRSpaceSet, BugSpace2Cover$);
5:     $AR$.Add2Tail($MaxCoverageSpace$);
6:     $DRSpaceSet$.Remove($MaxCoverageSpace$);
7:     $BugSpace2Cover$.RemoveAll($MaxCoverageSpace$.Files());
8:     CalculateCoverage($AR, Bug_{X\%}$);
9:     CalculateLocCoverage($AR, Bug_{X\%}$);
10: end while
11: return $AR$;
```

Algorithm 2 displays the pseudo-code of *ArchRoot* detection algorithm. First of all, the output, AR, is initialized to be an empty list. The input $Bug_{X\%}$ is copied to $BugSpace2Cover$ denoting the remaining bug-prone files that are not covered by AR yet. In each iteration of the while loop from line 3 to line 10, first, a DRSpace with a maximal coverage on $BugSpace2Cover$ is selected from $DRSpaceSet$ and added to AR (line 4 and 5); then the selected DRSpace is removed from $DRSpaceSet$ and the files covered by it are removed from $BugSpace2Cover$ (line 6 and 7); finally, the $Coverage$ and $LocCoverage$ of current AR with regards to $Bug_{X\%}$ is updated. The algorithm terminates when $BugSpace2Cover$ becomes empty, meaning all the files in the original $Bug_{X\%}$ have been coverd by AR.

Although a bug space can contain hundreds and even thousands of files, the ArchRoot detection algorithm can automatically identify a relatively small number of DRSpaces that capture the majority of bug-prone files. For example, in open source project Camel, the top 20 ArchRoots cover more than 70% of the 496 bug-prone files in $Bug_{30\%}$. In other words, 350 of the 496 files are architecturally aggregated in only 20 DRSpaces. Since each DRSpace is a group of architecturally connected files, the result actually implies the existence of strong architectural connections among the majority of the top bug-prone files in Camel. More importantly, we observed multiple architectural flaws in the ArchRoots. We will illustrate and discuss the characteristics of ArchRoots we identified in greater detail in Section 9.4.

9.3 Tool Support

Our approach is supported by our toolset, Titan. Figure 9.1 depicts an overview of the toolset. Titan consists of 4 data processing components and 1 visualization component. The visualization component, TitanGUI, which visualizes the output of the data processing components, has been introduce in greater detail in section 8.3. In this section, we will focus on introducing the other components.

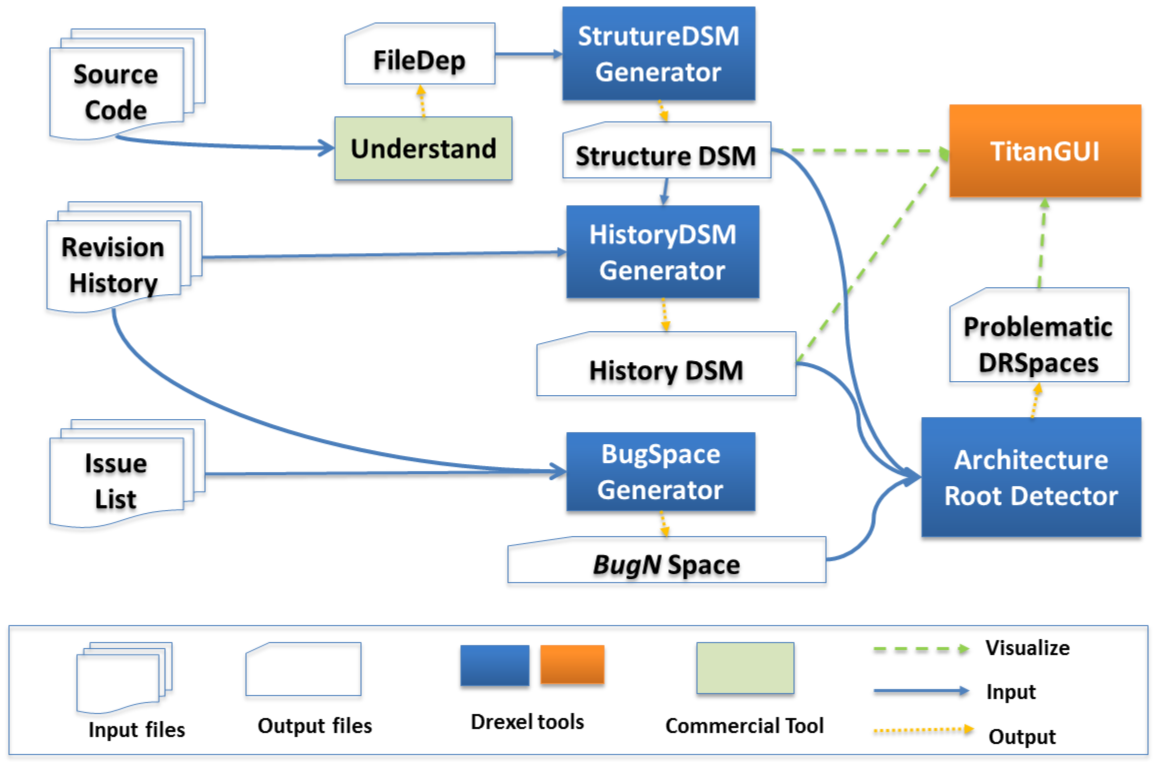


Figure 9.1: Titan Tool Chain

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 between which the revision history should be considered in the computation. Evolutionary coupling is exported into a history dsm, also in the form of a *.dsm* file, that records the co-change frequency between two files in a project.

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

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 specified 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.

TitanGUI TitanGUI is an interactive design structure matrix (DSM) user interface 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.

9.4 ArchRoots Analysis

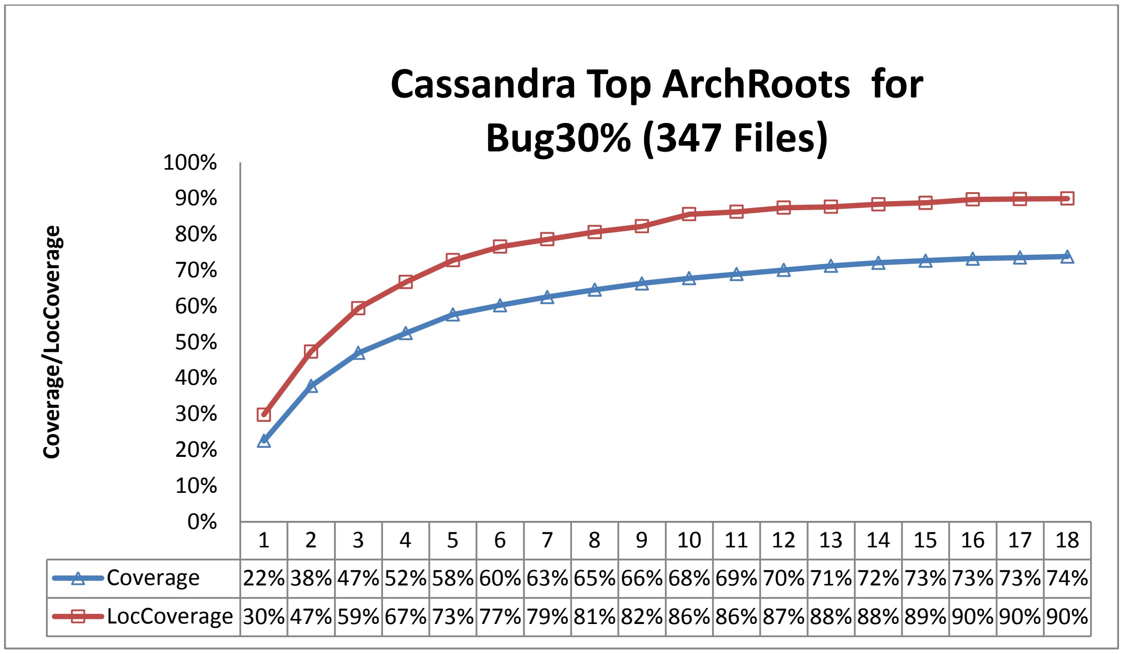
We applied the ArchRoot detection algorithm on the 15 open source projects listed in Table 8.1 in Section 8.4, and analyzed how ArchRoots aggregate bug-prone files. Despite the different characteristics of projects, we made consistent observations.

First, the top few (usually five) ArchRoots can cover a significant portion of bug-prone files, indicating strong architectural connections among a large number of bug-prone files in these projects. In addition, we discovered long-lived ArchRoots that persistently aggregating a large number of bug-prone files in multiple releases of each project. To reap the largest benefits when developers are trying to fix bugs, or even to refactor their codes, they should give the highest priority to the long-lived roots. Lastly, in each ArchRoot, we identified multiple, recurring architectural flaws, such as cyclic dependencies and modularity violations, that violate common design principles and indeed incurred maintenance "penalties" in the form of high change-/bug- rates. We believe that these architectural flaws could be the root causes of bug-proneness. We will present the details of ArchRoot analysis based on 15 open source projects in the following subsections.

9.4.1 Concentration of Bug-proneness

In a project, there can be hundreds and even thousands of bug-prone files. For example, as shown in Table 9.1, there are from 135 (ZooKeeper) to 2475 (OpenJPA) bug-prone files in the 15 open source projects. These bug-prone files are extracted from the revision history and bug tracking database of each project. For each project, we rank all the bug-prone files according to the number of times each is revised to fix bugs. Based on the ranking, we discriminate three different levels of bug spaces: $Bug_{100\%}$, $Bug_{30\%}$, and $Bug_{10\%}$, denoting the sets of all, the top 30%, and the top 10% most bug-prone files respectively. The sizes (number of files) of $Bug_{100\%}$, $Bug_{30\%}$, and $Bug_{10\%}$ of each project are shown in column 2, 4, and 6 respectively.

Given a bug space, the ArchRoot detection algorithm locates a set of DRSpaces concentrating the bug-prone files. By definition, each DRSpace is a group of architecturally connected files. Therefore, if only a few DRSpaces concentrate a large number of bug-prone files, it indicates that these bug-prone files are architecturally connected in only a few groups. Actually, our study shows that, in each project, a significant portion of bug-prone files are usually connected in only five DRSpaces. For example, Figure 9.2 visualizes the trend of the *Coverage* and *LocCoverage* by up to the top 18 ArchRoots for $Bug_{30\%}$ in Cassandra. The x-axis represents the top x ArchRoots, while the y-axis represents the *Coverage* and *LocCoverage* by the top x ArchRoots. In Cassandra, the top five ArchRoots cover 58% of the top 30% most bug-prone files (347 files). For a particular note, as the number of ArchRoots doubles from the top five to the top ten, the *Coverage* and *LocCoverage* only increase from 58% to 68% and from 73% to 86% respectively. It indicates that the top few ArchRoots have the highest concentration of bug-prone files, while the following ArchRoots contain more scattered bug-prone files.



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

Figure 9.2: Cassandra ArchRoots Coverage

We made consistent observations in all the 15 projects: the top five ArchRoots usually cover a significant portion of bug-prone files. Table 9.1 shows the $Coverage$ and $LocCoverage$ of the top five ArchRoots for $Bug_{100\%}$, $Bug_{30\%}$, and $Bug_{10\%}$ for the 15 projects. There are 135 (ZooKeeper) to 2475 (OpenJPA) files revised to fix bugs (shown in column $\#Fls$ under $Bug_{100\%}$) in these projects. The top five ArchRoots concentrate 18% (OpenJPA) to 77% (MINA) of these bug-prone files. Although the $Coverage$ by the top five ArchRoots in OpenJPA is relatively low (which is only 18%) compared to other projects, the top five ArchRoots actually concentrate 446 bug-prone files (calculated by 2475*18%, since there are totally 2475 bug-prone files in OpenJPA).

Admittedly, files with only a few bug fixes may not be truly bug-prone. They can be false-positives due to fixes for arbitrary reasons. In addition to $Bug_{100\%}$, we analyzed the top five ArchRoots for $Bug_{30\%}$ and $Bug_{10\%}$ as well, for $Bug_{30\%}$ and $Bug_{10\%}$ are less likely to be false-positive. It turns out that the top five ArchRoots also concentrate a significant portion of $Bug_{30\%}$ and $Bug_{10\%}$ (even higher than the $Coverage$ on $Bug_{100\%}$). As shown in the "$Bug_{30\%}$" column in Table 9.1 , from 35% (OpenJPA) to 91% (MINA) of the top 30% most bug-prone files are concentrated in the top five ArchRoots. Actually, in MINA, the top four ArchRoots already cover 91% of the bug-prone files (marked as 94%*(4) in Table 9.1). Similarly, as shown in column "$Bug_{10\%}$", from 54% (Camel) to 95% (ZooKeeper) of the top 10% most bug-prone files (from 16 files Avro to 254 files in CXF) are concentrated in the top five ArchRoots.

In summary, the top five ArchRoots can usually concentrate a significant portion of the top ranked bug-prone files. For a particular note, the $LocCoverage$ in general is consistent with $Coverage$ as shown in Table 9.1. This means that the concentration of bug-proneness in the top five ArchRoots is not merely because these roots contain a set of very large files. We propose, based on our observations, that when developers are trying to fix bugs, they should treat the bug-prone files as connected groups, instead of as a large number of individuals. The developers should also be aware of the architectural connections among the bug-prone files. To reap the largest benefits in bug fixing, or even refactoring, activities, the developers should especially pay attention to the top few ArchRoots with the highest concentration of bug-proneness.

Table 9.1: Loc_Coverage (LC) by the Top Five *ArchRoots*

Projects	$Bug_{100\%}$			$Bug_{30\%}$			$Bug_{10\%}$		
Release #	#Fls	Cov.	LocCov.	#Fls	Cov.	LocCov.	#Fls	Cov.	LocCov.
Avro (1.7.6)	148	63%	52%	46	89%	89%	16	94%*(2)	95%*(2)
Camel (2.12.4)	1553	23%	20%	496	38%	31%	194	54%	41%
Cassandra (2.1.2)	997	37%	51%	347	58%	82%	161	67%	81%
CXF (3.0.0)	1740	23%	19%	492	43%	35%	254	56%	57%
Derby (10.11.1.1)	910	45%	29%	318	60%	49%	130	74%	57%
Hadoop (2.5.0)	601	28%	60%	209	41%	44%	73	67%	63%
HBase (0.98.2)	1246	25%	19%	521	40%	30%	160	64%	64%
Mahout (0.9) (0.9)	750	31%	42%	312	43%	50%	149	56%	68%
MINA (1.1.7)	189	77%	88%	59	91%*(4)	94%*(4)	18	94%*(2)	99%*(2)
OpenJPA (2.2.2)	2475	18%	14%	770	35%	18%	210	79%	56%
PDFBox (1.8.7)	466	65%	59%	270	69%	62%	101	68%	91%
Pig (0.9.2)	699	49%	52%	196	66%	85%	91	68%	91%
Tika (1.7)	209	62%	59%	81	81%	86%	31	87%	88%
Wicket (6.19.0)	1321	27%	26%	411	43%	45%	194	57%	60%
ZooKeeper (3.4.5)	135	67%	62%	51	86%	62%	21	95% *(3)	95%*(3)

*(n) means that the *Cov.* and *LocCov.* is by the top n (n ¡ 5) DRSpaces instead of the top five DRSpaces because the coverage is already maximal.

