

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Erosão Arquitetural em Perspectiva: Um estudo
sobre regras arquiteturais, suas violações e como os
desenvolvedores lidam com o problema.

João Arthur Brunet Monteiro

Tese submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande como parte dos requisitos necessários para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Engenharia de Software

Dalton Serey (Orientador)

Jorge Figueiredo (Orientador)

Campina Grande, Paraíba, Brasil

©João Arthur Brunet Monteiro, 07/2014

DIGITALIZAÇÃO:
SISTEMOTECA - UFCG

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

M775e Monteiro, João Arthur Brunet.
Erosão arquitetural em perspectiva : um estudo sobre regras arquiteturais, suas violações e como os desenvolvedores lidam com o problema / João Arthur Brunet Monteiro. – Campina Grande, 2014.
94f. : il.

Tese (Doutorado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.

"Orientação: Prof. Dr. Dalton Dario Serey Guerrero. Prof. Dr. Jorge Abrantes de Figueiredo".

Referências.

1. Arquitetura de Software. 2. Estudos Empíricos. 3. Erosão Arquitetural de Software. I. Guerrero, Dalton Dario Serey. II. Figueiredo, Jorge Abrantes. III. Título.


CDU 004.2 (043)

"EROSÃO ARQUITETURAL EM PERSPECTIVA: UM ESTUDO SOBRE REGRAS ARQUITETURAIS, SUAS VIOLAÇÕES E COMO OS DESENVOLVEDORES LIDAM COM O PROBLEMA"


JOÃO ARTHUR BRUNET MONTEIRO


TESE APROVADA EM 11/07/2014



JORGE CESAR ABRANTES DE FIGUEIREDO, D.Sc, UFCG
Orientador(a)


DALTON DARIO SEREY GUERRERO, D.Sc, UFCG
Orientador(a)

MARCO TÚLIO DE OLIVEIRA VALENTE, Dr., UFMG
Examinador(a)


CLÁUDIO NOGUEIRA SANT'ANNA, Dr., UFBA
Examinador(a)


JACQUES PHILIPPE SAUVÉ, Ph.D, UFCG
Examinador(a)


TIAGO LIMA MASSONI, Dr., UFCG
Examinador(a)

CAMPINA GRANDE - PB



Universidade Federal
de Campina Grande

Declaro, para os devidos fins, que participei por videoconferência da apresentação da defesa da Tese de Doutorado de **João Arthur Brunet Monteiro**, e considero o trabalho aprovado.

Data: 11 / 7 / 2014

Assinatura do membro externo:

A handwritten signature in black ink is written over a horizontal line. The signature is highly stylized and cursive, consisting of several loops and a long, sweeping horizontal stroke that extends to the right.

Resumo

Erosão arquitetural é o processo de degradação da estrutura do software à medida em que se dá a sua evolução. Embora alguns trabalhos nessa área tenham apresentado exemplos de desvio entre a arquitetura planejada e a implementação do software em um momento específico do seu ciclo de vida, pouco se sabe a respeito dessa relação sob uma perspectiva evolutiva, isto é, como se dá esse distanciamento à medida em que o software evolui. Além disso, as abordagens propostas para verificação de conformidade apontam que o número de violações arquiteturais é tipicamente alto. No entanto, não há conhecimento a respeito da relevância dessas violações arquiteturais e como os desenvolvedores lidam com o problema durante o desenvolvimento do software. Esta tese apresenta três estudos empíricos cujo objetivo é aumentar o conhecimento sobre erosão arquitetural e como os desenvolvedores lidam com violações arquiteturais. Como resultado, em um primeiro estudo com quatro sistemas *open source*, foi possível demonstrar empiricamente o processo de erosão arquitetural em uma perspectiva evolutiva, além de demonstrar que poucas entidades de design são responsáveis pela maioria das violações arquiteturais. Além disso, através de uma análise quantitativa e qualitativa em 3 sistemas (Eclipse, BeeFS e epol), realizou-se i) uma caracterização de regras arquiteturais, ii) um estudo sobre a relevância das violações arquiteturais nesses sistemas e, iii) uma caracterização dos motivos que levam os desenvolvedores a cometerem violações arquiteturais. Por fim, com o intuito de entender a comunicação sobre aspectos de design/arquitetura em projetos *open source*, através da análise de dados de 77 sistemas, foi identificado que 25% das discussões em projetos mencionam algum aspecto de design e que poucos desenvolvedores contribuem para um espectro amplo de discussões. Esses poucos desenvolvedores são os que mais contribuem para o código projeto, isto é, há uma forte correlação entre commits e a quantidade de discussões que um desenvolvedor participa.

Abstract

Architectural erosion is the progressive lack of software structure over time. Previous studies on this subject concentrate on presenting conformance checking techniques and tools, and how effective they are in a single version of systems under analysis. However, there are still open research questions regarding the evolutionary nature of architectural violations. Besides that, little is known about the relevance of architectural violations and their impact on software development activities. This thesis describes three empirical studies performed to expand the current knowledge about architectural erosion phenomenon and how developers deal with architectural violations. As a result, in a first exploratory study with four open source systems, besides providing empirical data that shows the architectural erosion phenomenon in an evolutionary perspective, it is also demonstrated that few entities are responsible for the majority of architectural violations. Besides that, through quantitative and qualitative analysis in three systems (Eclipse, BeeFS and epol), this thesis presents: i) a characterization of architectural rules used in practice, ii) a study on the relevance of architectural violations of such systems, and iii) a characterization of the causes of architectural violations. At last, to provide knowledge on how developers conduct discussions about design/architectural aspects, this thesis presents an analysis on 77 open source systems which shows that on average 25% of the discussions in a project mention some design aspect and that very few developers contribute to a broader range of design discussions.

Contents

1	Introduction	1
1.1	Context	1
1.2	The Problem	3
1.3	Goal	4
1.4	Summary of the thesis	5
1.4.1	Implications	6
1.5	Outline of the document	7
1.6	Publications	8
2	Background	10
2.1	Software Architecture	10
2.1.1	Module View	12
2.1.2	Architectural Rules and Architectural Violations	13
2.1.3	Architectural Violations	14
2.2	Architecture Conformance Checking	15
2.3	Architectural Erosion	23
2.3.1	Definitions and related terms	23
2.3.2	Evidences of architectural erosion and its harmful effects	25
3	On the Evolutionary Nature of Architectural Violations	30
3.1	Contextualization	30
3.2	Study Design	31
3.2.1	Research Questions	31
3.2.2	Subjects	32

3.2.3	Data Collection	32
3.2.4	Procedures and Measures	34
3.2.5	Replication Package	35
3.3	Results	35
3.3.1	Addressing RQ1: How does the gap between code and architecture evolve over time?	35
3.3.2	Addressing RQ2: Are the violations equally spread over the design entities or they concentrate on a few ones?	39
3.3.3	Addressing RQ3: Once violations are fixed in a given version, do they appear again in future versions?	42
3.4	Discussion	44
3.4.1	Do not live with broken windows	44
3.4.2	Human factors	45
3.4.3	Critical core first	45
3.4.4	Recurring violations	45
3.5	Threats to Validity	46
3.6	Related Work	47
3.7	Summary	48
4	An Empirical Study of Architectural Rules and Violations	50
4.1	Contextualization	50
4.2	Study Design	51
4.2.1	Subjects	52
4.2.2	Data Collection Procedures and Analysis	53
4.3	Results	58
4.3.1	What kinds of architectural rules are expressed?	58
4.3.2	What kinds of architectural violations occur?	62
4.3.3	Which architectural violations are relevant to developers?	63
4.3.4	Why do developers commit violating code?	67
4.4	Threats to Validity	70
4.5	Discussion	71

4.6	Related Work	72
4.7	Summary	74
5	Do Developers Discuss Design?	75
5.1	Contextualization	75
5.2	Study Design	76
5.2.1	Data Set	77
5.2.2	Methodology	77
5.3	Results	79
5.4	Discussion	81
5.5	Related Work	82
5.6	Summary	83
6	Conclusions	84
6.1	Contributions	84
6.2	Future Work	85