9.4.2 Long-lived ArchRoots

During the life cycle of a software project, the architecture evolves and the bug-prone files varies from release to release. We have shown in Table 9.1 that, in a single release of a project, the top few (usually five) ArchRoots have high concentration of bug-proneness. More interestingly, how the top few, say five, ArchRoots vary from release to release? Thus, we analyzed the evolution of the top five ArchRoots identified in multiple releases of each project. We recognize the ArchRoots, identified in multiple releases but with the same leading file, as different snapshots of the same ArchRoot. The number of snapshots of an ArchRoot is its *Age*. For example, if a project has totally ten releases, the maximal possible age of an ArchRoot is ten. If an ArchRoot has five snapshots, its age is five. If an ArchRoot persists more than 40% of the total releases, we consider it as a long-lived ArchRoot. We believe that an ArchRoot surviving multiple releases deserves more attention than an ArchRoot only appeared in a few releases.

In each project, we identified several long-lived ArchRoots, which are also among the top five in each release. In Table 9.2, we listed the leading file, the age, the bug-prone ranking of the leading file, the $LocCoverage$, and the DSB (Design Space Bugginess is the percentage of files in a DRSpace that are also contained in a bug space) of each ArchRoot. We can make the following observations from this table.

First, a long-live ArchRoot is usually led by a bug-prone leading file: 35 of the 47 *long-lived ArchRoots* are led by a bug-prone leading file (The leading files are at least the top 52% most bug-prone). Furthermore, there are 31 and 15 ArchRoots led by a top 30% and a top 10% most bug-prone leading file respectively. This re-enforced our understanding of the influences of a bug-prone and high-impact design rule: it can **persistently** aggregate bug-prone files throughout the entire life cycle of a project.

Second, except two, for each long-lived ArchRoot, the $LocCoverage$ w.r.t $Bug_{30\%}$ is on average three times that of a random group containing the same number of files. Also, the DSB of each long-lived ArchRoot is on average three times that of a random group with an equal number of files. For a particular note, there are five long-lived ArchRoots, each covering more than half of the files from $Bug_{30\%}$. In addition, for 36 long-lived ArchRoots, the DSB is at least 20%, meaning at least one in every five files in each root is from $Bug_{30\%}$. This indicates that the long-lived ArchRoots are usually very bug-prone, and specifically, they are more bug-prone than average groups of files.

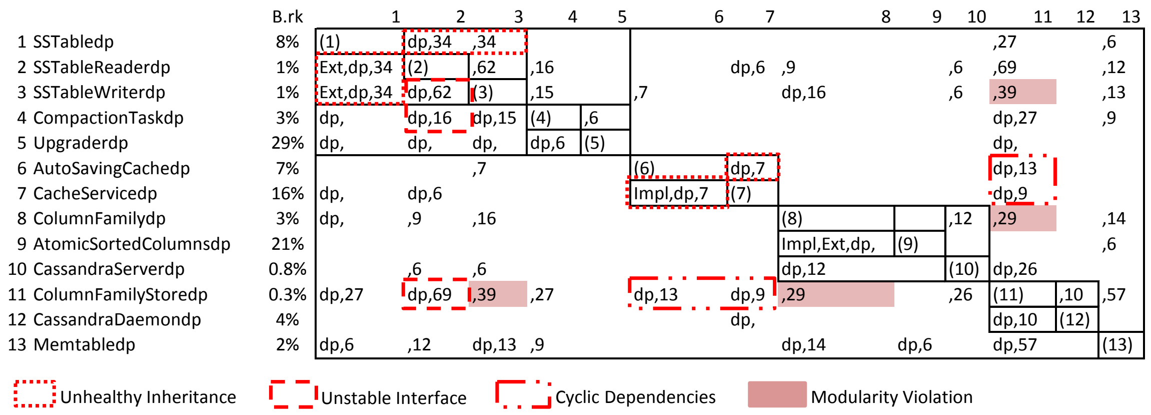
In summary, in each project, there exist several long-lived ArchRoots remaining the top five most bug-prone in multiple releases of a project. They are usually led by a bug-prone leading file. They are more bug-prone than random groups of files, covering and containing higher percentages of the top 30% most bug-prone files. The implication is that the most bug-prone ArchRoots can have persistent impact on the bug-proneness of a project. A long-lived ArchRoot, especially if it is led by a bug-prone design rule, can keep aggregating a large number of bug-prone files in multiple releases of a project. Higher bug rates can "grow" out of long-lived roots over time. To fundamentally reduce the over-all bug rates on files in the long run, long-lived ArchRoots, especially those led by a bug-prone design rule, should be granted the top priority for developer team to fix, or even refactor.

9.4.3 Architectural Flaws

In order to understand the root causes in ArchRoots that contribute to the bug-proneness, we investigated the architectural connections among files in the top *ArchRoots* using our Titan-GUI. We found that *ArchRoots* usually contain multiple, recurring architectural flaws, such as cyclic dependencies, unhealthy inheritance, unstable interfaces, and modularity violations. We consider these connections as architectural flaws, not only because they violate common design principles, but also because they indeed incur high co-changes.

In Figure 9.3, we illustrate the architectural flaws identified in part of a root identified in Cassandra. Using this as an example, we will qualitatively analyze how different flaws contribute to the high bug rates on files.

First, there exists unhealthy inheritance between the parent class *SSTabledp* (row 1) and its child class *SSTableReaderdp* (row 2). *SSTableReaderdp* and *SSTableWri-*



	B.rk	1	2	3	4	5	6	7	8	9	10	11	12	13
1 SSTabledp	8%	(1)	dp,34	,34								,27		,6
2 SSTableReaderdp	1%	Ext,dp,34	(2)	,62	,16			dp,6	,9		,6	,69		,12
3 SSTableWriterdp	1%	Ext,dp,34	dp,62	(3)	,15		,7		dp,16		,6	,39		,13
4 CompactionTaskdp	3%	dp,	dp,16	dp,15	(4)	,6						dp,27		,9
5 Upgraderdp	29%	dp,	dp,	dp,	dp,6	(5)						dp,		
6 AutoSavingCachedp	7%			,7			(6)	dp,7				dp,13		
7 CacheServicedp	16%	dp,	dp,6				Impl,dp,7	(7)				dp,9		
8 ColumnFamilydp	3%	dp,	,9	,16					(8)		,12	,29		,14
9 AtomicSortedColumnsdp	21%								Impl,Ext,dp,	(9)				,6
10 CassandraServerdp	0.8%		,6	,6					dp,12		(10)	dp,26		
11 ColumnFamilyStoredp	0.3%	dp,27	dp,69	,39	,27		dp,13	dp,9	,29		,26	(11)	,10	,57
12 CassandraDaemondp	4%							dp,				dp,10	(12)	
13 Memtabledp	2%	dp,6	,12	dp,13	,9				dp,14	dp,6		dp,57		(13)

Figure 9.3: Architectural Flaws in a Root in Cassandra

terdp extend and depend on class *SSTabledp* (cell[r2,c1] and cell[r3,c1]). But there is an inverted dependency from *SSTabledp* to its child class *SSTableReaderdp* (cell[r1:c2]). Martin [2003] states in the dependency inversion principle that abstractions should not depend on details and details should depend on abstractions. Therefore, an interface or abstract class should usually not depend on the concrete classes (except for particular circumstances that embrace the opposite, such as the template design pattern). Based on the reasoning, we consider the inverted dependency from the parent class to its child as a potentially flawed architectural connection. The history co-change shows that the parent class *SSTabledp* changed together with its child classes *SSTableReaderdp* and *SSTableWriterdp* 34 times in history.We assume that whenever one of the files in this unhealthy inheritance changes, the change will propagate to the other files in the inheritance relation, increasing the bug rates on these files. As shown in Column "B.rk", *SSTabledp* ranks the top 8% most bug-prone, and the two child classes rank the top 1% most bug-prone, among all other files.

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

There are also cyclic dependencies between *ColumnFamilyStoredp* (row 11) and two files: *AutoSavingCachedp* (row 6) and *CacheServicedp* (row 7). As shown on cell[r6,c11] and cell[c11,r6], *ColumnFamilyStoredp* and *AutoSavingCachedp* form a structural dependency cycle with each other. Similarly, there are a cyclic dependencies between *ColumnFamilyStoredp* and *CacheServicedp* (showin in cell[r7,c11] and cell[r11, c7]). Whenever, one file in a cycle changes, it is likely to propagate changes to other members in the cycle, accumulating the over-all maintenance costs. The history co-change indicates that *ColumnFamilyStoredp* changed together with *AutoSavingCachedp* and *CacheServicedp* 13 and 9 times respectively.

Last but not least, there are modularity violations among *ColumnFamilyStoredp* (row 11), *SSTableWriterdp* (row 3), and *ColumnFamilydp* (row 8). Modularity violation was first proposed by Wong et al. [2011] as the phenomenon where a set of files frequently change together in revision history without having any structural dependencies. In this case, *ColumnFamilyStoredp* has no structural dependencies with *SSTableWriterdp* or *ColumnFamilydp*, but it changes with them 39 (cell[r4, c11] and cell[r11,c4]) and 29 (cell[r8, c11] and cell[r11,c8]) times respectively. In a prior case study (Schwanke et al. [2013]), we found that modularity violation usually implies shared concepts between files that would benefit from a better encapsulation design. Whenever the shared concept changes in one file, the other files have to accommodate the change. Unless encapsulate the concept, files sharing the concept tend to change together frequently, causing bug rates to increase.

With numerous architectural flaws, it is not surprising to see from column "B.rk" that files in this root all rank above the top 30% most bug-prone. Actually, except three files, *Upgraderdp*, *CacheServicedp*, and *AtomicSortedColumnsdp*, all other files rank above the top 10% most bug-prone.

We can't enumerate all the architectural flaws in each ArchRoot. Mo et al. [2015] have verified that the flawed architectural connections among files have strong positive correlation with increased bug rates in software projects. A file involved in multiple architectural issues are more likely to be bug-prone than average files. The qualitative analysis supplements that these flaws could be the root causes of high bug rates, because they can propagate changes among files, making bugs hard to eradicate. The take away message is that, it is **unlikely** that a developer can make a single file bug-free without also fixing the other files that architecturally connect to it. For example, in order to fix bugs involving a set of files with a dependency cycle, the developers should probably first cut the cycle to prevent the changes from propagating. In summary, in order to fundamentally reduce the bug rates, the developer team should consider fixing these flaws first, probably by refactoring. Otherwise, these flaws are likely to keep incurring high bug rates and maintenance costs over time.

9.5 Limitations and Threats

Although the ArchRoot detection algorithm has shown great potential in revealing the architectural root causes of error-proneness, there is, in any research, limitations and threats. In this section, we will discuss the limitations and threats to validity.

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

Second, a bug space, used as an input to the *ArchRoot* detection algorithm, is extracted from a project's revision history by matching bug ticket IDs in commit messages. In reality, the links between commits and bug tickets may be missing due to various reasons. For example, the developers may fail to link their commits to the specific bug tickets simply because they forget to do so. Therefore, a project with a small bug space may not, therefore, truly be high quality. For projects with low bug tagging rates, we can examine the change spaces, instead of the bug spaces. A change space is a list of change-prone files ranked by their change-prone levels. Our approach can, in such a case, identify the architectural roots of change-proneness.

Third, the investigation of long-lived *ArchRoots* requires ample revision history. The 15 open source projects we studied have revision history covering four to eight years. The adequacy of revision history allowed the investigation of the evolution of *ArchRoots* over time. For software projects with substantially shorter revision histories, such an analysis cannot be conducted.

9.6 Summary

In this chapter, we proposed an algorithm to *automatically* locate a list of DRSpaces that architecturally aggregate the bug-prone files in a project. We consider these

DRSpaces as the architectural roots of bug-proneness, thus we call them the ArchRoots. The ArchRoots deserve special attention in maintenance activities due to their contributions to the reduced maintenance quality.

According to the study of 15 open source projects, we found that a significant percentage of bug-prone files in these projects are architecturally connected in only a few (usually five) ArchRoots, instead of being isolated from each other. Some ArchRoots survive multiple releases of a project. Their leading files, which are usually very bug-prone as well, keep aggregating a large number of bug-prone files over time. Consequently, these long-lived ArchRoots are more bug-prone than average groups of files. In addition, we observed multiple, recurring flawed architectural connections in these ArchRoots, such as cyclic dependencies, unhealthy inheritance, modularity violations, and unstable interfaces. Due to such flawed connections, it is difficult for the developers to make a single file bug-free, without also revising the other bug-prone files in an ArchRoot. As long as the flawed architectural connections are not fixed, the maintenance costs will keep accumulating over time, just like how debts accumulate penalties.

Based on the above observations, we believe that the flawed architectural connections are the root causes of maintenance difficulties. In order to reduce bug rates in the long run, the developer team should consider paying off the “debts” first, probably in the way of refactoring to fix the architectural flaws. The long-lived ArchRoots, with significant and persistent impacts on the maintenance quality, should be granted the top priority in maintenance activities.

Table 9.2: Long-lived ArchRoots for $Bug_{30\%}$

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

DSB stands for Design Space Bugginess, which is the percentage of top 30% most bug-prone files an ArchRoot contains. Avg. stands for Average. Std. stands for Standard Deviation. LocCoverage.Rate is the LocCoverage of an ArchRoot divided by the LocCoverage of a random group of files with an equal size. Similarly, DSB.Rate is the DSB of an ArchRoot divided by that of a random group of files with an equal size.

10. Architectural Debt (ArchDebt)

We have observed that maintenance costs will keep increasing as long as the flawed architectural connections are not fixed. Thus, the flawed architectural connections are like “debts” that need to be paid off. Otherwise, “penalties”, in terms of high change- or bug-rates, will keep accumulating. The term “technical debt” (TD), first proposed by Cunningham [1992], has been used as a metaphor to describe the consequences of shortcuts taken in software development to achieve immediate goals. In this chapter, based upon our observations from prior chapters, we define a particular form of TD—an Architectural Debt (ArchDebt)—as a group of architecturally connected files that incur high maintenance costs over time due to their flawed architectural connections. In the case of an ArchDebt, the developers sacrifice the long-term maintenance quality by postponing refactoring their codes to fundamentally fix the architectural flaws. However, they may be subject to higher future maintenance “penalties”.

An ArchDebt, as a special form of TD, has essential differences from debts in real life. First, in real life, we know what debts we have. But, in software architecture, we don’t know which and how files are involved in ArchDebts. Although an ArchRoot usually contains multiple architectural flaws with high maintenance costs over time, we can’t treat an ArchRoot directly as a debt, because an ArchRoot contains both high maintenance and normal files. The diagnosing of architectural flaws in an ArchRoot still requires a certain level of expertise and manual inspection. Second, in real life, we know how much each debt costs and its interest rate. But, it is not clear how much an ArchDebt has cost in a project, or how fast the maintenance costs will accumulate in the future (the interest rate). Without knowing these key parameters—the costs and the interest rates—of ArchDebts, the developer team can’t make informed decisions for their project: whether, when, and where they should invest in a refactoring to pay off a "debt".

In this chapter, we provide an approach to automatically identify ArchDebts, quantify and model the growing trend of such "debts". To *automatically* identify groups of files that are *true* ArchDebts, we define four typical architectural flaw patterns, which capture all possible combinations of structural and evolutionary relations among files. We can identify groups of files involved in "debts" by matching these patterns. To better model evolutionary coupling, we develop a novel History Coupling Probability (HCP) matrix, which models the evolutionary coupling between files using probabilities of change propagation between files. We use the propagation probabilities to replace the simple co-change numbers in the original DRSpace modeling. In addition, the actual maintenance costs on files, such as the monetary costs or human labor hours, can't be accurately measured. Thus, we approximately quantify the maintenance costs on files involved in an ArchDebt by the lines of code revised to fix bugs in them. Finally, to answer what's the interest rate on each ArchDebt, we monitor the change of maintenance costs spent on each ArchDebt over time. We use four types of regression models to describe four typical types of interest rate: linear, exponential, logarithmic, and polynomial regression models for stable, increasing, decreasing and fluctuating interest rates respectively.