List of Figures

1.1	Example of architectural rules regarding components and their relationships.	2
2.1	Example of architectural rules in module views. Green arrows are allowed dependencies, while red arrows are forbidden dependencies between modules.	13
2.2	Overview of Architecture Conformance Checking Technique.	16
2.3	The Reflexion Model Approach[1]	18
2.4	Architectural Model and Reflexion Models for NetBSD Virtual Memory Subsystem[1].	19
2.5	SAVE architecture conformance checking example.[2].	20
2.6	Architecture conformance checking with DCL and DCLCheck[3].	21
2.7	DSM for JUnit[4].	22
2.8	DSM Rule View[4].	22
2.9	Mozilla top level view[5]	26
2.10	FindBugs release 0.7.2[6]	27
2.11	FindBugs release 1.3.5[6]	28
2.12	Snippet of the reflexion model for Excel. Solid arcs are convergences; dashed arcs are divergences; dotted arcs are absences.[7]	29
3.1	Introduced and Fixed Violations per Version	36
3.2	Architectural debt per version. The line represents the Cumulative Architectural Debt.	36
3.3	Quantiles for the architectural debt	39
3.4	Distribution of violations per system. CV = classes with violations and CN = classes with no violations.	39
3.5	Frequency of classes per amount of violations (Version 1)	40

3.6	Distribution of violations per class	40
3.7	Peak of ArgoUML Fixed violations. Classes in the distribution tail were restructured.	46
4.1	Data sources	53
4.2	Amount of violations per Eclipse release over time	64
5.1	Methodology applied to build the design discussion classifier.	77
5.2	Empirical Results	80

List of Tables

3.1	Subject Systems	32
3.2	Top-10 data. DC = Number of different classes in Top-10 during the studied period.	41
3.3	Recurring violations data. RV = #Recurring Violations and MDR = The modal value of the degree of recurrence.	43
3.4	Total number of violations in version 19	44
4.1	Subjects	52
4.2	Architectural rules expressed in each system, <i>S</i> , <i>P</i> and <i>O</i> are sets of design entities which for these systems were packages, classes, interfaces, methods or fields	59
4.3	Violations per project	62
4.4	Relevance of Violations	64
4.5	Causes of architectural violations	67

Chapter 1

Introduction

1.1 Context

Software architecture has become a key concern to reach success in software development. During the last twenty years, specially along the decade of 1990, a number of software engineering researchers dedicated effort to define this concept [8; 9; 10; 11; 12]. Despite the fact that these definitions address different perspectives of architecture (e.g.: dynamic, static or external environments), most of them rely on components and their relationships. In this context, it is a commonplace to state that “software architecture is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them.” [12].

An architecture can be seen as a set of architectural decisions/rules [13]. Since the most popular mechanism to describe an architectural rule relies on structural properties of components [14], structural architectural rules are often used to describe important decisions taken during design phase. For example, usually, architects decompose the structure of the system using layers to create a clear separation between presentation, business logic and data access objects of an application. Figure 1.1 informally illustrates architectural rules related to layered systems. For instance, among other constraints specified, presentation objects must not directly depend on data access objects.

Most software practitioners regard architectural rules as fundamental concern to develop software. Good architectures are frequently credited for easy to maintain and evolve software systems, and as a sign of internal quality. However, implementations that do not follow

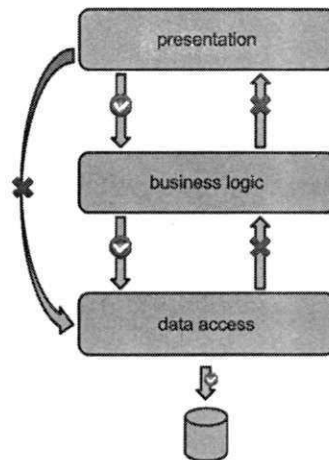


Figure 1.1: Example of architectural rules regarding components and their relationships.

intended architectural rules are frequent enough to be considered the norm, not the exception [15; 7].

Several reasons can cause divergences between intended and implemented architecture. In particular, researchers refer to the lack of awareness about the architectural rules as one of these root causes [16; 8; 17]. Frederick Brooks, for example, coined the term *Conceptual Integrity* to express the uniformity of the understanding that the development team has about the software [17]. Brooks states that it is better to have one good idea than a system without conceptual integrity – many uncoordinated and inconsistent architectural rules. In this scenario, communication is the key concern to achieve, maintain and enhance conceptual integrity. Architectural rules should often be discussed and spread over the team to avoid lack of conceptual integrity. However, while performing adaptive maintenance to accommodate new features, besides the inherent complexity of adding new code, developers have to deal with existent complex structure. As a result, as time goes by, systems evolve and their complexity increase, unless work is done to maintain or reduce it [18]. One of the main reasons that leads to the increasingly complexity of the system is the addition of dependencies between code entities that were not designed to be coupled to each other – an example of architectural violation. In the example aforementioned, an architectural violation would be a method dependency between classes of `presentation` module and classes of `data access` module.

The cumulative growth of architectural violations cause architectural erosion [8], also referred by the literature as structural degeneration [19], code decay [20], design erosion [21], and several other terms. Despite this vast terminology and some divergences between the definitions, all the authors relate the term to the progressive lack of software structure, a clear manifestation of software aging [22].

Architectural erosion is not an accidental event. The literature indicates several cases, including popular open source (e.g.: Mozilla, FindBugs and Ant) and proprietary systems (e.g.: Microsoft Excel and Axis), that eroded over time [5; 6; 23; 7; 21]. In this context, Mozilla is one of the most remarkable cases. First, because it was initially designed to be the open-source version of Netscape and, for this reason, developers decided to use the existent source code as basis for Mozilla. However, as the system evolved, due to structural complexity, developers decided to redevelop Mozilla from scratch. Second, because even this new re-thought architecture of Mozilla significantly eroded in its short lifetime [5].

The literature investigates and documents the harmful effects of architectural erosion [24]. The impact of this problem is always associated with maintenance costs. In the worst case (e.g.: Mozilla and Axis), when architectural erosion is neglected over time, the system reaches a state in which it demands total redevelopment, because maintaining eroded architecture is costly and cumbersome. Even when the system does not reach the worst case, architectural erosion still compromise maintenance activities. It is worth to remember that Mozilla lost large portion of the browser market due to inability to perform adaptive maintenance [25].

1.2 The Problem

Architectural erosion is an important problem that must be understood in order to be controlled. In this context, architecture conformance checking approaches [1; 26; 27; 3] play an important role because they uncover architectural violations, thus, they can detect architectural erosion over time. Although the state-of-the-art of architecture conformance checking is advanced, there is still lack of knowledge in some aspects that can leverage the adoption of this technique, such as, what kinds of rules are expressed by developers and what is the relevance of architectural violations detected by architectural conformance approaches.

Moreover, previous studies on this subject concentrate on presenting conformance checking techniques and tools, and how effective they are, by applying them in one single version of the system under analysis. There is still lack of knowledge on aspects such as lifecycle and location of architectural violations, and how developers deal with this problem during software development. When studying architectural erosion, it is important to consider the time dimension, once the concept requires an evolutionary perspective understanding.

In summary, **the knowledge about architectural erosion is limited**. In particular, little is known about the relevance of architectural violations detected by architectural conformance checking approaches, how developers perceive and deal with this problem during software development, and what are the causes of nonconformances between model and implementation.

1.3 Goal

The main goal of this thesis is to investigate the architectural erosion phenomenon, its causes, and how developers perceive and deal with this problem during software development. I intend to provide a foundation to extend research into architectural conformance checking in order to leverage the adoption of these approaches during software development.

Our specific goals are described below:

- expand the current knowledge about architectural erosion phenomenon by approaching it through an evolutionary perspective,
- investigate architectural violations location and lifecycle over time,
- provide a characterization of the rules that developers express and the violations that occur in practice,
- provide quantitative and qualitative evidences about the relevance of architectural violations
- investigate the reasons that lead developers to commit violating code, and
- investigate how developers conduct discussions about architectural aspects.

1.4 Summary of the thesis

In this thesis, I addressed the lack of knowledge about architectural erosion and how developers deal with this problem during software development. Due to this fact, the outcome of this work is a set of empirical studies that raise knowledge on this topic rather than an approach and its evaluation. I raised empirical evidences from three studies. First, I studied the evolution of four widely known open-source systems, analyzing the lifecycle of more than 3,000 violations. This analysis led to the following observations: 1) development teams of all studied projects seem to be aware of the presence of architectural violations in the code and all of them do perform perfective maintenance aimed at eliminating such violations; 2) despite all effort, the number of architectural violations, in the long term, is continuously growing; 3) in all studied systems there is a critical core, i.e., just a few design entities are responsible for the majority of violations; and 4) some violations seem to be “respawning”, i.e., they are eliminated, but are likely to be back in future versions of the system.

After that, I conducted an empirical study to more broadly understand what rules about architecture developers want to and do express, the ways in which implementation violate expressed rules and how developers view gaps between the implementation and the architecture that occur. I analyzed three systems: the open-source Eclipse platform, a proprietary distributed system and a proprietary web-based system. Through this analysis I was able to provide: i) a characterization of the 880 rules expressed by developers and the 521 violations that occurred, ii) quantitative and qualitative data on the relevance of architectural violations and how developers deal with them, and iii) a characterization of the reasons that lead developers choose to sometimes violate intended architecture.

In a third study, I conducted an initial investigation on the presence of discussions related to design aspects in 77 open-source projects. I adopted the term design rather than architecture for two reasons. First, to enable interchanging the design term as an *activity* and as an *artifact*. Second, the discussions may contain a broader range of aspects compared to the architectural rules and violations that I have been addressing until this study. In this context, I provide quantitative data that shows that developers address design discussions through comments in commit, issues, and pull requests. To achieve this, I built a discussions’ classifier and automatically labeled 102,12 discussions from 77 projects. Based on this data, I make

four observations about the projects: i) on an average, 25% of the discussions in a project are about design; ii) on average, 26% of developers contribute to at least one design discussion; iii) only 1% of the developers contribute to more than 15% of the discussions in a project; and iv) these few developers who contribute to a broad range of design discussions are also the top committers in a project.

1.4.1 Implications

Through the quantification and characterization of aspects that have not been addressed before, this work contributes to better understand the architectural erosion phenomenon and how developers deal with it in practice. The knowledge raised in this thesis has some implications for current practice in both: i) expressing and checking architectural rules against implementation; and ii) support refactoring activities to fix architectural violations. For example, I found that the majority of architectural violations reported are either exceptions to the rules or irrelevant in the sense that developers do not address or deal with them during software development. This scenario empirically supports the idea that not only the code but also architecture evolves over time and rules have to be kept up-to-date in order to reflect the decisions taken. In this vein, researchers could focus effort to provide mechanisms to automate exceptions detection. By doing this, it would be possible to advance in both activities: architectural changes recommendation and classification, and prioritization of architectural violations reported according to its severity, as static analysis tools such as FindBugs [28] achieve. Moreover, although there are several irrelevant violations, I also found that developers consider some of the violations relevant and perform refactoring activities to fix them. This observation has some implications for further research on refactoring suggestions to fix architectural violations. Terra et al. [29] took a step towards this direction by suggesting, for example, the application of the move method refactoring. However, this is a fertile field that could be further investigated.

Also, researchers in this area often credit the design/architectural erosion problem to the lack of awareness about design decisions [16; 17], which leads developers to commit code without concerning design aspects. To support such argument, researchers usually perform study cases in a small sample of subjects by conducting interviews to qualitatively raise knowledge about how developers discuss design. However, the state-of-the-art falls short in quan-

titatively demonstrating how and which developers drive design discussions in a project. In particular, for open-source projects, design concerns are spread over discussions in commits, issues, and pull requests. To the best of our knowledge, there is no study that approaches such information to understand how developers drive design discussions in such environment. Although related research works provide great qualitative contributions in this field, a broader quantitative study can improve the foundation to extend research into design practice. As a consequence of better understanding how developers drive discussions, such study may enable practices to mitigate architectural erosion, for example.

1.5 Outline of the document

I organized the remainder of this document as follows:

- **Chapter 2** reviews the core concepts and studies related to this thesis,
- **Chapter 3** describes the study design, results and analysis of an empirical study conducted to assess architectural erosion over time, answering the following research questions:
 - How does the gap between code and architecture evolve over time?
 - Are the violations equally spread over the design entities or they concentrate on a few ones?
 - Once violations are solved in a given version, do they appear again in future versions?
- **Chapter 4** presents an empirical study of architectural rules, including a categorization of rules, data on the relevance of architectural violations and a categorization of causes that lead developers to commit violating code, answering the following research questions:
 - What kinds of architectural rules are expressed?
 - What kinds of violations occur?
 - Which violations are relevant to developers?

- Why do developers commit violating code?
- **Chapter 5** presents an initial study to investigate the following questions:
 - To what extent do developers discuss design in open-source projects?
 - Which developers discuss design?
- **Chapter 6** presents the final remarks as well as the future work related to this thesis.

1.6 Publications

As a result of this thesis, I have published or submitted for publication the following papers:

- Brunet, J.; Bittencourt, R. A.; Guerrero, D.; Figueiredo, J. *On the Evolutionary Nature of Architectural Violations*. In Proceedings of the 19th International Conference on Reverse Engineering (WCRE 2012), Kingston, Canada, October 2012.
- *Five years of Software Architecture Checking: A Case Study of Eclipse*. Brunet, J; Murphy, G. C.; Serey, D; Figueiredo, J. *Minor review (06/17/2010)* at IEEE Software.
- Bittencourt, R. A.; Brunet, J.; Murphy, G. C.; Guerrero, D.; Figueiredo, J. *Using Software History to Prioritize Violation Warnings from Software Architecture Conformance Checkers*. *Under review* at Journal of Software and Systems (JSS). Submission date: April, 22nd.
- Brunet, J.; Murphy, G. C.; Terra, R.; Figueiredo, J.; Serey, D. *Do developers discuss design?* In Proceedings of the 11th Working Conference on Mining Software Repositories (MSR), Challenge Track, 2014.
- Melo, I., Brunet, J., Guerrero, D., Figueiredo, J. *Verificação de Conformidade Arquitetural com Testes de Design - Um Estudo de Caso*. I Workshop Brasileiro de Visualização, Evolução e Manutenção de Software (VEM - CBSOft), 2013.

Although not directly related to this thesis, I have also published the following papers during my PhD studies:

-
- Brunet, J.; Serey, D., and Figueiredo, J. *Structural Conformance Checking with Design Tests: An Evaluation of Usability and Scalability*. Proceedings of the 27th International Conference on Software Maintenance (ICSM 2011), Williamsburg, USA, September 2011.
 - Terra, R., Brunet, J., Miranda, L. F., Valente, M. T., Serey, D., Castilho, D., and Bigonha, R. S. *Measuring the structural similarity between source code entities*. In 25th International Conference on Software Engineering and Knowledge Engineering (SEKE), pages 753-758.

Chapter 2

Background

In this chapter, I establish a background on the concepts required to enable the understanding of this thesis. First, I introduce a discussion regarding the definition of software architecture, architectural rules and architectural violations. After that, I provide an overview of architecture conformance checking approach and discuss the most important works on this area. Then, I discuss architectural erosion definitions and describe works that have empirically demonstrated this problem and its effects on software evolution.

2.1 Software Architecture

Due to its importance in industry and academy, the concept of software architecture has become a major discipline in software engineering. During the last 20 years, several attempts to define software architecture has been proposed. It is hard to find a unique and precise definition of this term, but both researchers and practitioners agree on the importance of this concept.

In the beginning of the decade of 90, Perry and Wolf presented a seminal work in which they proposed concepts' definitions and directions to this field [8]. Among other concerns, they were interested in distinguishing software architecture from software design. According to Perry and Wolf, software architecture is concerned about defining architectural elements, their interactions and the constraints involved in these interactions, while software design includes the detailed definition of interfaces of design elements, their algorithms and procedures, and data types.

Budgen has taken a similar approach to differentiate software design from software architecture [9]. The author discusses the existence of a high level design (architectural design) and a low level design (detailed design). Similarly to Perry and Wolf, Budgen states that the architectural design is concerned about components and their interactions, while detailed design describes algorithms and procedures, and detailed interfaces of low level abstractions.

A well known definition of software architecture is based on the concept of components and connectors [10]. This definition explores a runtime perspective of the software elements. In this context, components regard to the principal processing units and data storage, while connectors are related to interaction mechanisms between the components, which includes communication links and protocols, data flow, and access to shared data.

Garlan and Perry [11] define software architecture as a the structure of the components of a system, their relationships, and principles and guidelines governing their design evolution over time. By the same token, Perry and Wolf use a building architecture metaphor to state that software architecture is a set of architectural elements that interacts to each other and have a particular form [8]. The authors also state that the rationale for the choices made in defining an architecture is an important aspect to define this concept.