We applied the ArchDebt approach on seven open source projects. We found that the ArchDebts consume up to 85% of the total bug fixing effort in these projects. Most interestingly, the most expensive and high-impact ArchDebts don't involve any direct structural dependencies among files. Instead, groups of files are heavily coupled with each other in revision history. It indicates the lack of sufficient design to encapsulate change-/bug-prone concepts shared among files. Finally, we found that the majority of ArchDebts have a stable interest rate over time, meaning during each release cycle, the developer team have to spend a stable amount of costs to fix bugs in an ArchDebt.

As the last step in this dissertation to bridging the gap, the ArchDebt approach enables software practitioners to diagnose and manage architectural flaws—the root causes of bug-proneness and high maintenance costs—in a systematic way. The identification and quantification of ArchDebts have pushed forward the TD concept from a metaphor toward an actionable practice. Our approach can not only automatically locate groups of files as ArchDebts, but also quantify the "costs" and the "interest rate" of each debt. Informed decisions can be made, in terms of whether, where, and when to invest in a refactoring to fundamentally increase software quality.

The rest of this chapter is organized as following. Section 10.1 formally defines Architectural Debt (ArchDebt). Section 10.2 introduces the approach to identify ArchDebts, quantify the maintenance costs of such debts, and model the interest rate of such debts. Section 10.3 evaluates the usefulness of the ArchDebt approach in identifying true debts. Section 10.4 discusses the interest rate and evolution of ArchDebts over time. Section 10.5 discusses limitations and threats to validity for our ArchDebt quantification approach. Section 10.6 concludes this chapter.

10.1 ArchDebt Definition

An Architectural Debt is a group of architecturally connected files that incur high maintenance costs over time due to their flawed architectural connections.

An Architectural Debt Formal Definition. We first formally define software architecture of a system, implemented at release r, as a set of overlapping DRSpaces:

$$SoftArch_r = \{DRSpace_1, DRSpace_2, ..., DRSpace_n\} \tag{10.1}$$

where n is the number of DRSpaces, each revealing a different aspect of the architecture, e.g., each dependency type can form a distinct DRSpace, which was illustrated in section 8.2.

We define an *Architectural Debt (ArchDebt)* as a group of *architecturally connected files* that incur high maintenance costs over time due to their flawed connections, as follows:

$$ArchDebt =< FileSetSequence, DebtModel > \tag{10.2}$$

The first element, *FileSetSequence*, is a sequence of file groups, each extracted from a different project release:

$$FileSetSequence = (FileSet_1, FileSet_2..., FileSet_m) \tag{10.3}$$

where m is the number of releases that *ArchDebt* impacts, $m \leq R$, the total number of system releases. $FileSet_r$, $r = 1...m$ is an architecturally connected file group in release r. The number of files in each $FileSet$ may vary in different releases.

The second element, *DebtModel* is a formula capturing the growing trend, i.e. interest rate, of the architecture debt, in the form of maintenance costs spent on $FileSetSequence$.

10.2 Identifying and Quantifying ArchDebts

Given the formal definition of *ArchDebt*, we will first identify $FileSetSequence$, and then build a $DebtModel$ to capture the "interest rate" based on the costs $FileSetSequence$ has incurred. Since there are numerous DRSpaces in each release, and numerous file groups in each DRSpace that can be debt candidates, we illustrate our process of searching for *FileSetSequence* as an analogy to searching for a specific web page on the internet, consisting of the following steps as shown in Figure 10.1:

1) Crawling: this step collects a subset of DRSpaces from each $SoftArch_r$, r from 1 to R, similar to crawling and collecting web pages.

2) Indexing: this step identifies (indexes) a specific file group, *FileSet*, from each *DRSpace* selected in the first step, then locate sequences of related *FileSets* in different releases as a *FileSetSequence*.

3) Modeling: we measure the maintenance costs incurred by each $FileSet_r$ in a sequence, and model the cost variation. An *ArchDebt* is a *FileSetSequence* whose costs increase over time.

4) Ranking: we rank the severity of each *ArchDebt* according to the amount of maintenance costs they have accumulated in the project's evolution history.

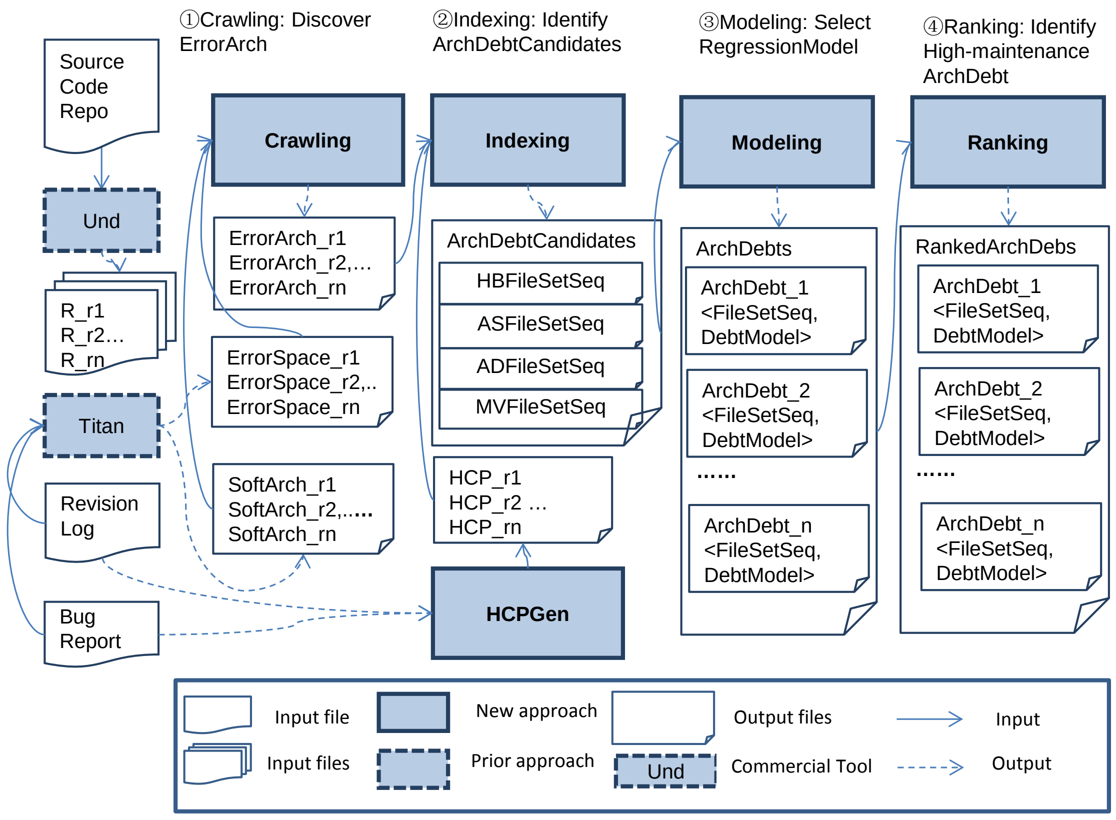

①Crawling: Discover ErrorArch
②Indexing: Identify ArchDebtCandidates
③Modeling: Select RegressionModel
④Ranking: Identify High-maintenance ArchDebt
Source Code Repo
Und
R_r1 R_r2… R_rn
Titan
Revision Log
Bug Report
Crawling
ErrorArch_r1 ErrorArch_r2,… ErrorArch_rn
ErrorSpace_r1 ErrorSpace_r2,.. ErrorSpace_rn
SoftArch_r1 SoftArch_r2,….. SoftArch_rn
Indexing
ArchDebtCandidates
HBFileSetSeq
ASFileSetSeq
ADFileSetSeq
MVFileSetSeq
HCP_r1 HCP_r2 … HCP_rn
HCPGen
Modeling
ArchDebts
ArchDebt_1 <FileSetSeq, DebtModel>
ArchDebt_2 <FileSetSeq, DebtModel>
……
ArchDebt_n <FileSetSeq, DebtModel>
Ranking
RankedArchDebs
ArchDebt_1 <FileSetSeq, DebtModel>
ArchDebt_2 <FileSetSeq, DebtModel>
……
ArchDebt_n <FileSetSeq, DebtModel>
Input file
Input files
New approach
Prior approach
Output files
Und
Commercial Tool
Input
Output

Figure 10.1: Approach Framework

10.2.1 Crawling: Selecting High-maintenance ***DRSpaces***

We first define the set of bug-prone files in a particular release r as a bug space: $BugSpace_r$=$\{f_1, f_2, ..., f_n\}$, where file f_i, $i = 1...n$, was revised to fix bugs at least once from release 1 to release r. According to this definition: $BugSpace_r$ is a subset of $BugSpace_{r+1}$. For each release r, we select a set of $DRSpaces$ from $SoftArch_r$, each led by a file in $BugSpace_r$, and form a $SelectedDRSpace$ set as the output of $Crawling$:

$$SelectedDRSpace_r = Crawling(SoftArch_r, BugSpace_r) \tag{10.4}$$

Each DRSpace in $SelectedDRSpace_r$ is led by a bug-prone file in $BugSpace_r$, and contains other files that depend on the leading bug-prone file. If there are n files in $BugSpace_r$, there are n DRSpaces in $SelectedDRSpace_r$.

10.2.2 Indexing: Identify ***ArchDebt*** Candidates

Next we need to find the $FileSetSequences$ that are debt candidates. Files in such a sequence must have changed together in the revision history. We first calculate a history coupling model—HCP matrix—and then we filter file groups using 4 architecture flaw patterns, which we call *indexing patterns.*

HCP Matrix In the prior chapters, we used a DSM to model *history coupling*: each cell in the DSM displays the number of times two files changed together. To manifest how a change to a file influences other files, we propose an extended history model: the **history coupling probability (HCP) matrix**. Although each column and row in a HCP still represents a file, we use a cell in the matrix to record the *conditional probability* of changing the file on the column, if the file on the row has been changed, indicating the odds of change propagation from one file to another.

Figure 10.2 shows a small example to illustrate the generation of a HCP. Part 1 of Figure 10.2 shows 4 files A, B, C, and D, that change in 4 commits: Commit1{A,B} (Commit1 changes A and B), Commit2{A,B}, Commit3{B,D}, and Commit4{A,C}. First, we compute the pair-wise change conditional probabilities between any pair of files. For example, the probability of changing file A, under the condition that file C has changed, denoted by $Prob\{A|C\}$, is the number of times A and C change in the same commits divided by the total number of changes to C. Similarly, $Prob\{C|A\}$ is the number of times A and C change in the same commits divided by the total number of changes to A. Hence, $Prob\{A|C\}$ is 1/1, indicating that A *always* changes with C, and $Prob\{C|A\}$ is 1/3, indicating a probability of 1/3 that C changes with A. In this relation, we recognize C as *dominant* and A as *submissive* because $Prob\{A|C\} > Prob\{C|A\}$. We compute the probabilities between every pair of files and get the graph in part 2 of Figure 10.2.

Next, as shown in part 3 of Figure 10.2, we compute the N-Transitive-Closure of the graph in part 2 to identify history dependencies between files that change in distinct but potentially related commits. The conditional probabilities between files without direct history connections are the multiplication of the probabilities on the transitive links. For example, file B and C never change in the same commits, but they change with file A in Commit1 and Commit4. Hence, there are transitive history connections between B and C. $Prob\{B|C\}$ is $Prob\{B|A\}$*$Prob\{A|C\}$=0.7*0.2=0.21, and $Prob\{C|B\}$ is $Prob\{C|A\}$*$Prob\{A|B\}$=1*0.7=0.7. We only keep links with probabilities of at least 0.3 to avoid keeping weak connections (the selection of 0.3 is still experimentally, we will discuss it as a threat to validity later). In case there are multiple paths between two files, which may suggest different conditional probabilities between two files. We keep the highest probability. Part 4 shows the N-Transitive-Closure which is stored in an adjacency matrix, called a HCP matrix.

For each release r of a project, we compute a HPC matrix (HPC_r), consisting of files in $BugSpace_r$, from the bug-fixing revision history between release 1 to release r.

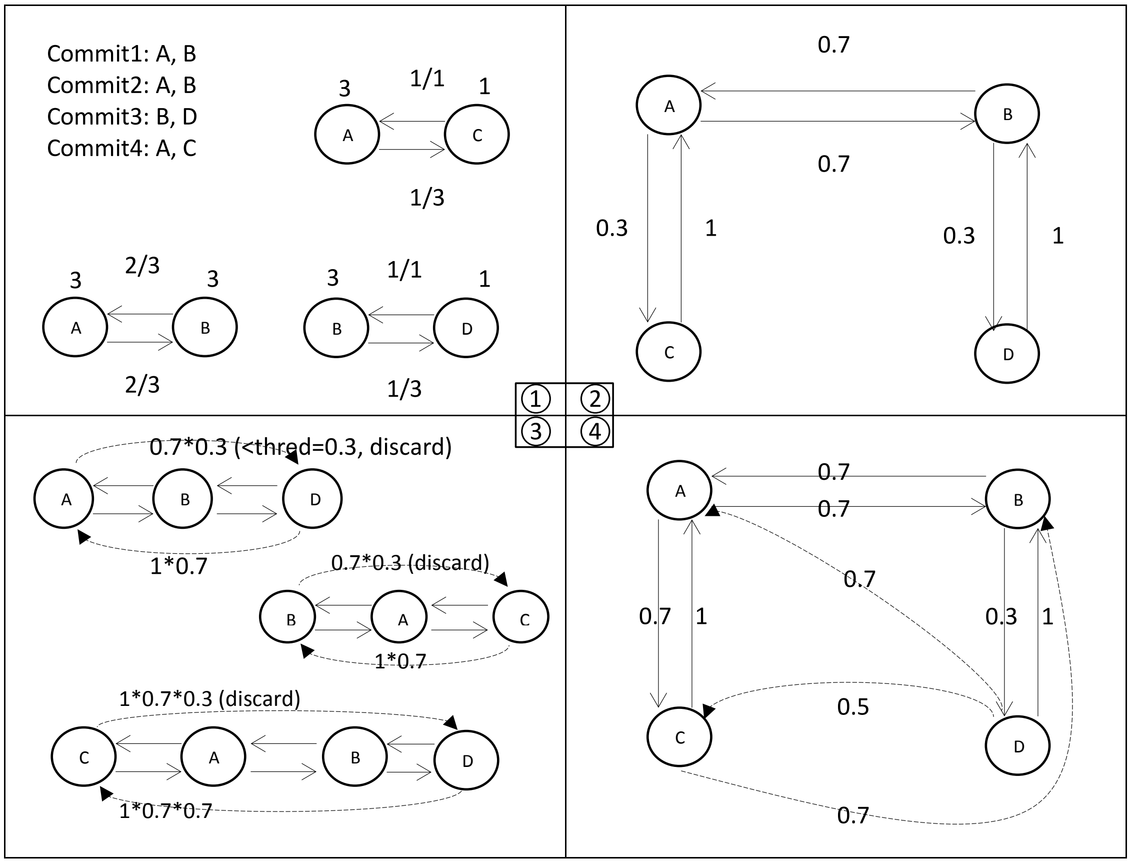


Figure 10.2: Generate HPC Matrix

Indexing Patterns Now we compute the interaction between $SelectedDRSpaces_r$ and HCP_r to find $FileSet_r$ from each release. We observe that, in most cases, even though the number of files in a $FileSet$ may vary in different releases, they are always connected to at least one file over all releases. For example, if more child classes are defined to extend a parent class over time, the group of files connected to the parent class grows. We thus call this one special file the *Anchor file* of the group, denoted as file a. We thus define $FileSet_r$ as:

$$\begin{aligned} &FileSet_r = \{a, M_r | M_r = \{m_i : i \text{ from } 1 \text{ to } n\}| \\ &\forall m_i \in M_r, m_i \text{ architecturally connected with } a \text{ in release } r\} \end{aligned} \tag{10.5}$$

where $FileSet_r \in FileSetSequence$, a is the anchor file, and the files contained in M_r may change with release r. We call M_r the member files of a in release r.