As we can see, the aforementioned definitions rely on the concept of components, their relationships and some other aspects related to the decisions taken during design phase. Although some of the perspectives of software architecture might be related to dynamic aspects and non-functional requirements, in fact, a typical approach to define software architecture relies on the structural properties of the software. For example, Bass et al. [12] coined a popular definition of software architecture that relies on structural properties. According to them, "software architecture is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them." Still, Clements et al. [10] explores a similar definition which states that a software architecture is "the set of structures needed to reason about the system, which comprise software elements, relationships between them, and properties of both."

Software architecture is related to a number of definitions, artifacts, decisions and strategies. Therefore, it is complex to express its documentation in one single view. One of the main reasons to document an architecture is to improve the communication and knowledge transfer among stakeholders. In this context, the more complex is the document describing

the architecture, the more difficult is to spread the knowledge among the team. For this reason, Clements et al.[10] defined three different views (module, component-and-connector and allocation) that are suitable for different communication purposes. In the module view, the elements are modules, which are units of implementations (e. g. components, packages and classes). Documents of this view describe structural properties of these modules, their responsibility and interactions between them. On the other hand, component-and-connector view relies on components' runtime properties, including for example, the communication protocols between them. It aims at describing, for instance, the major executing components and how they interact. The allocation view aims at describing the relationship between software components and external environments in which they are created and executed. For example, in this view, one can express the type of processor that an element will be executed on. In summary, module view expresses structural properties of implementation units, component-and-connectors view expresses their runtime properties, and allocation view express the relationship between software components and external environments, usually computation units.

2.1.1 Module View

As said before, a typical architectural description relies on structural information about components and their relationships. For this reason, the module view is one of the most used views to describe an architecture [14]. This is due to the fact that it includes styles to describe a number of relations suitable to express dependencies between object-oriented components. These styles are the following:

- *Decomposition style*: Allows architects to describe modules (e. g., a set of classes) and relate them by the “is a submodule of” relation;
- *Uses style*: Allows architects to describe dependencies of a specific module;
- *Layers style*: Useful to express a particular scenario of “uses” styles in which modules that compose layer N are only allowed to use services of layer N - 1;
- *Generalization*: This style supports the “inherits-from” or “is-an-instanceof” relation between two modules.

2.1.2 Architectural Rules and Architectural Violations

Software architecture can be described by a set of architectural rules and decisions [13]. Rules often rely on structural properties of the source code, which makes the module view the most popular mechanism to describe the rules [30]. In a module view, stakeholders specify modules and dependency constraints between them. For example, stakeholders may decompose the structure of the system using layers to create a clear separation between presentation, business logic and data access objects of an application. Figure 2.1 informally illustrates architectural rules that may apply to a layered system. In this example, among other specified rules, the presentation module must not depend on the data access module.

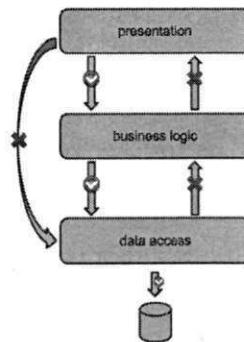


Figure 2.1: Example of architectural rules in module views. Green arrows are allowed dependencies, while red arrows are forbidden dependencies between modules.

In this thesis, I formally define an architectural rule as:

$$\mathbf{entity}_1 \langle \mathit{modifier} \rangle \langle \mathit{type\ of\ dependency} \rangle \mathbf{entity}_2 \quad (2.1)$$

I define each of these terms in turn:

entity can be a *module*, *type* or *code element*. A *module* is a set of *types*. Typical examples of modules are packages or subsystems, while classes and interfaces are examples of types. *Code elements* are entities from source code enclosed by a type (e.g., methods, fields, inner classes).

Modifier is one of the self-explanatory terms *must* or *must-not*.

Type of Dependency encompasses the common low-level dependencies between source code entities. Typical types of dependency in object-oriented languages are: method call, object creation, field access, generalization, realization, caught exception, thrown exception, returned type and received parameter. When stakeholders refer to the general concept of dependency between high-level entities, I use the general term *depends on*. For instance, the rule “*presentation must not depend on data access*” establishes that there must not be any low-level dependency between presentation and data access.

2.1.3 Architectural Violations

One of the main contributors to increasing system complexity is adding dependencies between code entities originally designed not to be coupled – an instance of architectural violation. For the architecture described in Figure 2.1, Code 1 illustrates an architectural violation, since a method from the presentation layer (`presentation.MainWindow.start()`) calls a method from the data access layer (`data.access.Data.getInfo()`).

Código 1 Example of Architectural Violation

```
1 package presentation;
2 public class MainWindow {
3
4     public void start(data.access.Data data) {
5         ...
6         data.getInfo();
7         ...
8     }
9 }
```

In the context of this work, the definitions are strictly related to code-level violations. I have been using an informal concept of violation throughout this work, but, for the sake of clarity, I shall define three levels of violations: code-level, type-level and module-level violations.

Code-level violation: A code-level violation is an unexpected dependency (namely, a di-

vergence in the reflexion model terminology [1]) between two source code elements (e.g., method, field, or class). It is uniquely defined by two participating code elements causing the violation and the violation type (e.g., field access, method call). A set of code-level violations between each two types compose a type-level violation. In the example illustrated by Code 1, the code-level violation is:

presentation.MainWindow.start() calls data.access.Data.getInfo()

Type-level violation: A type-level violation is an unexpected dependency (namely, a divergence) between two types (e.g., class, interface). It is uniquely defined by two participating types causing the violation. A set of type-level violations between each two modules composes a module-level violation. In the example illustrated by Code 1, the type-level violation is:

presentation.MainWindow depends on data.access.Data

Module-level violation: A module-level violation is an unexpected dependency (i.e., a divergence) between two modules defined in the high-level model. It is uniquely defined by two participating modules. One or more code-level violations are lifted to a module-level violation, through the mapping provided between source code and the high-level model (such as in the reflexion model technique). When it is an absence, it only exists in this level. When it is a divergence, it is made of one or more lower-level violations. In the example illustrated by Code 1, the module-level violation is:

presentation depends on data.access

2.2 Architecture Conformance Checking

The IEEE's Software Engineering Body of Knowledge [31] defines software verification as an activity which main goal is to assure that the internal properties of the software has been developed as intended. While software validation focuses on *what* has been implemented, software verification focuses on *how* the software has been implemented.

Both static and dynamic analysis are used to apply software verification. According to the IEEE's Software Engineering Body of Knowledge, static analysis collects information

(usually from source code) about the application under analysis without requiring its execution. On the other hand, dynamic analysis collects information by monitoring the system under execution.

Software verification is a key activity to guarantee software quality. Even though it is not directly related to functional requirements, assuring that a software is being implemented following architectural rules is a step towards to develop the right product.

Architecture conformance checking is the software verification technique that detects architectural violations. Several research works have been dedicating effort to apply architecture conformance checking, that is, verifying whether an implementation follows or not its architectural rules. In a nutshell, this activity is based on the comparison between a set of architectural rules and the implementation of the software. Figure 2.2 illustrates the overview of this activity.

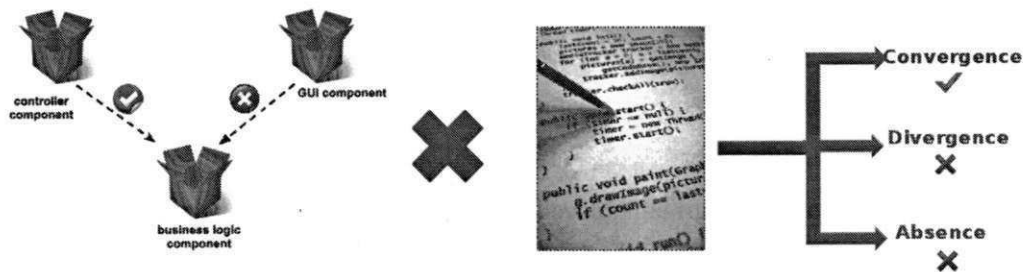


Figure 2.2: Overview of Architecture Conformance Checking Technique.