We also define two boolean expressions to describe the relationships between two files (x and y) in release r: $S_r(x \rightarrow y)$ and $H_r(x \rightarrow y)$. $S_r(x \rightarrow y)$ means y structurally depends on x in release r. $H_r(x \rightarrow y)$ means x is more likely to propagate changes to y in revision history than the opposite direction. We also say that x is dominant and y is submissive in their co-changes between release 1 to release r. In HCP_r, $HCP_r[x, y]$ is the probability of changing y, given x has changed. If $HCP_r[x, y] > HCP_r[y, x]$, then x is dominant and y is submissive. $HCP_r[x, y] = HCP_r[y, x]$ means x and y are equally dominant. Formally:

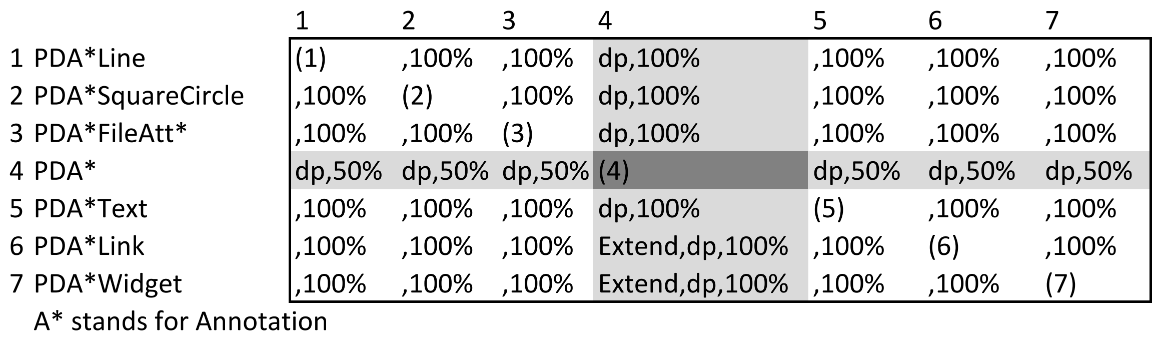
$$\begin{aligned} &\text{In release } r, \\ &S_r(x \rightarrow y) \text{ is true if } y \in DRSpace_{r\_}x\text{, otherwise it is false} \\ &H_r(x \rightarrow y) \text{ is true if } HCP_r[x, y] >= HCP_r[y, x] \\ &\wedge HCP_r[x, y] \neq 0\text{, otherwise it is false} \end{aligned} \tag{10.6}$$

For any pair of a and m in a $FileSet_r$, we identify 4 relationships: $S_r(a \rightarrow m)$, $S_r(m \rightarrow a)$, $H_r(a \rightarrow m)$, and $H_r(m \rightarrow a)$. Each relationship could be either true or false. We enumerated all 16 combinations of these 4 relationships. The 4 combinations with $H_r(a \rightarrow m)$ and $H_r(a \rightarrow m)$ $false$ are irrelevant to our analysis (as we need history to measure debt). From the remaining 12 possible combinations, we defined 4 indexing patterns—*Hub, Anchor Submissive, Anchor Dominant, Modularity Violation*. Each pattern corresponds to prototypical architectural issues that proved to correlate with reduced software quality Mo et al. [2015].

Given any anchor file $a \in BugSpace_r$, we could calculate its $FileSet_{r\_a}$ using $SelectedDRSpace_r$ and HCP_r through the lens of the 4 indexing patterns:

Hub—the anchor file and each member have structural dependencies in both directions and history dominance in at least one direction. The anchor is an architectural hub for its members. This pattern corresponds to cyclic dependency, unhealthy inheritance (if the anchor file is a super-class or interface class), and unstable interface (if the anchor file has many dependents). Informally such structures are referred to as "spaghetti code", or "big ball of mud". A $FileSet_{r\_a}$ with anchor file a in release r that matches a *hub* pattern is denoted by $HBFileSet_{r\_a}$ and is calculated as:

$$\begin{aligned}
&HBFileSet_{r\_a} = Index_{HB}(a, SelectedDRSpace_r, HCP_r)\\
&= \{a, M_r | \forall m \in M_r, S_r(a \to m) \land S_r(m \to a)\\
&\land (H_r(a \to m) \lor H_r(m \to a))\}
\end{aligned} \tag{10.7}$$



	1	2	3	4	5	6	7
1 PDA*Line	(1)	,100%	,100%	dp,100%	,100%	,100%	,100%
2 PDA*SquareCircle	,100%	(2)	,100%	dp,100%	,100%	,100%	,100%
3 PDA*FileAtt*	,100%	,100%	(3)	dp,100%	,100%	,100%	,100%
4 PDA*	dp,50%	dp,50%	dp,50%	(4)	dp,50%	dp,50%	dp,50%
5 PDA*Text	,100%	,100%	,100%	dp,100%	(5)	,100%	,100%
6 PDA*Link	,100%	,100%	,100%	Extend,dp,100%	,100%	(6)	,100%
7 PDA*Widget	,100%	,100%	,100%	Extend,dp,100%	,100%	,100%	(7)

A* stands for Annotation

Figure 10.3: Hub

Figure 10.3 is a Hub *FileSet* for the PDFBox project, anchored by *PDAnnotation*. The dark grey cell represents the anchor file (cell[4,4] for *PDAnnotation*). The cells showing the history and structure relationships between member files and the anchor file are in lighter grey. In this *HBFileSet*, the anchor file structurally depends on each member file, and each member file also structurally depends on the anchor file. When the anchor file changes, each member file has a 50% probability of changing as well. When a member file changes, the anchor file always changes with it. A *HBFileSet* is potentially problematic because the anchor file, like a hub, is strongly coupled with every member file both structurally and historically.

Anchor Submissive—each member file structurally depends on the anchor file, but each member historically dominates the anchor. This pattern corresponds to an unstable interface, where the interface is submissive in changes. An *Anchor Submissive FileSet* with anchor a in release rt is:

$$\begin{aligned} &ASFileSet_{r\_a} = Index_{AS}(a, SelectedDRSpace_r, HCP_r) \\ &= \{a, M_r \mid \forall m \in M_r, S_r(a \rightarrow m) \wedge \qquad (10.8) \\ &\neg\, S_r(m \rightarrow a) \wedge H_r(m \rightarrow a) \end{aligned}$$

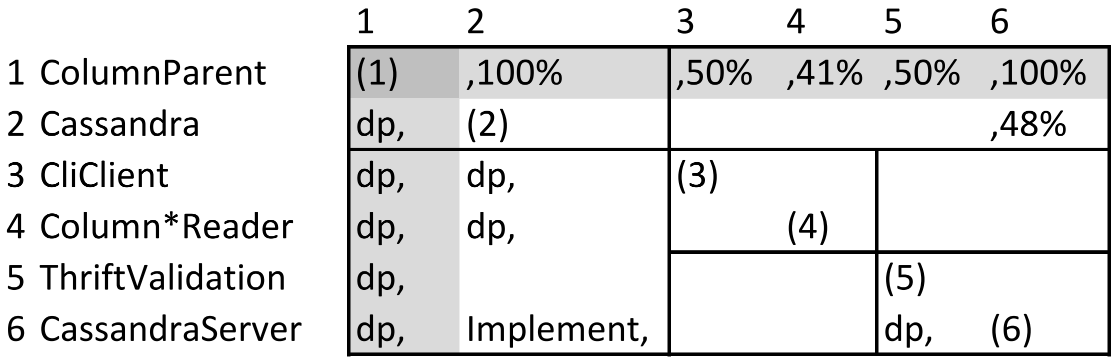
	1	2	3	4	5	6	7	8
1 AbstractType	(1)							
2 UUIDSerializer	,100%	(2)	,50%			,100%		,50%
3 UUIDType	ext,dp,33%	dp,	(3)				,33%	,50%
4 AbstractCell	dp,50%			(4)				
5 TypeCast	dp,33%		,33%		(5)		,33%	,33%
6 IntegerSerializer	,100%	,100%	,50%			(6)		,50%
7 LongType	ext,dp,67%		,67%		,33%		(7)	dp,67%
8 DateType	ext,dp,40%		,60%				dp,40%	(8)

Figure 10.4: Anchor Submissive

Figure 10.4 shows an *ASFileSet* with anchor *AbstractType* in the Cassandra project. Each member file structurally, directly or indirectly, depends on the anchor file, but when the member files change, the anchor file changes with each of them, with historical probabilities of 33% to 100%. A *ASFileSet* is problematic because the history dominance is in the opposite direction to the structural influences: the anchor file should influence the member files, not the other way around.

Anchor Dominant—each member file structurally depends on the anchor file and the anchor file historically dominates each member file. This pattern corresponds to the other type of unstable interface, where the interface is dominant in changes. An *Anchor Dominant FileSet* with anchor a in release rt can be calculated as:

$$\begin{aligned} & ADFileSet_{r\_a} = Index_{AD}(a, SelectedDRSpace_r, HCP_r) \\ & = \{a, M_r | \forall m \in M_r, S_r(a \rightarrow m) \wedge \\ & \neg S_r(m \rightarrow a) \wedge H_r(a \rightarrow m)\} \end{aligned} \tag{10.9}$$



	1	2	3	4	5	6
1 ColumnParent	(1)	,100%	,50%	,41%	,50%	,100%
2 Cassandra	dp,	(2)				,48%
3 CliClient	dp,	dp,	(3)			
4 Column*Reader	dp,	dp,		(4)		
5 ThriftValidation	dp,				(5)	
6 CassandraServer	dp,	Implement,			dp,	(6)

Figure 10.5: Anchor Dominant

Figure 10.5 shows an *ADFileSet* calculated using anchor *ColumnParent* in Cassandra. Each member file (from row 2 to row 6) structurally depends on (cell[2 to 6:1]) the anchor file (row 1), and when the anchor file changes, the member files change as well with probabilities from 41% to 100% (cell[1:2 to 6]). A $ADFileSet$ presents potential problems where the anchor file is unstable and propagates changes to member files that structurally depend on it.

Modularity Violation—there are no structure dependencies between the anchor and any member, however, they historically couple with each other. In a *modularity violation*, the anchor file and the member files share some common assumptions ("shared secrets"), but these are not represented in any structural connection. A $MVFileSet$ with anchor a in release r can be calculated as:

$$\begin{aligned} &MVFileSet_{r\_a} = Index_{MV}(a, SelectedDRSpace_r, HCP_r) \\ &= \{a, M_r | \forall m \in M_r, \neg S_r(a \to m) \wedge \neg S_r(m \to a) \\ &\wedge (H_r(m \to a) \vee H_r(a \to m))\} \end{aligned} \tag{10.10}$$

	1	2	3	4	5	6	7	8
1 JMXETPEMBean	(1)	,100%	,44%	,50%		,100%	,100%	,50%
2 DebuggableTPExecutor		(2)	,31%					
3 StorageService			(3)	dp,	dp,Use,			
4 ColumnFamilyStore			dp,	(4)				
5 MessagingService			dp,		(5)		dp,	
6 NodeProbe			,44%		dp,	(6)		
7 StatusLogger		,50%		dp,50%	dp,	,50%	(7)	
8 JMXCTPExecutor	,50%	,100%	,31%	,100%	,50%	,50%	,50%	(8)

Figure 10.6: Modularity Violation

Figure 10.6 is a $MVFileSet$ with anchor $JMXCTPExecutor$ (row 12) in Cassandra. The anchor file, on the bottom of the matrix, is structurally isolated from the member files. However, when the anchor file changes, there are historically 31% to 100% probabilities that the member files change as well, and when the member file *JMXETPEMBean* (on row 1) changes, the anchor file has a 50% chance to change with it. This pattern identifies potential problems where the anchor file and the member files share common assumptions, without explicit structural connections, and these assumptions are manifested by historical co-change relationships.

Identify *ArchDebtCandidates* by anchor file For each release r, we use each a in $BugSpace_r$ as the anchor file to calculate a $FileSet$ for each of the 4 patterns: $HBFileSet_{r\_a}$, $ASFileSet_{r\_a}$, $ADFileSet_{r\_a}$, and $MVFileSet_{r\_a}$. The *FileSetSequence* in the *Hub* pattern with anchor file a is denote by *HBFileSetSequence*$_a$. Similarly, for anchor a, we can identify *AS*-, *AD*-, and *MV*- *FileSetSequence*$_a$. Using any bug-prone file as the anchor, we can identify 4 *FileSetSequence*, each of which is an *ArchDebtCandidate*.

As a result, for each $a \in BugSpace_r$ and for each release r , we can exhaustively detect 4*$| \cup_{r=1}^{n} BugSpace_r|$ candidates, which equals 4*$|BugSpace_n|$ because $BugSpace_n$ is a super set of all $BugSpace$ in earlier releases.

10.2.3 Modeling: Build Regression Model

Now that we have identified the $FileSetSequences$, candidates of Archdebt, we further: (1) measure maintenance costs incurred by each $FileSet$ within a *FileSetSequence*, and (2) formulate a *DebtModel* to capture cost variation.

Measure *ArchDebtCandidates* From each *FileSetSequence*, we first exclude each $FileSet_r$ that only contains 1 file (the anchor file) since it doesn't involve architecture problems. After that, we define the **age** of a *FileSetSequence* as the number of $FileSets$ in it after unqualified $FileSets$ are filtered out.

Then, for each $FileSet_r$, we measure the maintenance effort, denoted by *Effort_FileSetr*$_r$, it consumes by the end of release r. For any file $f \in$ *FileSet*$_r$, we approximate its maintenance costs as the amount of bug-fixing churn on it by the end of release r. We denote the maintenance cost for file f by release r as $BugChurn_{r\_f}$. *Effort_FileSetr*$_r$ is the sum of maintenance costs spent on each file in the set:

$$\mathit{Effort\_FileSet}_r = \sum\nolimits_{\forall f \in FileSet_r} BugChurn_{r\_f} \tag{10.11}$$

To qualify as a real debt, first a *FileSetSequence* should have long-lasting impacts. This can be evaluated using the age of *FileSetSequence*. Second, *FileSetSequence* should consume increasing amount of maintenance effort. Suppose a software system has n releases. Let $FileSet_f$ and $FileSet_l$ be the first and last element in *FileSetSequence*. A *FileSetSequence* is identified as a real debt if it satisfies the following conditions:

$$\begin{cases} age >= n/c; \\ \mathit{Effort\_FileSet}_l > \mathit{Effort\_FileSet}_f. \end{cases}$$

where c is a tunable parameter. In this dissertation, c=2, meaning that *FileSetSequence* influences at least half of the releases. Otherwise, the candidate is not a meaningful debt, at least not yet. The second condition requires that the maintenance costs on *FileSetSequence* increase over time (when an anchor file architecturally connects to smaller numbers of member files over time, due to reasons such as refactoring, a candidate may exhibit reducing maintenance costs over time, and thus cannot be a debt).

Formulate *DebtModel* For each *FileSetSequence* identified as a real debt, we select a suitable regression model as its $DebtModel$ to describe the growing trend (the interest rate) of maintenance costs over time. We use four types of regression models:

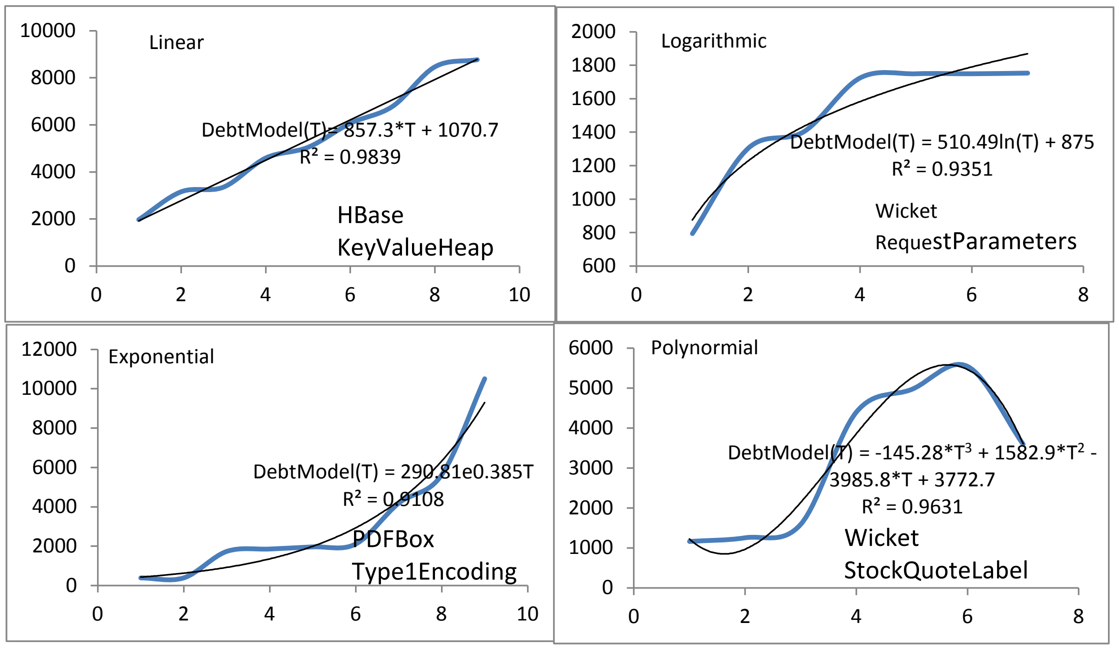


Figure 10.7: 4 Types of Regression Model

linear, logarithmic, exponential, and polynomial (up to degree 10). Figure 10.7 shows typical examples of these 4 models. Each model represents a coherent scenario. In a linear model (part 1 of Figure 10.7), the penalties of a debt increase at a stable rate in each version. In a logarithmic model (part 2), the penalties of a debt increase more slowly over time (for example, when developers refactor a group of files, it become easier to make the next change to them, so the interest rate on the debt drops over time). In an exponential model (part 3), the penalties of a debt increase at ever-faster rates over time (for example, the structure of a tangled group of files gets exponentially worse, often in the early stages of a project, before anyone worries about modularity). In a polynomial model (part 4), the penalties of a debt increase with many fluctuations over the set of releases.

We calculate the maintenance costs—$Effort\_FileSetr_r$ for each $FileSet_r$ in a *FileSetSequence* using equation 10.11. The $Effort\_FileSetr_r$ of all $FileSet_r$ in a *File-*

SetSequence form an array that we call $Effort\_Array$. $Effort\_Array[i]$ = *Effort_FileSetr*$_r$, where *FileSet*$_r$ is the ith element of *FileSetSequence*. We define an integer array $T[i] = r$, where r is the release number of the ith element in *FileSetSequence*. Each release r is numbered by its order in the release in history. In the $DebtModel$ of a *FileSetSequence*, $Effort\_Array$ is the independent value and T is the dependent value. "ModelSelector" (shown in Algorithm 3) selects a regression model for the relationship between T and $Effort\_Array$. The formula and R^2 of the regression model are returned as $DebtModel$:

$$DebtModel = ModelSelector(EffortArray, T) \tag{10.12}$$

We define a global parameter R^2_{thresh} (the R^2 threshold) for *ModelSelector*. R^2_{thresh} ranges from 0 to 1; the higher the value, the stricter $Effort\_Array$ and T fit the selected model. Our *ModelSelector* algorithm first tries to fit the $Effort\_Array$ and T into a linear regression model. If the R^2_{Lin} of the linear model reaches the threshold R^2_{thresh}, it returns the linear model. If not, it builds both logarithmic model and exponential model, and computes their R^2 values. If the R^2 values of both models reache R^2_{thresh}, it returns the model that gives a higher R^2. Otherwise, it returns the model that reaches the threshold. If the debt fits neither of them with $R^2 >= R^2_{thresh}$, it tries polynomial models of degrees up to 10. A polynomial model where $R^2_{poly} >= R^2_{thresh}$ or the degree reaches 10, whichever is satisfied first, is returned.

In the *ModelSelector* algorithm, we give higher priority to linear, logarithmic, and exponential models over polynomial models. We do not simply pick the best fit (i.e., the model with highest R^2). The reason is that the linear, logarithmic, and exponential models present three general types of penalty interest rate: stable, decreasing, and increasing. The polynomial model, however, catches minor fluc-

ALGORITHM 3: ModelSelector ($EffortArray$,T)

```
1: $model_{Lin} \leftarrow LinearFit$(EffortArray,T)
2: $R^2_{Lin} \leftarrow model_{Lin}.getR^2()$
3: if $R^2_{Lin} >= R^2_{thresh}$ then
4:     return $model_{Lin}$
5: end if
6: $model_{Log} \leftarrow LogFit$(EffortArray,T)
7: $R^2_{Log} \leftarrow model_{Log}.getR^2()$
8: $model_{Exp} \leftarrow ExpFit$(EffortArray,T)
9: $R^2_{Exp} \leftarrow model_{Exp}.getR^2()$
10: if $R^2_{Log} >= R^2_{thresh}$ and $R^2_{Exp} >= R^2_{thresh}$ then
11:     if $R^2_{Log} > R^2_{Exp}$ then
12:
13:         return $model_{Log}$
14:     end if
15:
16:     return $model_{Exp}$
17: end if
18: if $R^2_{Log} >= R^2_{thresh}$ then
19:
20:     return $model_{Log}$
21: end if
22: if $R^2_{Exp} >= R^2_{thresh}$ then
23:
24:     return $model_{Exp}$
25: end if
26: $model_{poly} \leftarrow PolyFit$(EffortArray,T, 10)
27:
28: return $model_{poly}$
```

tuations of the penalty trend, most likely a result of noise due to extraneous factors. For example, the debt in part 1 of Figure 10.7, intuitively a linear model ($DebtModel(r) = 857 * r + 1070$ with R^2 of 0.98), can fit into a polynomial model $DebtModel(r) = -2*r^6+59*r^5-680*r^4+3874*r^3-11342*r^2+16538*r-6466$, with a higher R^2 (0.99). The polynomial model fits better (higher R^2), but the linear model is preferred. As long as a debt penalty generally ($R^2 >= R^2_{thresh}$, where e.g. R^2_{thresh} is 0.8) fits into a linear, logarithmic or exponential model, we choose those models.

For each *FileSetSequence*, we identify its *DebtModel*. This completes our *ArchDebt* identification.

10.2.4 Ranking: Identify High-maintenance ***ArchDebt***

Not all architectural debts have the same severity in terms of the maintenance costs they incur. Debts with higher maintenance consequences deserve more attention. We rank all the identified architectural debts according to their accumulative maintenance cost as follows.

We define a pair $p_f < f, BugChurn_f >$, where f is a bug-prone file, $BugChurn_f$ is the maintenance costs for f, approximated by bug-fixing churn on f. Let *EffortMap* be the set of p_f, such that $\forall f \in BugSpace_n$ (n is the latest release), there exists a $p_f \in$ *EffortMap*. *EffortMap* is one of the inputs to the *ranking* algorithm. The other input is the identified *ArchDebts*.

$$RankedDebts = ranking(ArchDebts, EffortMap) \tag{10.13}$$

In the ranking algorithm 4, we rank the importance of each *ArchDebt* according to *EffortMap* in a loop. In each iteration, we select *maxArchDebt* that consumes the largest portion of effort for files in *EffortMap* from *ArchDebts*. The effort for duplicate files are excluded, and the iteration terminates when all *ArchDebts* are ranked. The top debts returned consume the largest possible maintenance effort, and deserve more attentiosn and higher priority.

ALGORITHM 4: ranking ($ArchDebts$,$EffortMap$)

```
1: $RankedDebts \leftarrow \emptyset$
2: while $ArchDebts$ is not $\emptyset$ do
3:     $maxDebt$ = MaxDebt($EffortsMap$,$ArchDebts$)
4:     $RankedDebts$.addtoTail($maxDebt$)
5:     $ArchDebts$.remove($maxDebt$)
6:     $EffortsMap$.removeAllFiles($maxDebt.FileSetSequence$)
7: end while
8: return $RankedDebts$
```

10.3 Evaluation

To evaluate the effectiveness of our approach, we investigate the following research question:

RQ: Whether the file groups identified in ArchDebts generate and grow significant amount of maintenance costs? That is, are they true and significant *debts?*

If the identified file groups only consume a small portion of overall maintenance effort, then they do not deserve much attention. Similarly, if the identified file groups cover a large portion of the system itself, it is not surprising if they also consume the majority of maintenance effort. In both cases, we cannot claim that they are debts worthy of attention.

10.3.1 Subjects

We chose 7 Apache open source projects as our evaluation subjects. These projects differ in scale, application domain, length of history, and many other project characteristics. They are: Camel—a integration framework based on Enterprise Integration Patterns; Cassandra—a distributed DBMS; CXF—a Web services framework; Hadoop—a framework for reliable, scalable, distributed computing; HBase—the Hadoop distributed, scalable, big data store; PDFBox—a library for working with PDF documents; and Wicket—a component-based web application framework. A summary of these projects is given in Table 10.1. The second column is the start to end time and the total number of months (in parentheses) for each project. The third column "#R" shows the number of releases selected per project. We selected releases to ensure that the time interval between two releases is approximately 6 months. The column "#Cmt" is the number of commits made over the selected history. The column "#Iss" is the number of bug reports, extracted from the project's bug-tracking system. The last column shows the size range, measured as the number of files in the first and the last selected release.

Table 10.1: Subject Projects

Subject	Length of history (#Mon)	#R	#Cmt	#Iss	#Files
Camel	7/2008 to 7/2014 (72)	12	14563	2790	1838 to 9866
Cassandra	9/2009 to 11/2014 (62)	10	14673	4731	311 to 1337
CXF	12/2007 to 5/2014 (77)	13	8937	3854	2861 to 5509
Hadoop	8/2009 to 8/2014 (60)	9	8253	5443	1307 to 5488
HBase	12/2009 to 5/2014 (53)	9	6718	6280	560 to 2055
PDFBox	8/2009 to 9/2014 (62)	12	2005	1857	447 to 791
Wicket	6/2007 to 1/2015 (92)	15	8309	3557	1879 to 3081

10.3.2 Evaluation Results

To answer our research question, we measured the amount of maintenance effort spent on the ArchDebts we identified. Since we can not directly measure the amount of effort in working hours or budgets, we use bug-fixing churn as an approximation: the number of lines of code modified and committed to fix bugs.

We use HBase as an example to illustrate our observations. Figure 10.8 shows the percentage of maintenance effort associated with the files in *FileSets* of all iden-

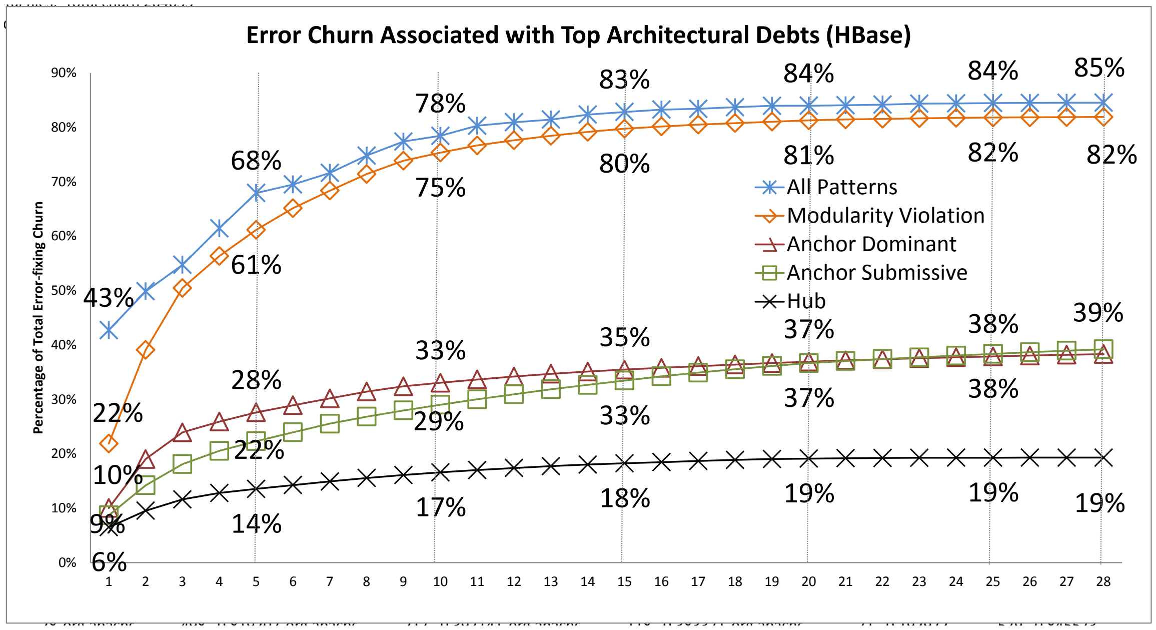


Figure 10.8: Debt Churn Consumption (HBase)

tified *ArchDebts* in HBase. The x-axis is the number (from 1 to 28) of identified architectural debts. The y-axis is the accumulated percentage of maintenance effort associated with the top x *ArchDebts*. Each line represents the percentage of each type of debt. This figure depicts, from bottom to top, you can see: *Hub*, *Anchor-Submissive*, *Anchor-Dominant*, and *Modularity Violation* debts respectively. The line on the top is the total percentage of the 4 types of debts. The values of the top line are not simply the sum of the values of the 4 types because different types of debts may share some files. Thus we make the following observations in HBase.

(1) Architectural debts consume a significant percentage (85%) of the total project maintenance effort. A significant portion of the maintenance effort is spent on paying interest on related groups of files. If they can identify such debts early, a project can save significant effort by paying down the debts via refactoring Kazman et al. [2015]. As the number of debts increases, the total does not reach 100% because not all bugs are architecturally connected. Occasionally, developers introduce bugs that can be fixed in isolation.

(2) The top few architectural debts consume a large percentage of maintenance effort. The top **5** *Modularity Violation* debts in HBase consume **61%** of total effort, wherease **all** *Modularity Violation* debts consume **82%** of total effort. Similar observations hold for *Anchor-Submissive*, *Anchor-Dominant*, and *Hub* debts. The lines flatten as the number of debts increases, indicating that most of the effort concentrates in the top few debts. This means that instead of reviewing all identified debts, project leaders only need to focus on the top few.

(3) *Modularity Violation* debt is the most common and expensive debt. *Hub* debts consume the least percentage of effort, while *Anchor-Dominant* and *Anchor-Submissive* take similar percentages. We can see that the line for *Modularity Violation* is close to the line for the sum of all types. This is because *Modularity Violation* debts involve the files in other debts as well.