Basically, there are three categories to group the relations between these two artifacts [32]:

- **Convergence:** Indicates compliance between architectural rules and implementation. That is, the dependencies detected in the implementation are allowed to happen or were implemented as intended;
- **Divergence:** Indicates a not allowed relationship between two or more components in the implementation of the software;
- **Absence:** Indicates that the code does not implement a planned relationship.

Architecture conformance checking might be automated or not. For example, it is common to apply manual code inspections [33] and Design Review [34] in order to detect differences between planned and implemented architecture. In the context of agile methodologies such as XP [35] and SCRUM [36], pair programming and code review activities are used to verify a number of various software defects, including functional and non-functional specification's deviation.

In contrast to manual verifications techniques, there are a number of approaches that aims at automating architecture conformance checking. In the next subsections, I will discuss some of these approaches.

Reflexion Models

Murphy et al. [1] proposed one of the most known approach to bridge the gap between source code and high-level models - Reflexion Models. Figure 2.3 illustrates the approach. In a nutshell, in order to detect convergences, divergences, and absences, it is necessary to conduct the following activities. First, it is required to define the planned architecture. Second, the implemented architecture must to be extracted from the implementation. After that, elements of the planned architecture must to be mapped onto elements of implemented architecture. Then, one can check the compliance between architecture and implementation.

Figure 2.4 shows an example of reflexion model of the NetBSD Virtual Memory Subsystem. The architectural model (high level model) is on the left, while the reflexion model describing convergences, divergences and absences is on the right. The weight of edges indicates the number of low level relationships between two modules. The mapping of this example is illustrated by Code 2. Each line of the mapping associates entities in the architectural model with entities in the implementation.

By analyzing high-level models and mappings, one might ask whether reflexion models approach is feasible to be adopted in large projects. First, reflexion models approach is partial. That is to say, the approach does not require architects to specify all the modules and constraints of the architecture. This feature enables architects and developers to focus on the most relevant architectural rules to be verified. Besides that, to enable reflexion models adoption, Murphy et al. developed techniques to simplify engineer's task of defining planned architecture and mappings. Still, they developed tools to automate implemented architecture

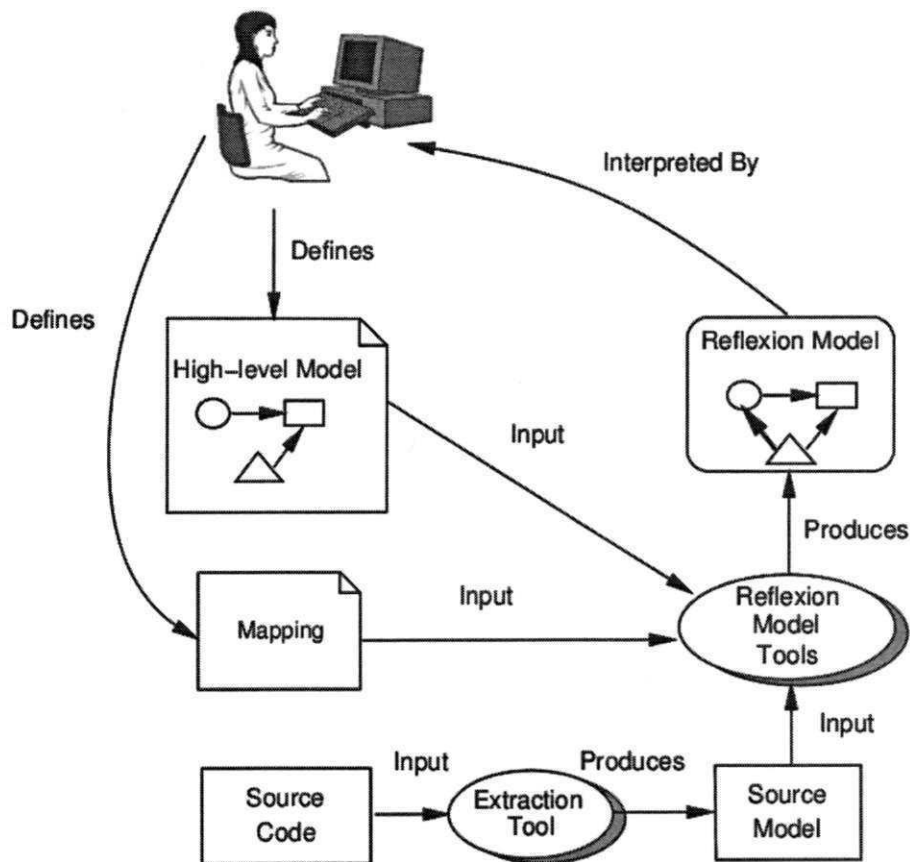


Figure 2.3: The Reflexion Model Approach[1]

extraction and compliance checking.

Código 2 Mapping of the NetBSD Virtual Memory Subsystem example[1].

```

1 [ file=.*pager.* mapTo=Pager ]
2 [ file=mn-xnap.* mapTo=VirtAddressMaint ]
3 [ file=vm_fault\.c mapTo=KernelFaultHdler ]
4 [ dir=[un]fs mapTo=FileSystem ]
5 [ dir=sparc/mem.* mapTo=Memory ]
6 [ file=pmap.* mapTo=HardwareTrans ]
7 [ file=vm_pageout\.c mapTo=VMPolicy ]

```

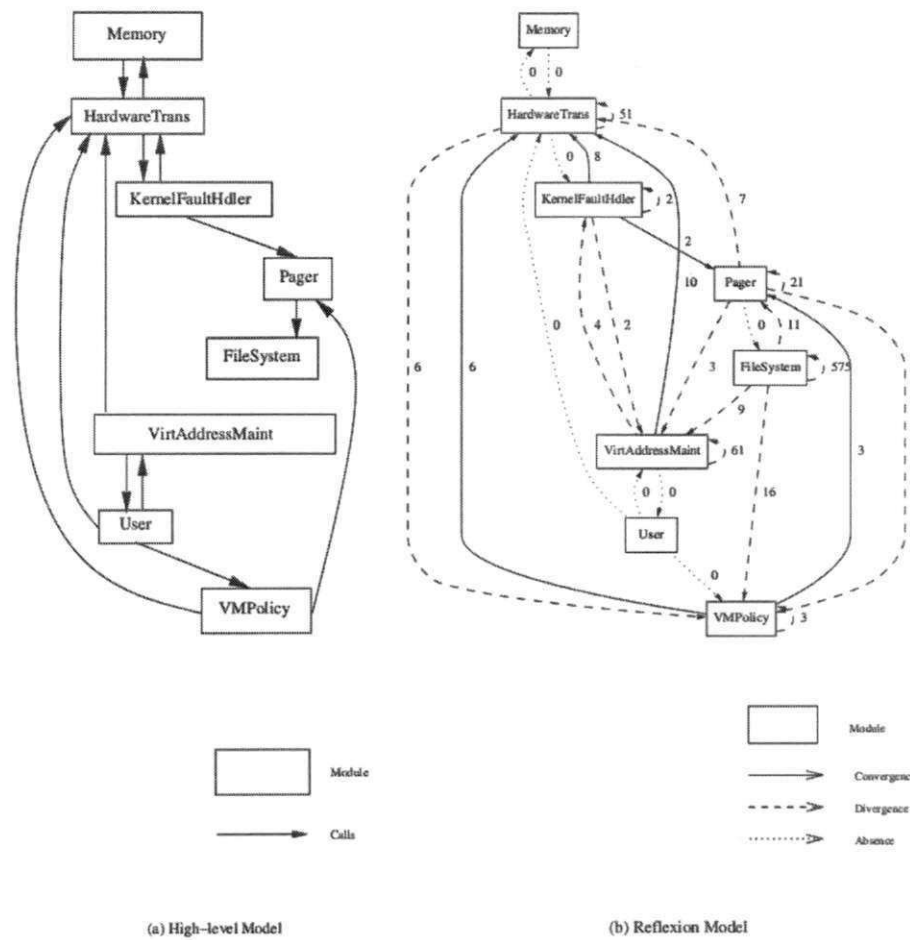


Figure 2.4: Architectural Model and Reflexion Models for NetBSD Virtual Memory Subsystem[1].

SAVE

SAVE (Software Architecture Visualization and Evaluation) is an Eclipse plug-in for evaluation of software architectures [2]. The tool allows architects to graphically express components and relationship between them and to assure compliance of existing systems with their architecture expressed. Figure 2.5 illustrates an example of architecture conformance checking using SAVE. The authors designed decoration items to express convergences (green check mark), divergences (yellow exclamation mark) and absences (red cross) in order to improve results' visualization. In addition, a blue question mark is used when two modules

have more than one type of these relations.

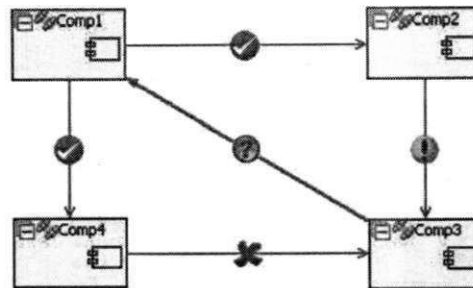


Figure 2.5: SAVE architecture conformance checking example.[2].

DCL Check

Instead of graphically expressing architectural rules, Terra and Valente [3] proposed an approach to apply architecture conformance checking by means of using a domain-specific dependency constraint language (DCL). DCL is a declarative language that provides a number of statements to define modules and constraints between them. Code 3 shows an example of architectural rule expressed using DCL. As we can see, modules and mappings are expressed by the `module` statement, while the dependency constraint between the modules is expressed by the `only...can-access` statement.

Código 3 DCL architectural rule specification example.

```

1 module GUI: org.foo.gui.*
2 module Controller: org.foo.controller.*
3 module BusinessLogic : org.foo.bl.*
4 only Controller can-access BusinessLogic

```

The approach also includes a conformance checking tool, named `dclcheck`, that automatically check architectural rules expressed in DCL language against Java implementations. Figure 2.6 shows an overview of the activities performed to achieve architecture conformance checking using the approach. First, using DCL, an architect expresses architectural rules based on the implementation of the system and some architectural model. Second,

`dclcheck` applies static analysis to verify the implementation against the specified rules. Then, the architectural violations detected are reported as architectural drifts.

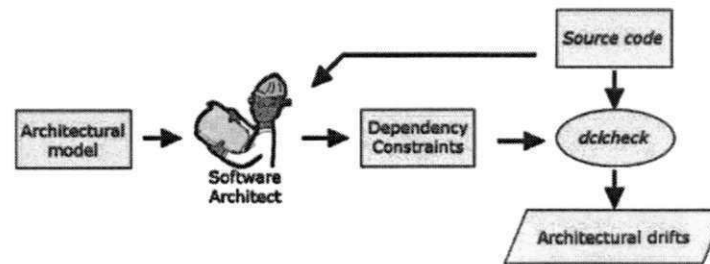


Figure 2.6: Architecture conformance checking with DCL and DCLCheck[3].

ArchJava

ArchJava [26] is an extension of Java programming language whose main goal is to ensure that the implementation conforms to architectural decisions. In the same way that Terra and Valente [3] provide means to programmatically specify architectural rules, ArchJava expands the Java programming language to support the concepts of components and ports. However, ArchJava performs dynamic analysis to verify architectural rules. In particular, the author address the problem of assuring a consistency property called communication integrity [37], which establishes that, during execution, implementation components should only communicate directly with the components they are connected to in the architecture.

Dependency Structure Matrix

Dependency Structure Matrix (DSM) is a square matrix that allows architects to describe dependencies between modules of a system [38]. In this representation, row and columns are used to denote modules, while a cross mark in a row A and column B denote that module B depends on module A. In some cases, architects may also use a number instead of a cross to indicate the strength of the dependency.

The idea to perform architecture conformance checking using DSM relies on the comparison between the the DSM extracted from code and the one expressed by architects. In this context, Lattix Inc's Dependency Manager [4] is one of the most popular tools to achieve

this. Figure 2.7 shows an example of DSM extracted from JUnit framework [39] binaries. The DSM shows, for example, that JUnit clearly separates user interface layers from the business logic, once `awtui`, `swingui` and `textui` modules do not depend on the other modules.

		1	2	3	4	5	6
JUnit	awtui	1	.				
	swingui	2		.			
	textui	3			.		
	extensions	4		1	.		
	runner	5	3	8	4	.	
	framework	6	5	7	6	6	5

Figure 2.7: DSM for JUnit[4].

		1	2	3	4
System	Subsystem1	1	.	1	
	Subsystem2	2	1	.	
	Subsystem3	3			.
	Subsystem4	4	1	1	

Figure 2.8: DSM Rule View[4].

To specify architectural rules, Lattix provides two options: i) a declarative language similar to DCL [3] and; ii) a graphical environment. Figure 2.8 illustrates an example of graphical architectural rules specification using the tool. Allowed and not allowed dependencies are denoted by green marks and black marks, respectively. A red triangle indicates an architectural violation.

Design Fragments

Fairbanks et al. [40] coined the term “design fragment” to refer to a pattern of how a program interacts with a framework to accomplish a goal. Through the Design Fragment Language,

architects are allowed to express the expected structure and behavior of developers' code and the expected structure and behavior of frameworks' code, including the existence of classes, methods, and attributes, and method calls between them. Then, by comparing design fragments and real implementations, one can verify whether the code is in conformance with framework design/architectural rules.

2.3 Architectural Erosion

This section is divided in two parts. First, I discuss architectural erosion definitions and related terms. Later, I discuss works that have demonstrated architectural erosion and its harmful effects on software structure.

2.3.1 Definitions and related terms

Lehman et al. have built an initial body of knowledge on software evolution establishing, among other concepts, the so called Lehman's laws [18]. One of these laws states that, as time goes by, software evolution and maintenance become increasingly hard and complex activities, unless the team dedicate effort to cope with this problem. This growing complexity inhibits developers to change the code in a proper manner, which leads to the lack of software structure [22].

Frederick Brooks was one of the first authors to explore software structural problems in detail as software evolves [17]. In his book, entitled *The Mythical Man Month*, Brooks discusses structural problems based on his own experience while developing a batch processing operation system called OS/360. According to him, software structure brittleness is a property that increases as software evolves and leads to the resistance to change, or at least to properly change the system. The main reason that leads to this scenario is the lack of conceptual integrity, a system property coined by Brooks to refer to the consistency and coherence of architectural decisions. The architecture of a system without conceptual integrity has an structural problem, named architectural drift [8], which is a state that a project reaches when the team lose control of the architecture. Architectural drift is closely related to the lack of conceptual integrity, once it is due to insensitivity of developers about the architecture.

Insensitivity is also referred as lack “architectural awareness” [16]. This term refers to several important aspects regarding the behavior of the team while evolving the software. In summary, a developer is aware of the architecture when she makes changes to a module respecting the architectural rules. The study performed by Unphon and Dittrich reinforces Brooks observations. By analyzing data of 15 semi-structured interviews, they could find that, due to lack of properly communication, developers tend to forget about the architectural decisions taken during design phase.

Insensitivity and lack of architectural awareness lead developers to introduce architectural violations in the code, which contributes to the increasing brittleness of a system [8]. The cumulative growth of architectural violations, thus, causes software structural deterioration – **architectural erosion**. Several terms are related to architectural erosion. Among them, code decay [20], design erosion [21], and architectural degeneration [19]. In a nutshell, all these concepts capture the notion of structural architectural problems related to the increasingly difficulty to maintain and evolve software.

Eick et al. used a medical metaphor to define and explore code decay [20]. In this context, code decay is seen as a disease, which has been caused by some reason and affects the system’s health. A code is decayed if it is “more difficult to change that it should be”. Due to the terminology, one might think that code decay relates only to low-level implementation issues. However, the study shows evidences of correlation between effort to implement changes and structural problems, once it negatively affects the coupling and modularity of a system. Similarly to Perry and Wolf, who stated that architectural erosion are cause due to architectural violations, Eick et al. identified that violations of the original design principles cause code decay.

Structural problems are also regarded as design erosion [21]. Based on industrial case studies, Gorp and Bosch identified a number of causes for this structural problem. Among them, they highlight that design decisions are difficult to track, due to the notations used to express them. However, unlike Parnas [22], who assume that, through hard work and cooperation, structural problems might disappear, Gorp and Bosh argue that design erosion is inevitable and can only be delayed. Still, they reinforce the necessity to address the causes of design erosion, not only focusing on its symptoms.