We made consistent observations from all 7 projects, as summarized in Table 10.2. Column "All Debts Ch%" shows that, for all 7 projects, from 51% to 85% of the total maintenance effort is consumed by architectural debts. And, a large percentage (31% to 50%) of the effort is consumed by the top 5 *Modularity Violation* debts (shown in sub column "Ch%" under "Modularity Vio"). *Modularity Violation* debts impact the largest number of files and consume the greatest effort, *Hub* debts consume the least, while *Anchor-Submissive* and *Anchor-Dominant* rotate their orders.

If a debt contains a large number of files, it is not surprising that they take a large percentage of effort. We observed, however, that **(4) the top 5 architectural debts contain only a small number of files, but consume a large amount of the total project effort.** We compare the number of files in the top 5 architectural debts versus the percentage of effort they take. For example, in table 10.2, column "Modularity Vio" under "Top 5 Debts" shows that, in Camel, there are 206 files (13%

of all the bug-prone files) in the top 5 *ModularityViolation* debts, and these 206 files consume 32% of the total project bug-fixing effort. Similarly, in Camel, the top 5 *Anchor Submissive*, *Anchor Dominant*, and *Hub* debts contain only 1%, 4%, and 2% of the bug-prone files, but consume 7%, 16%, and 5% of the total effort respectively. From the column "All 4 types" under "Top 5 Debts", we can observe that, for all the projects, the top 5 architectural debts contain from only 11% to 32% of the bug-prone files, but consume 27% to 49% of the total effort. The average ratio of percentage of effort to the percentage of files in the top 5 debts is 2.

Finally, we analyze the file size (in lines of code) of the debts we identified. Much research has shown that file size correlates with bug rates and churn. We would like to know that the debts identified by our approach are not just a set of large files. To show this we counted the LOC of the files in the top 5 debts, and observed that the sizes of these files are randomly distributed. Figure 10.9, for example, shows the file size distribution of the top 5 *Modularity Violation* debts in Cassandra. The x-axis is the range of file size: 10% means the top 10% largest files, 10-20% means files in the 10-20% range in LOC, and so forth. The y-axis is the percentage of files in the top 5 debts that belong to each size range. For example, 22% of the files in top 5 debts are in the top 10% largest files, and 11% of the files are in the range of 90-100% range (that is, the smallest files). The top 5 debts do contain a non-trivial number of large files (22% from the top 10% size range), consistent with other studies showing that large files tend to be problematic. But Figure 10.9 shows that the top 5 debts contain files in *all* size ranges.

In summary, we can claim that the architectural debts identified by our approach are truly debts that account for a large amount (from 51% to 85%) of maintenance effort. Most (31% to 61%) of the maintenance effort concentrates in the top 5 architectural debts, which contain only a small percentage (13% to 25%) of the project's

Table 10.2: Top 5 Debt:#Files vs Churn

Projects	All Debts	Top 5 Debts									
	Ch%	All 4 types		Modularity Vio		Anchor Sub.		Anchor Dom.		Hub	
		Fls	Ch%	Fls	Ch%	Fls	Ch%	Fls	Ch%	Fls	Ch%
Camel	59%	230(15%)	35%	206(13%)	32%	20(1%)	7%	60(4%)	16%	40(2%)	5%
Cassandra	72%	273(28%)	57%	196(20%)	50%	72(7%)	28%	33(3%)	32%	26(3%)	16%
CXF	56%	200(11%)	27%	136(8%)	20%	70(4%)	6%	22(1%)	10%	12(1%)	3%
Hadoop	51%	145(25%)	44%	118(20%)	42%	45(8%)	22%	10(2%)	16%	10(2%)	6%
HBase	85%	349(30%)	67%	290(25%)	61%	87(7%)	15%	36(3%)	27%	23(2%)	13%
PDFBox	67%	133(32%)	49%	107(25%)	45%	35(8%)	12%	30(7%)	26%	17(4%)	10%
Wicket	62%	295(22%)	38%	214(16%)	31%	130(10%)	11%	35(3%)	13%	14(1%)	7%

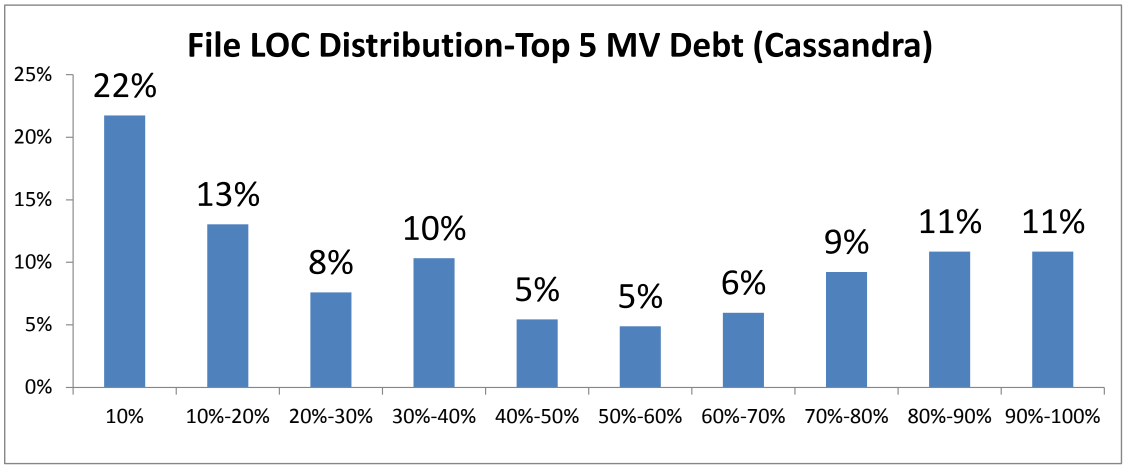


Figure 10.9: Top 5 Debts File Size Distribution (Cassandra)

files.

10.4 Discussion

We now discuss which model best describes the *interest rate* of an *ArchDebt* and illustrate how our approach helps to understand and monitor the evolution of ArchDebts.

10.4.1 The Interest Rate of ArchDebt

For each *ArchDebt*, we search for a suitable regression model to capture its interest rate, as introduced in 10.2.3, using R^2_{thresh} of 0.75 and 0.8 respectively. The results are reported in Table 10.3. The first column is project name. The second column is the number of instances of *ArchDebt* identified in a project. The third and forth columns are model distributions for R^2_{thresh} of 0.75 and 0.8 respectively.

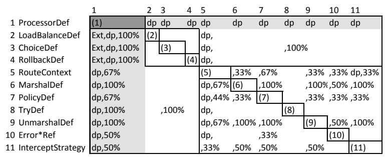
When R^2_{thresh}=0.75, in all the projects, about half (46% to 65%) of the debts fit a linear regression model (with $R^2 >= 0.75$). For other debts where a linear model doesn't fit, a small percentage fits either a logarithmic (4% to 22%) or exponential (0% to 7%) model (with $R^2 >= 0.75$), and a polynomial model fits 25% to 41% of the identified debts.

When R^2_{thresh}=0.8, the models are less noise-tolerant. We can see that linear model is still common (36% to 62%) for all projects. But a small portion of debts, from 6% (HBase, 31% minus 25%) to 18% (PDFBox, 51% minus 33%), can no longer fit into linear, logarithmic, or exponential models, but fit a polynomial model.

In summary, when R^2_{thresh} is 0.75, the linear model is most common—about half of the debts fit into it. This indicates that half of *ArchDebts* accumulate maintenance interest at a constant rate. Only a small portion of debts accumulate interest at a faster (less than 7% in exponential) or slower (less than 22% in logarithmic) rate. About 1/3 of the identified debts accumulate costs with a more fluctuating rate, which is captured by a polynomial model. More *ArchDebts* fit into a polynomial model as R^2_{thresh} increases.

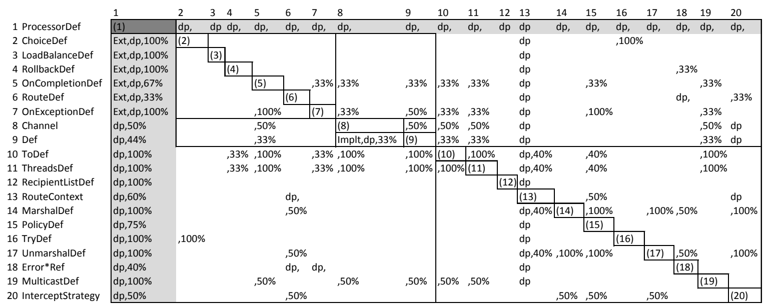
10.4.2 Architectural Debt Evolution

We showed, that the top 5 debts consume a large amount of effort. We manually inspected the evolution of these debts, and now illustrate how architectural flaws



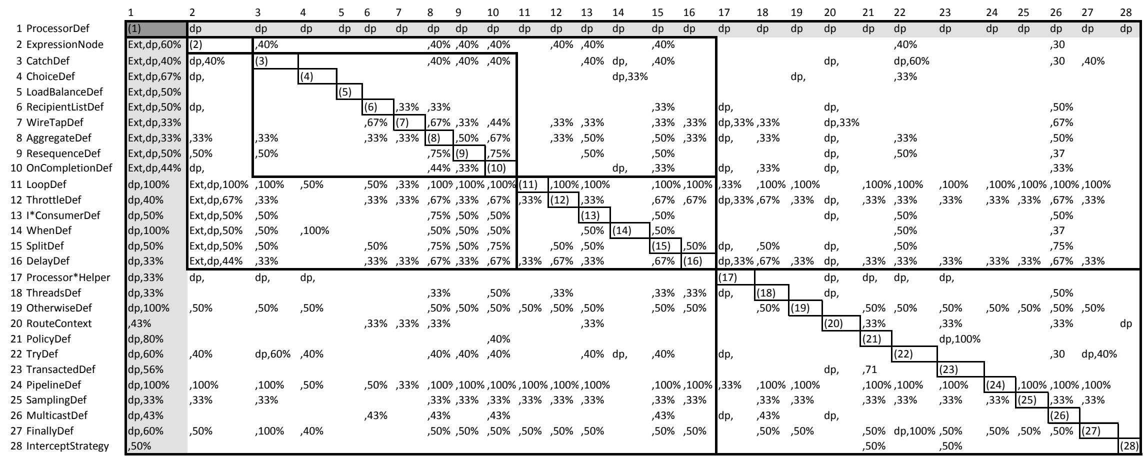
	1	2	3	4	5	6	7	8	9	10	11
1 ProcessorDef	(1)	dp	dp	dp	dp	dp	dp	dp	dp	dp	dp
2 LoadBalanceDef	Ext,dp,100%	(2)			dp,						
3 ChoiceDef	Ext,dp,100%		(3)		dp,			,100%			
4 RollbackDef	Ext,dp,100%			(4)	dp,						
5 RouteContext	dp,67%				(5)	,33%	,67%		,33%	,33%	dp,33%
6 MarshalDef	dp,100%				dp,67%	(6)	,100%		,100%	,50%	,100%
7 PolicyDef	dp,67%				dp,44%	,33%	(7)		,33%	,33%	,33%
8 TryDef	dp,100%		,100%		dp,			(8)			
9 UnmarshalDef	dp,100%				dp,67%	,100%	,100%		(9)	,50%	,100%
10 Error*Ref	dp,50%				dp,		,33%			(10)	
11 InterceptStrategy	dp,50%				,33%	,50%	,50%		,50%		(11)

(a) R-2.0.0, Age 1, #Files 11, Churn 392



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1 ProcessorDef	(1)	dp,	dp	dp,	dp,	dp,	dp,	dp,	dp,	dp,	dp,	dp	dp	dp,	dp,	dp,	dp,	dp,	dp,	dp,
2 ChoiceDef	Ext,dp,100%	(2)											dp			,100%				
3 LoadBalanceDef	Ext,dp,100%		(3)										dp							
4 RollbackDef	Ext,dp,100%			(4)									dp					,33%		
5 OnCompletionDef	Ext,dp,67%				(5)		,33%	,33%	,33%	,33%	,33%		dp		,33%				,33%	
6 RouteDef	Ext,dp,33%					(6)							dp					dp,		,33%
7 OnExceptionDef	Ext,dp,100%				,100%		(7)	,33%	,50%	,33%	,33%		dp		,100%				,33%	
8 Channel	dp,50%				,50%			(8)	,50%	,50%	,50%		dp						,50%	dp
9 Def	dp,44%				,33%			Implt,dp,33%	(9)	,33%	,33%		dp						,33%	dp
10 ToDef	dp,100%			,33%	,100%		,33%	,100%	,100%	(10)	,100%		dp,40%		,40%				,100%	
11 ThreadsDef	dp,100%			,33%	,100%		,33%	,100%	,100%	,100%	(11)		dp,40%		,40%				,100%	
12 RecipientListDef	dp,100%											(12)	dp							
13 RouteContext	dp,60%					dp,							(13)		,50%					dp
14 MarshalDef	dp,100%					,50%							dp,40%	(14)	,100%		,100%	,50%		,100%
15 PolicyDef	dp,75%												dp		(15)					
16 TryDef	dp,100%	,100%											dp			(16)				
17 UnmarshalDef	dp,100%					,50%							dp,40%	,100%	,100%		(17)	,50%		,100%
18 Error*Ref	dp,40%					dp,	dp,						dp					(18)		
19 MulticastDef	dp,100%				,50%			,50%	,50%	,50%	,50%		dp						(19)	
20 InterceptStrategy	dp,50%					,50%								,50%	,50%		,50%			(20)

(b) R-2.2.0, Age 2, #Files 20, Churn 771



	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 ProcessorDef	(1)	dp	dp	dp	dp	dp	dp	dp	dp	dp	dp	dp	dp	dp	dp	dp	dp	dp	dp	dp	dp	dp	dp	dp	dp	dp	dp	dp
2 ExpressionNode	Ext,dp,60%	(2)	,40%					,40%	,40%	,40%		,40%	,40%		,40%							,40%				,30		
3 CatchDef	Ext,dp,40%	dp,40%	(3)					,40%	,40%	,40%			,40%	dp,	,40%					dp,		dp,60%				,30	,40%	
4 ChoiceDef	Ext,dp,67%	dp,		(4)										dp,33%					dp,			,33%						
5 LoadBalanceDef	Ext,dp,50%				(5)																							
6 RecipientListDef	Ext,dp,50%	dp,				(6)	,33%	,33%							,33%		dp,			dp,						,50%		
7 WireTapDef	Ext,dp,33%					,67%	(7)	,67%	,33%	,44%		,33%	,33%		,33%	,33%	dp,33%	,33%		dp,33%						,67%		
8 AggregateDef	Ext,dp,33%	,33%	,33%			,33%	,33%	(8)	,50%	,67%		,33%	,50%		,50%	,33%	dp,	,33%		dp,		,33%				,50%		
9 ResequenceDef	Ext,dp,50%	,50%	,50%					,75%	(9)	,75%			,50%		,50%					dp,		,50%				,37		
10 OnCompletionDef	Ext,dp,44%	dp,						,44%	,33%	(10)				dp,	,33%		dp,	,33%		dp,						,33%		
11 LoopDef	dp,100%	Ext,dp,100%	,100%	,50%		,50%	,33%	,100%	,100%	,100%	(11)	,100%	,100%		,100%	,100%	,33%	,100%	,100%		,100%	,100%	,100%	,100%	,100%	,100%	,100%	
12 ThrottleDef	dp,40%	Ext,dp,67%	,33%			,33%	,33%	,67%	,33%	,67%	,33%	(12)	,33%		,67%	,67%	dp,33%	,67%	,33%	dp,	,33%	,33%	,33%	,33%	,33%	,67%	,33%	
13 I*ConsumerDef	dp,50%	Ext,dp,50%	,50%					,75%	,50%	,50%			(13)		,50%					dp,		,50%				,50%		
14 WhenDef	dp,100%	Ext,dp,50%	,50%	,100%				,50%	,50%	,50%			,50%	(14)	,50%							,50%				,37		
15 SplitDef	dp,50%	Ext,dp,50%	,50%			,50%		,75%	,50%	,75%		,50%	,50%		(15)	,50%	dp,	,50%		dp,		,50%				,75%		
16 DelayDef	dp,33%	Ext,dp,44%	,33%			,33%	,33%	,67%	,33%	,67%	,33%	,67%	,33%		,67%	(16)	dp,33%	,67%	,33%	dp,	,33%	,33%	,33%	,33%	,33%	,67%	,33%	
17 Processor*Helper	dp,33%	dp,	dp,	dp,													(17)			dp,	dp,	dp,	dp,					
18 ThreadsDef	dp,33%							,33%		,50%		,33%			,33%	,33%	dp,	(18)		dp,						,50%		
19 OtherwiseDef	dp,100%	,50%	,50%	,50%				,50%	,50%	,50%	,50%	,50%	,50%		,50%	,50%		,50%	(19)		,50%	,50%	,50%	,50%	,50%	,50%	,50%	
20 RouteContext	,43%					,33%	,33%	,33%					,33%							(20)	,33%		,33%			,33%		dp
21 PolicyDef	dp,80%									,40%											(21)		dp,100%					
22 TryDef	dp,60%	,40%	dp,60%	,40%				,40%	,40%	,40%			,40%	dp,	,40%		dp,					(22)				,30	dp,40%	
23 TransactedDef	dp,56%																			dp,	,71		(23)					
24 PipelineDef	dp,100%	,100%	,100%	,50%		,50%	,33%	,100%	,100%	,100%	,100%	,100%	,100%		,100%	,100%	,33%	,100%	,100%		,100%	,100%	,100%	(24)	,100%	,100%	,100%	
25 SamplingDef	dp,33%	,33%	,33%					,33%	,33%	,33%	,33%	,33%	,33%		,33%	,33%		,33%	,33%		,33%	,33%	,33%	,33%	(25)	,33%	,33%	
26 MulticastDef	dp,43%					,43%		,43%		,43%					,43%		dp,	,43%		dp,						(26)		
27 FinallyDef	dp,60%	,50%	,100%	,40%				,50%	,50%	,50%	,50%	,50%	,50%		,50%	,50%		,50%	,50%		,50%	dp,100%	,50%	,50%	,50%	,50%	(27)	
28 InterceptStrategy	,50%																				,50%		,50%					(28)

(c) R-2.12.4, Age 11, #Files 28, Churn 2134

Figure 10.10: Camel Hub Debt Evolution-Anchor ProcessorDefinition

Table 10.3: Debt Costs Model Distribution

Project	#Ds	$R^2_{threshold} = 0.75$				$R^2_{threshold} = 0.8$			
		Lin	Log	Exp	Poly	Lin	Log	Exp	Poly
Camel	199	52%	19%	0%	30%	39%	20%	2%	39%
Cassandra	180	61%	7%	2%	30%	53%	6%	3%	39%
CXF	189	56%	12%	1%	32%	45%	10%	4%	41%
Hadoop	74	46%	7%	7%	41%	36%	8%	3%	53%
Hbase	204	65%	7%	2%	25%	62%	4%	2%	31%
PDFBox	85	59%	4%	5%	33%	39%	1%	9%	51%
Wicket	153	46%	22%	1%	30%	38%	17%	1%	44%

evolve into debts over time. As an example, consider the top Hub debt with anchor file *ProcessorDef* (referred to as $PDef$ in the following) in Camel (Figure 10.10). We have provided 3 snapshots of this debt—in release 2.0.0 (age 1), release 2.2.0 (age 2), and release 2.12.4 (age 11)—to show its evolution. Snapshots from age 3 to 10 are similar to age 11. "Ext" and "Impl" stand for "extend" and "implement", "dp" denotes all other types of structural dependencies.

In release 2.0.0(shown in Figure 10.10(a)), $PDef$ forms a hub with 10 member files: 3 files are its subclasses, 7 files are its general dependents, and $PDef$ structurally depends on all of them. Note that in this snapshot, all files, except *InterceptStrategy*, depend on *RouteContext* (column 5). The 11 files in this hub structurally form a strongly connected graph. According to the revision history, $PDef$ changes with all member files with probabilities from 50% to 100% (column 1). The dependents (on rows 5 to 11) of $PDef$ are highly coupled with each other. This is problematic in 3 ways: 1) the parent class $PDef$ depends on each subclass and each dependent class (unhealthy inheritance proposed by Mo et al. [2015]); 2) the parent class is unstable and often changes with its subclasses and dependent classes (unstable interface Mo et al. [2015]). 3) *RouteContext* forms cyclic dependencies with 9 files (cycles). Without fixing these flaws, we expect the maintenance costs of this group to grow.

In release 2.2.0 (shown in Figure 10.10(b)), the impacts of this hub have enlarged—$PDef$ has 3 more subclasses and 6 more general dependents, and it depends on each of them as well. Each newly involved file also depends on *RouteContext* (column 13). The revision history shows that $PDef$ changes with its subclasses and dependents with probabilities of 33% to 100%. Also, the subclasses and dependents (rows 5 to 11) of $PDef$ are highly coupled with each other—changing any of them is likely to trigger changes to the rest. In following releases, the hub grows further. Up to release 2.12.4 (shown in Figure 10.10(c)), $PDef$ has 9 subclasses and 18 general dependents—the size of the hub tripled compared to the start, and, as always, $PDef$ depends on each of them. In addition, 6 of the 18 general dependents (rows 11 to 16) of $PDef$ also become its grandchildren. The inheritance tree has increased in width and depth. The revision history shows $PDef$ still changes with its dependents with probabilities from 33% to 100%. The files in this snapshot are tightly coupled with each other, and so changing any file is likely to trigger changes to others.

The maintenance costs spent on this debt fit a linear regression model: $DebtModel(rt) = 158.75 * rt + 509.35$ with $R^2 = 0.89$. This means that, in every release, developers contribute 158.75 more lines of code to fix bugs in the hub anchored by $PDef$. Although this model can only be obtained *after* the costs and penalty have accumulated, one could use our approach to detect architecture flaw patterns at any point (as described in Mo et al. [2015]), monitor how file groups grow, monitor the formation of debts, and prevent significant costs by investing in proper refactorings (Kazman et al. [2015]).

10.5 Limitations and Threats

We now briefly discuss the limitations and threats to validity for the ArchDebt quantification approach.

First, since we have only examined 7 projects and all of these are Apache projects, we can not guarantee that our results will generalize to other projects.

Second, similar to the analysis of the evolution of long-lived ArchRoots, the ArchDebt quantification approach relies on enough revision history as well. For projects without enough history data, our approach can still identify groups of files with the potential to become architectural debt. The building of a $DebtModel$ relies on having adequate history data. But our pattern matching approach is still feasible for projects with short history. We plan to evaluate the effectiveness of our approach on projects without enough history in our future work.

Third, as have been dicussed in the prior chapters, our approach relies on mining error-prone files from the revision history and bug tracking data. We use the bug report id that developers enter into commits to locate error-prone files. The availability and accuracy of such information heavily depend on the project's protocols. This is both a limitation and threat to validity to our approach.

Finally, we can't guarantee that error-fixing churn is the best maintenance effort approximation proxy. In our future work, we plan to explore more proxies, such as the amount of communications, the turn-around time for bug reports, etc. We are currently collaborating with an industry project that records real effort data, and we plan to compare this with our proxy measures of effort in our future work.

10.6 Summary

In this chapter, we formally defined a special form of TD, called Architectural Debts, on which maintenance "penalties" keep accumulating due to flawed architectural connections among files. And we contributed an approach to automatically identify groups of files involved in ArchDebts by matching four typical architectural flaw patterns. We quantified the maintenance costs spent on each ArchDebt, and monitored the growing trend of each debt to model its “interest rate”. We used four types of regression models to describe stable, increasing, decreasing, and fluctuating interest rates.

In the application on seven open source projects, this approach identified true debts that generate and grow significant (up to the 85% of the total) maintenance costs in these projects. Most interestingly, the most expensive and high-impact ArchDebts don’t involve any direct structural dependencies, instead groups of files are heavily coupled in revision history. This indicates the lack of design to better encapsulate undocumented assumptions shared among files. In addition, stable interest rates are found to be the most common interest type. In other words, during each release cycle, the developer team have to devote a stable amount of effort to fixing bugs involving files in an ArchDebt. Lastly, we illustrated how an architectural flaw evolved into a debt over time using an ArchDebt we identified.

The ArchDebt approach has further bridged the gap between software architecture and maintenance quality, built upon the DRSpace modeling and the ArchRoot detection. Software practitioners can use this approach to analyze and manage the architectural flaws that contribute to the maintenance difficulties in a systematic and automatic way. We believe that our approach has great potential in the early identification and prioritization of the concrete refactoring opportunities in software projects. Based on the cost and interest rate of each ArchDebt, informed decisions can be made in terms of whether, where and when to refactor, to fundamentally improve software quality as the long term goal.

Part V

Conclusions

11. Conclusions

In this dissertation, we have contributed a methodology that models, analyzes, and monitors software architecture in respect to addressing maintenance quality concerns.

First, we proposed a new architecture representation, the DRSpace modeling, embracing Baldwin and Clark [2000]'s design rule theory. It simultaneously captures the modular structure and the relevant maintenance quantify information of software architecture. It represents software architecture as multiple, overlapping DRSpaces. Each DRSpace represents a cohesive aspect of the architecture, which is composed of the leading files—the key design rules of the space—and the independent modules decoupled by the leading files. Each DRSpace captures the evolutionary couplings among files (which are not captured in existing architecture models) as a special form of architectural connections. Our studies have shown that the files led by bug-prone leading files are also likely to be bug-prone. Therefore, high-impact and bug-prone design rules should be given higher priority in bug fixing activities.

Based upon the DRSpace modeling, we further proposed an ArchRoot detection algorithm to automatically identify the most problematic DRSpaces of a system. We call these DRSpaces the ArchRoots of bug-proneness. We found, based on the studies of 15 software projects, that the majority of the bug-prone files in the projects are usually concentrated in the top few ArchRoots. It implies that the developers should focus on the top few ArchRoots with the highest concentration of bug-prone files to reap the largest benefits in bug-fixing activities. We also observed that some long-lived ArchRoots have persistent and significant impacts on the maintenance quality of a software project. We believe that the flawed architectural connections contained in these roots are the root causes of maintenance difficulties. Consequently, to fundamentally improve the maintenance quality in the long run, the developers should consider refactoring these roots to fix the architectural flaws.

Last but not least, based on the analysis of the ArchRoots, we formally defined a particular type of Technical Debt, the ArchDebts. An ArchDebt is a group of files that keep incurring high maintenance costs over time due to their flawed architectural connections. The developer team can pay off such "debts" in the project by refactoring, or they can take shortcuts by continuing to add new features. The former will delay the planed progress. The latter will lead to higher future maintenance "penalties". To solve this dilemma, we contributed an approach to automatically identify such "debts" by matching four typical architectural flaw patterns. Each identified "debt" is a potential refactoring opportunity. Our studies have shown that this approach can identified true and significant debts that worth attention. In the projects we studied, the identified ArchDebts consume up to 85% of the total maintenance effort. To further support informed refactoring decision-making, we quantified the key parameters—the "costs" and the "interest rates"—on the "debts". We prioritized the identified "debts" according to these parameters. It turned out that the most high-impact and expensive debts involve groups of files frequently change together without any direct structural dependencies. This suggests shared "secrets" among these files that should be better encapsulated. Ultimately, our approach enables software practitioners to make informed refactoring decisions based on the systematic analysis, rather than one's intuition or experience.

In summary, the methodology introduced in this dissertation has demonstrated great potential in bridging the gap between software architecture and maintenance quality. To the best of our knowledge, it is original in directly and systematically linking software architecture and maintenance quality concerns. We envision that this methodology has the potential for changing how software architecture is analyzed, monitored, and maintained in practice for addressing maintenance quality concerns.

12. Future Directions

The research in software architecture has huge potential in solving various practical problems in software engineering. Based on the studies and techniques presented in this dissertation, the following directions are particularly valuable and accessible.

(1)**Increase software architecture awareness in maintenance activities.** The work presented in this dissertation has revealed that architecture problems could be the root causes of error-proneness and high-maintenance costs in software projects. Compared to fixing the architecture problems by refactoring, preventing the introduction of such problems in the first place could be more advantageous. Increasing architecture awareness in the development environment can help developers to avoid introducing expensive architecture flaws.

Automatically recognizing and monitoring the evolution of applied design patterns (canonical design solutions for recurring problems in software design) in a software project could be a way of increasing architecture awareness. For example, when a developer commits a change that breaks a design pattern in the project, he/she shall be notified, or even be disapproved, of the violation. In our *DRSpace* model, each design pattern in a software system can be represented using a separate *DRSpace*, but, how to *automatically* recognize and monitor the evolution of design patterns using the DRSpace model still needs to be explored. I plan to explore such potential of our *DRSpace* model to increase architecture awareness in development environment.

(2)**Facilitate testing using software architecture.** Software architecture could be used to facilitate testing. There may exist test dependencies among components with architectural dependencies among one-another. The integration test of a set of components in a system is not ready to execute, until all the involved components have "passed" status in the unit test. In order to increase test efficiency, unit tests of components without architectural dependencies should be maximally paralleled; unit tests of components with dependencies should be properly ordered. In addition to that, integration test on a set of components in a system can start as soon as all involved components have "passed" status in the unit test. Before the test cases are written, or even before the details of components are implemented, the arrangement of parallel and sequential unit tests, as well as early integration test plans, can be computed from the architecture of a software project. Our *DRSpace* model captures the design rules and independent modules in software systems. The tests of independent modules could be maximally paralleled. I plan to explore the potential of software architecture, using our *DRSpace* model, to facilitate test activities.

Facilitating testing based on architecture is particularly valuable for modern software system, which is composed of other sub-systems. In such scenarios, using the high level architecture view to guide testing is crucial for testing efficiency. And the APIs that connect sub-systems should be tested the first and the most thoroughly.

(3)**Identify architecture problems which are responsible for quality attributes.** The design, implementation, and evolution of software architecture are driven by quality requirements. As a result, architecture is the foundation for achieving the quality attributes. However, the potential of software architecture in analyzing various quality problems, such as performance bottle neck, security pitfalls, scalability constraints, has not been fully explored. Whether and how architecture decisions affect the quality aspects have not been fully answered. For example, how to identify high latency architecture components that are the performance and scalability bottle neck in software systems? What could be the security vulnerabilities related to the SOA (Service Oriented Architecture) in web-based applications?

The work in this direction is very challenging, but is also extremely valuable, especially for software systems that emphasis, or even rely on, quality attributes to survive (e.g. performance, scalability, and security are important for software systems in the domains of cloud computing and big data). The research in this direction not only requires background in software engineering, but also requires backgrounds in related domains. I plan to collaborate with researchers who have expertise in the related domains to solve these problems.

Bibliography

Aldrich, J., Chambers, C., and Notkin, D. (2002). ArchJava: Connecting software architecture to implementation. In *Proc. 24th International Conference on Software Engineering*, pages 187–197.