Architectural degeneration is also to refer to structural problems [19]. Again, the focus

remains on problems, causes and effects of structural degeneration during software lifecycle. As we can see, despite the fact that the literature adopts different terminology to refer to structural problems (from now I will use the term architectural erosion), researches on this area agree on the causes, effects and activities to delay or even stop structural problems. In summary, architectural erosion is due to architectural violations. The reason that lead developers to introduce architectural violations is insensitivity about architectural decisions/rules due to ineffective communication. Researchers also point that, if it is applied during software development, architectural conformance checking might be useful to early detect architectural violations and enhance sensitivity and awareness about architectural rules.

2.3.2 Evidences of architectural erosion and its harmful effects

A number of studies illustrate examples of systems that eroded over time. In this context, the web browser Mozilla [41] is one of the most popular cases in which architectural violations compromised its structure [5]. Mozilla was idealized to be the open-source version of Netscape project. When the development team decided to build it, the idea was to use the existent Netscape source code as basis for Mozilla. However, over time, developers realized that the code had structural problems that were making evolution harder that it should be. As a result, Mozilla was redeveloped from scratch, which demanded a lot of effort to be done. One can imagine that the new version of Mozilla was free of structural problems after the redevelopment of the system. However, Godfrey and Lee carried out an in-depth analysis of Mozilla and concluded that its architecture significantly eroded in its short lifetime. Among the reasons that caused this scenario, the coupling between unrelated modules of Mozilla increased the complexity of its structure. Figure 2.9 illustrates the top level view of the extracted architecture of Mozilla. In this figure it is possible to identify undesirable couplings between unrelated components. In particular, functional dependencies between the image processing library and the network and tools subsystems reveals architectural dependencies that should not occur – architectural violations.

The literature also describes the typical case in which developers have to implement a number of new features in short period and, thus, compromise software's structure [42]. This scenario is illustrated by the Axis system, which is a server printer that, in the beginning of its usage, only had to support one client device. However, as the system became popular,

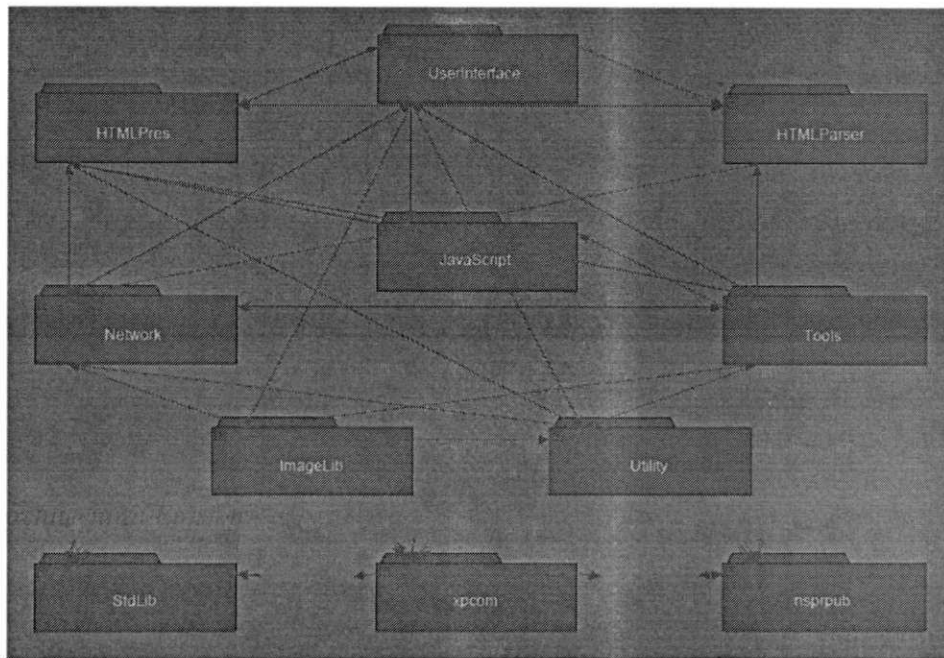


Figure 2.9: Mozilla top level view[5]

developers had to implement a series of new drivers to support a number of other client devices. Similarly to Mozilla, the development team decided to build the system from scratch, while maintaining the old software, until release the new one. As Gulp and Bosch reported, after few years of success using the new architecture, developers were developing a third new version of the system based on new architectural decisions. That is, despite the effort to cope with structural problems, due to architectural erosion, this system had to be redesigned two times in a short period.

Others widely used open source projects, such as FindBugs [28], Ant [43] and Linux Kernel [44] suffered architectural erosion over time. FindBugs, for example, in a period of approximately 4 years of development, evolved to an architecture with a number of cyclic dependencies between modules. Figure 2.10 and Figure 2.11 show the summary of this evolution. As we can see, in the beginning, the architecture had a few components and no cyclic dependencies between them. However, modules became interdependent over time. As a result, the architecture became a “tangle” of 20 modules connected to each other, compromising architecture understanding, evolution and maintenance [6].

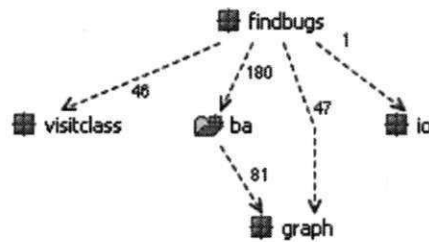


Figure 2.10: FindBugs release 0.7.2[6]

Regarding Ant project, by comparing two distinct versions (1.4.1 and 1.6.1) of this project, Dalgarno [23] could identify that the architecture became large and complex to be understood due to massive coupling between unrelated components. As he reported, in version 1.4.1, Ant had well defined and separated layers – taskdefs, ant, and utils. However, over time, these layers became complex and, mainly, interconnected to each other without respecting layers communication constraints. In particular, dependency from the lower-level *ant* layer to the top-level *taskdefs* layer had been introduced.

Linux kernel is another example of system that had to be redesigned due to architectural erosion. The version 2.4 of the kernel took almost two years to be released because the previous version (2.2) demanded a massive restructuring to enhance performance and accommodate new features [21].

Murphy and Notkin describe a case study in which a Microsoft engineer applies Reflexion Models technique to reengineer Excel project [7]. Although the authors focus on demonstrating the feasibility of the approach in the context of large real-world systems, it is also possible to note structural problems in the Excel project, once results of reflexion models computation indicates a large number of divergences (83) between the high level model and the implementation. As it had occurred in other examples, divergences regard to unexpected calls between two unrelated components. In the case of Excel project, 34 violations are due to dependencies between Sheet and File components, as Figure 2.12 illustrates.

Feikas et al. [15] also conducted an assessment in an industrial case studies to evaluate, among other aspects, architectural erosion and its causes. By analyzing the architectural conformance checking results in three case studies, they identified that between 9% and 19%

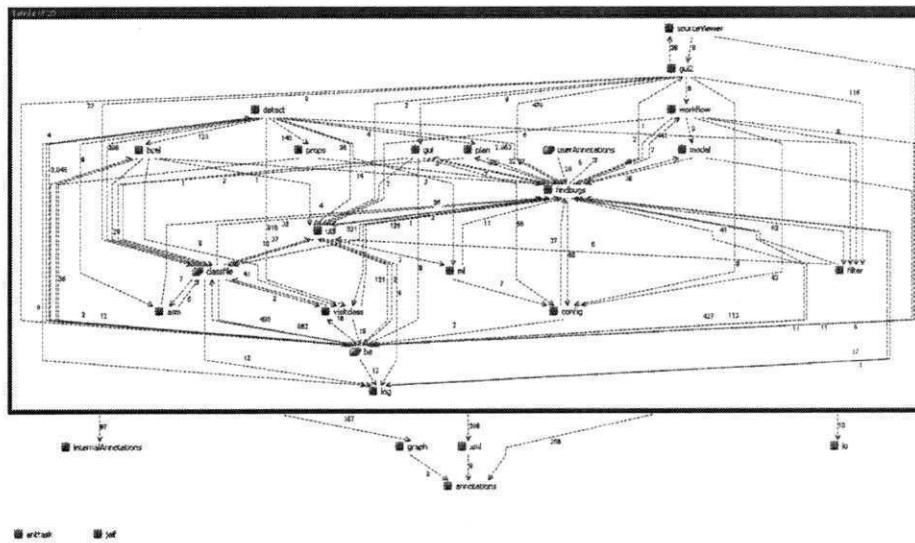


Figure 2.11: FindBugs release 1.3.5[6]

of all the dependencies are architectural violations, in the sense that they did not conform to the architectural rules. That is to say, the systems analyzed meaningfully diverged from the intended architecture. However, the authors found an interesting observation regarding these violations. A meaningful portion of them are due to deficiencies in the documentation. Based on this fact, the authors reinforce the need to continuously evolve the architectural model based on the ongoing decisions.