Allen, R. and Garlan, D. (1994). Beyond definition/use: Architectural interconnection. In *Proceedings of the Workshop on Interface Definition Languages*, IDL '94, pages 35–45, New York, NY, USA. ACM.

Alves, N. S., Ribeiro, L. F., Caires, V., Mendes, T. S., and Spinola, R. O. (2014). Towards an ontology of terms on technical debt. In *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*, pages 1–7. IEEE.

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

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

Bansiya, J. and Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Eng.*, 28(1):4–17.

Basili, V. R., Briand, L. C., and Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761.

Bass, L., Clements, P., and Kazman, R. (2012). *Software Architecture in Practice.* Addison-Wesley, 3rd edition.

Bavota, G., Gethers, M., Oliveto, R., Poshyvanyk, D., and Lucia, A. d. (2014). Improving software modularization via automated analysis of latent topics and dependencies. *ACM Trans. Softw. Eng. Methodol.*, 23(1):4:1–4:33.

Briand, L. C., Wüst, J., Daly, J. W., and Porter, D. V. (2000). Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51(3):245–273.

Brondum, J. and Zhu, L. (2012). Visualising architectural dependencies. In *Proceedings of the Third International Workshop on Managing Technical Debt*, MTD '12, pages 7–14, Piscataway, NJ, USA. IEEE Press.

Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., and Zazworka, N. (2010). Managing technical debt in software-reliant systems. pages 47–52.

Cataldo, M., Mockus, A., Roberts, J. A., and Herbsleb, J. D. (2009). Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35(6):864–878.

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

Clements, P. C. (1996). A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, IWSSD '96, pages 16–, Washington, DC, USA. IEEE Computer Society.

Cunningham, W. (1992). The WyCash portfolio management system. In *Addendum to Proc. 7th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 29–30.

Curtis, B., Sappidi, J., and Szynkarski, A. (2012). Estimating the principal of an application's technical debt. *IEEE Software*, 29(6):34–42.

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

Eaddy, M., Zimmermann, T., Sherwood, K. D., Garg, V., Murphy, G. C., Nagappan, N., and Aho, A. V. (2008). Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, 34(4):497–515.

Falessi, D., Kruchten, P., Nord, R. L., and Ozkaya, I. (2014). Technical debt at the crossroads of research and practice: report on the fifth international workshop on managing technical debt. *ACM SIGSOFT Software Engineering Notes*, 39(2):31–33.

Fenton, N. E. and Ohlsson, N. (2000). Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8):797–814.

Freeman, E. T., Robson, E., Bates, B., and Sierra, K. (2004). *Head First Design Patterns*. O'Reilly Media.

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

Garcia, J., Krka, I., Mattmann, C., and Medvidovic, N. (2013). Obtaining ground-truth software architectures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 901–910, Piscataway, NJ, USA. IEEE Press.

Garlan, D. (2003). *Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events*. Springer Berlin Heidelberg.

Garlan, D., Bachmann, F., Ivers, J., Stafford, J., Bass, L., Clements, P., and Merson, P. (2010). *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2nd edition.

Griffith, I. and Izurieta, C. (2014). Design pattern decay: The case for class grime. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 39:1–39:4, New York, NY, USA. ACM.

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

Henry, S. and Kafura, D. (1981). Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518.

Huynh, S., Cai, Y., and Sethi, K. (2008a). Design rule hierarchy and analytical decision model transformation. Technical Report DU-CS-08-04, Drexel University. `https://www.cs.drexel.edu/node/13664`.

Huynh, S., Cai, Y., and Sethi, K. (2008b). Design rule hierarchy and model transformations. Presented at *Student Research Forum* of 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering. (Best Student Poster Award).

Jahanian, F. and Mok, A. K. (1994). Modechart: A specification language for real-time systems. *IEEE Trans. Softw. Eng.*, 20(12):933–947.

Jones, J. A., Harrold, M. J., and Stasko, J. (2002). Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA. ACM.

Kazman, R., Abowd, G., Bass, L., and Webb, M. (1994). Saam: A method for analyzing the properties of software architectures. In *Proc. 16th International Conference on Software Engineering*, pages 81–90.

Kazman, R., Asundi, J., and Klein, M. (2001). Quantifying the costs and benefits of architectural decisions. In *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on*, pages 297–306.

Kazman, R., Barbacci, M., Klein, M., Carriere, S. J., and Woods, S. G. (1999). Experience with performing architecture tradeoff analysis. In *Proc. 16th International Conference on Software Engineering*, pages 54–64.

Kazman, R., Cai, Y., Mo, R., Feng, Q., Xiao, L., Haziyevy, S., Fedaky, V., and Shapochkay, A. (2015). A case study in locating the architectural roots of technical debt. In *Proc. 37th International Conference on Software Engineering*.

Kazman, R. and Carriere, S. J. (1999). Playing detective: Reconstructing software architecture from available evidence. 6(2):107–138.

Kim, S., Zimmermann, T., Whitehead, J., and Zeller, A. (2007). Predicting faults from cached history. In *Proc. 29st International Conference on Software Engineering*, pages 489–498.

Kouroshfar, E., Mirakhorli, M., Bagheri, H., Xiao, L., Malek, S., and Cai, Y. (2015). A study on the role of software architecture in the evolution and quality of software. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 246–257.

Kruchten, P., Nord, R. L., and Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21.

Kruchten, P. B. (1995). The 4+1 view model of architecture. *IEEE Software*, 12:42.

Leszak, M., Perry, D. E., and Stoll, D. (2000). A case study in root cause defect analysis. In *Proc. 22rd International Conference on Software Engineering*.

Li, Z., Avgeriou, P., and Liang, P. (2015). A systematic mapping study on technical debt and its management. *J. Syst. Softw.*, 101(C):193–220.

MacCormack, A., Rusnak, J., and Baldwin, C. Y. (2006). Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015–1030.

Maldonado, E. and Shihab, E. (2015). Detecting and quantifying different types of self-admitted technical debt. *SIGSOFT Softw. Eng. Notes*.

Mancoridis, S., Mitchell, B. S., Chen, Y.-F., and Gansner, E. R. (1999). 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.

Maranzano, J., Rozsypal, S., Zimmerman, G., Warnken, G., Wirth, P., and Weiss, D. (2005). Architecture reviews: Practice and experience. *IEEE Software*, 22:34.

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

Martini, A. and Bosch, J. (2015). The danger of architectural technical debt: Contagious debt and vicious circles. In *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*, pages 1–10.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320.

Menzies, T., Greenwald, J., and Frank, A. (2007). Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13.

Meyer, B. (1988). *Object-Oriented Software Construction*.

Mo, R., Cai, Y., Kazman, R., and Xiao, L. (2015). Hotspot patterns: The formal definition and automatic detection of architecture smells. In *Proc. 15th Working IEEE/IFIP International Conference on Software Architecture*.

Mo, R., Garcia, J., Cai, Y., and Medvidovic, N. (2013). Mapping architectural decay instances to dependency models.

Moha, N., Guéhéneuc, Y.-G., Le Meur, A.-F., and Duchien, L. (2008). A domain analysis to specify design defects and generate detection algorithms. In *Proc. 11th International Conference on Fundamental Approaches to Software Engineering*, pages 276–291.

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

Nagappan, N., Ball, T., and Zeller, A. (2006). Mining metrics to predict component failures. In *Proc. 28th International Conference on Software Engineering*, pages 452–461.

Newton, P. and Browne, J. C. (1992). The code 2.0 graphical parallel programming language. In *Proceedings of the 6th International Conference on Supercomputing*, ICS '92, pages 167–177, New York, NY, USA. ACM.

Ohlsson, N. and Alberg, H. (1996). Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22:886–894.

Ostrand, T. J., Weyuker, E. J., and Bell, R. M. (2004). Where the bugs are. In *Proc. 13thACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 86–96.

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

Palsberg, J., Xiao, C., and Lieberherr, K. (1995). Efficient implementation of adaptive software. *ACM Trans. Program. Lang. Syst.*, 17(2):264–292.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–8.

Peters, L. (2014). Technical debt: The ultimate antipattern - the biggest costs may be hidden, widespread, and long term.

Posnett, D., D&#039;Souza, R., Devanbu, P., and Filkov, V. (2013). Dual ecological measures of focus in software development. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 452–461, Piscataway, NJ, USA. IEEE Press.

Ran Mo, Yuanfang Cai, R. K. L. X. and Feng, Q. (2016). Decoupling level: A new metric for architectural maintenance complexity. In *Proceedings of the 2016 International Conference on Software Engineering*, ICSE '16.

Robillard, M. P. (2008). Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology*, 17(4):18:1–18:36.

Sangal, N., Jordan, E., Sinha, V., and Jackson, D. (2005). 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.

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

Schwanke, R. W. and Hanson, S. J. (1994). Using neural networks to modularize software. *Machine Learning*, 15(2):137–168.

Seaman, C., Nord, R. L., Kruchten, P., and Ozkaya, I. (2015). Technical debt: Beyond definition to understanding report on the sixth international workshop on managing technical debt. *SIGSOFT Softw. Eng. Notes*, 40(2):32–34.

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

Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., and Zelesnik, G. (1995). Abstractions for software architecture and tools to support them. *IEEE Trans. Softw. Eng.*, 21(4):314–335.

Shull, F., Falessi, D., Seaman, C., Diep, M., and Layman, L. (2013). Technical debt: Showing the way for better transfer of empirical results. In Mnch, J. and Schmid, K., editors, *Perspectives on the Future of Software Engineering*, pages 179–190. Springer Berlin Heidelberg.

Stephen H. Edwards, Wayne D. Heym, T. J. L. M. S. and Weide, B. W. (1994). Specifying components in resolve. pages 29–39.

Terry, A., Hayes-Roth, F., Erman, L., Coleman, N., Devito, M., Papanagopoulos, G., and Hayes-Roth, B. (1994). Overview of teknowledge's domain-specific software architecture program. *SIGSOFT Softw. Eng. Notes*, 19(4):68–76.

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

Tzerpos, V. and Holt, R. C. (1997). The orphan adoption problem in architecture maintenance. In *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on*, pages 76–82.

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

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

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

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

Xiao, L., Cai, Y., and Kazman, R. (2014a). Design rule spaces: A new form of architecture insight. In *Proc. 36th International Conference on Software Engineering*.

Xiao, L., Cai, Y., and Kazman, R. (2014b). Titan: A toolset that connects software architecture with quality analysis. In *Proc. 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, FSE 2014, pages 763–766.

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

Vita

Lu Xiao received a Bachelor of Engineering in Network Engineering from Beijing University of Posts and Telecommunications in 2009. In 2014, she received the 1st Prize at the Student Research Competition at the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE). In the following year, her work, entitled *Detecting and Preventing the Architectural Roots of Bugs*, won the 1st Prize at the inter-disciplinary Grand Finals of the ACM Student Research Competition. In 2016, she was awarded the College of Computing & Informatics Outstanding Graduate Student Research Award at Drexel University.

Book Chapter:

- A Decision-Support System Approach to Economics-Driven Modularity Evaluation
  Yuanfang Cai, Rick Kazman, Carlos V.A. Silva, **Lu Xiao**, and Hong-Mei Chen
  in Ivan Mistrik, Rami Bahsoon, Rick Kazman, and Yuanyuang Zhang, *Economics-Driven Software Architecture, 1st Edition* , Pages 105-126, June 18, 2014, Morgan Kaufmann Imprint.

Journal Paper:

- Manufacturing Execution Systems: A Vision for Managing Software Development
  Martin Naedele, Hone-Mei Chen, Rick Kazman, Yuanfang Cai, **Lu Xiao**, Carlos V.A. Silva
  JSS 2015, Journal of Systems and Software, Volumne 101, March 2015, Pages 59-68.
  Impact factor 1.485.

Full Conference Papers:

- Towards an Architecture-Centric Approach to Security Analysis
  Qiong Feng, Rick Kazman, Yuanfang Cai, Ran Mo, and **Lu Xiao**

WICSA 2016, 13th Working IEEE/IF IP Conference on Software Architecture. Pages 221-230. Venice, Italy, April 5 - 8, 2016.

- Identifying and Quantifying Architectural Debts
  Lu Xiao, Yuanfang Cai, Rick Kazman, Ran Mo, and Qiong Feng
  ICSE 2016, Research Track, Proceedings of the 38th International Conference on Software Engineering. Pages 488-498. Austin, US, May 14 - 22, 2016.

- Decoupling Level: A New Metric for Architectural Maintenance Complexity
  Ran Mo, Yuanfang Cai, Rick Kazman, **Lu Xiao**, and Qiong Feng
  ICSE 2016, Research Track, Proceedings of the 38th International Conference on Software Engineering. Pages 499-510. Austin, US, May 14 - 22, 2016.

- A Case Study in Locating the Architectural Roots of Technical Debt
  Rick Kazman, Yuanfang Cai, Ran Mo, Qiong Feng, **Lu Xiao**, Serge Haziyev, Volodymyr Fedak, and Andriy Shapochka
  ICSE 2015, Industry Track, Proceedings of the 37th International Conference on Software Engineering. Pages 179-188. Florence, Italy, May 16 - 24, 2015.

- Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells
  Ran Mo, Yuanfang Cai, Rick Kazman and **Lu Xiao**
  WICSA 2015, Research Track, 12th Working IEEE/IFIP Conference on Software Architecture. Pages 51-60. Montreal, Canada, May 4-7 2015.

- A Study on the Role of Software Architecture in the Evolution and Quality of Software
  Ehsan Kouroshfar, Mehdi Mirakhorli, Hamid Bagheri, **Lu Xiao**, Sam Malek, and Yuanfang Cai
  MSR 2015, Research Track, Proceedings of the 12th Working Conference on Mining Software Repositories. Pages 246-257. Florence, Italy May 16 - 18, 2015.

- Design Rule Spaces: a New Form of Architecture Insight
  Lu Xiao, Yuanfang Cai, and Rick Kazman
  ICSE 2014, Research Track, Proceedings of the 36th International Conference on Software Engineering. Pages 967-977. Hyderabad, India, May 31 - June 7, 2014.

- A Replication Case Study to Measure the Architectural Quality of a Commercial System

Derek Reimanis, Clemente Izurieta, Rachael Luhr, **Lu Xiao**, Yuanfang Cai, and Gabe Rudy
ESEM 2014, Industry Track, Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Article No. 31. Torino, Italy, Sept. 18-19 2014.

- Measuring Architecture Quality by Structure plus History Analysis
  Robert Schwanke, **Lu Xiao**, and Yuanfang Cai
  ICSE 2013, Industry Track, Proceedings of the 2013 International Conference on Software Engineering. Pages 891-900. San Francisco, CA, USA, May 18 - 23, 2013.

- A Moving-Object Index for Efficient Query Processing with Peer-Wise Location Privacy
  Dan Lin, Christian S. Jensen, Rui Zhang, **Lu Xiao**, Jiaheng Lu
  VLDB 2012, Research Track, Proceedings of International Conference on Very Large Data Bases. Pages 37-48. Istanbul, Turkey, 2012.

Tool Demonstration:

- Titan: a Toolset that Connects Software Architecture with Quality Analysis
  Lu Xiao, Yuanfang Cai, and Rick Kazman
  FSE 2014, Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. Pages 763-766. Hong Kong, China, Nov 16-21, 2014.

Doctoral Symposium:

- Quantifying Architectural Debt
  Lu Xiao
  FSE 2015, Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. Pages 1030-1033. Bergamo, Italy, Aug. 30 - Sept. 4, 2015.