Architectural erosion was also identified by Rosik et al. during the *de novo, in vivo* development of a commercial system, named DAP [45]. The authors describe their experience in applying conformance checking during software evolution and developers' actions in face of the feedback given by the process of architectural conformance checking. As a result, they identified that the analyzed system diverged from the intended architecture and that developers tend to keep a number of violations unsolved. According to the authors, in most cases this is due to the risk of the changes, which comprise a number of restructuring activities.

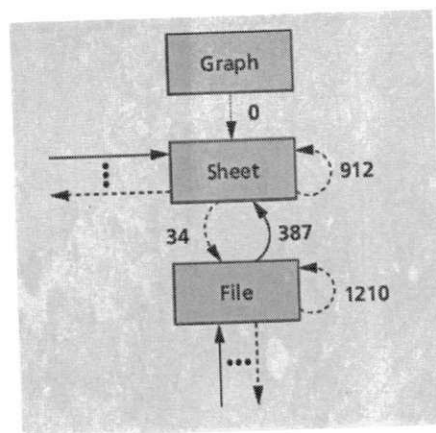


Figure 2.12: Snippet of the reflexion model for Excel. Solid arcs are convergences; dashed arcs are divergences; dotted arcs are absences.[7]

Chapter 3

On the Evolutionary Nature of Architectural Violations¹

3.1 Contextualization

While there are techniques and tools to detect violations and check whether an implementation conforms to a given architectural reference model, many violations go totally unobserved by development teams — sometimes even unsuspected. In fact, major releases of large and relevant software products have meaningful amounts of architectural violations [47; 7].

Previous studies on this subject concentrate on presenting conformance checking techniques and tools, and how effective they are. In this study, I take a different approach. I focus on architectural violations lifecycle and location over time rather than on identifying them in a single version of the software. In order to do so, I have performed a longitudinal and exploratory study on the evolutionary nature of the architectural violations of four open source systems. The main goal is to better understand how violations unfold as time passes and to build empirical knowledge regarding their temporal behavior. I use the reflexion model technique [48] as an example of static architecture checking technique, due to the easiness of deriving its required high-level module views from existing documentation of open source

¹Parts of this chapter appeared in the Proceedings of the 2012 Working Conference on Reverse Engineering (WCRE) [46]. In addition, an extension of this work is currently under review at the Journal of Software and Systems (submission date: Apr, 22nd).

systems.

In this chapter, I present the results of the aforementioned study as a first contribution to a body of knowledge of architectural erosion. The remainder of this chapter is organized as follows. Section 3.2 describes the study design. In Section 3.3, I present the experimental results. After that, in Section 3.4 and Section 3.5, I discuss results and threats to validity, respectively. In Section 3.6 I discuss related work, and finally, Section 3.7 concludes the chapter with final remarks.

3.2 Study Design

This section describes the experimental design conceived to guide the exploratory study. First, I present the research questions. Then, I introduce the subjects. After that, I describe the data collection. Finally, I present the applied experimental procedures and provide information about the replicability of this study.

3.2.1 Research Questions

To investigate how architectural violations evolve over time, their location, and how the development teams deal with them, I have formulated the following research questions:

- **RQ1: How does the gap between code and architecture evolve over time?**
I investigated the number of introduced and fixed architectural violations over time.
- **RQ2: Are the violations equally spread over the design entities or they concentrate on a few ones?**
I investigated the ratio between classes with violations and the total of classes, the distribution of violations per class and the classes with most violations.
- **RQ3: Once violations are fixed in a given version, do they appear again in future versions?**
I investigated the presence of recurring violations – violations that are fixed, but reappear from time to time.

3.2.2 Subjects

The subjects of the study comprise four Java systems. Table 3.1 shows, for each system, the studied period, their size (KLOC) and frequency of commits. Ant² is a popular Java-based build automation tool. ArgoUML³ is an open source UML modeling tool. Lucene⁴ is a text search engine library written entirely in Java. And, SweetHome3D⁵ is an interior design application that allows placing furniture in 2D plants with 3D previews.

Table 3.1: Subject Systems

Subject	Studied period (first / last)	Revisions (first / last)	Size (KLOC) (min / max)	# Monthly Commits (min / max)
Ant	Jan-2007 / Oct-2007	500,752 / 584,500	232 / 239	22 / 164
ArgoUML	Feb-2007 / Nov-2007	12,103 / 13,713	397 / 813	120 / 286
Lucene	Jun-2010 / Feb-2011	978,784 / 1,075,001	247 / 336	58 / 173
SweetHome3D	Jun-2009 / Feb-2010	2,069 / 2,382	75 / 96	6 / 99

The requirements for the subjects were: systems from medium to large size that contained architectural documentation; systems with frequent short-term commits; commits should happen on a daily or short-term basis to allow the generation of meaningful bi-weekly data; software versions should be available from software repositories using version control systems. In addition, there should be an adequate time frame for extracting empirical data (I used a nine-month development period for adequate longitudinal observation). Finally, system had to have compilable source code in Java due to the design extraction tool used, which reports facts from the bytecodes of Java systems [27].

3.2.3 Data Collection

To analyze architectural violations, some choices and assumptions were made about the experimental design, software versions and evolution period.

Assuming that an architectural module view remains stable for a longer period (e.g., some months between software releases), and that source code changes very frequently, sometimes more than once a day, I mined source code from software repositories at different time instants (bi-weekly), and computed architecture checks for each of these instants. Sampling should not be too frequent (commits) neither too sparse (releases). Too frequent sampling

²ant.apache.org

³argouml.tigris.org

⁴lucene.apache.org

⁵www.sweethome3d.com

leads to noise because they are more likely to be unstable changes. On the other hand, too sparse sampling implies very few data points to analyze. Furthermore, analyzing releases could raise another threat. The longer the period the more likely for the architecture reference model to change. Thus I produced bi-weekly violation lists for each bi-weekly source code version. In this study, I extracted 20 bi-weekly versions for each subject system in a period of nine months. The first version is used as a reference, and the other 19 versions have their violations analyzed, i.e., when they first appear and whether they are fixed, if that happens.

The violation lists were produced by applying the Reflexion Model technique [48]. In a nutshell, this technique consists in extracting high-level models, mapping the implementation entities onto these models and comparing the two artifacts, i. e., the high level models and the implemented design, checking where they agree and where they disagree. In this study, the high-level models were extracted from system documentation. SweetHome3D had design tests in the JDepend tool with packages as modules and assertions as the allowed dependencies between modules. Ant had a module view based on packages in the Lattix LDM tool. Lucene had a layered view diagram and I (in collaboration with other researchers) have performed the mapping, using the package names as the basis for module attribution. Finally, ArgoUML had the most detailed design documentation: a set of module views and the packages that made up each module. The high-level models represent relevant features of the systems, but they are not intended to be complete. Thus, some features can be missing in the models. The mapping for the four systems was performed through regular expressions based on the names of packages that made up those modules.

In order to better understand the quantitative data collected from violation lists, I also collected data from other sources. This activity was performed following two strategies: i) using SVN visual diff tool to compare subsequent versions of the software repositories and ii) manual inspection of the developers' public mailing list by date of interest. The former provided the code changes between two subsequent versions, which were used to explain some quantitative data. The latter provided data about architectural discussions and decisions of the development team, which were used to confirm some of the findings.

3.2.4 Procedures and Measures

Because the goal was to analyze the evolutionary nature of violations, I had to compute their lifetime. To accomplish this, I uniquely identified a violation through an *id*. This *id* is a tuple that contains the following information:

- *caller*: fully qualified name of the source code entity that violates the architectural rule;
- *callee*: fully qualified name of the source code entity used by the caller;
- *type*: violation dependency type. I considered the following types of dependency: method calls, field access, generalization, realization, caught and thrown exceptions, returned types and received parameters.

For the sake of clarity, let us analyze an example of an architectural violation:

- **caller**: `main.ConditionTask.execute()`
- **callee**: `gui.ConditionBase.countCond()`
- **type**: `calls`

This violation means that the `execute()` method from the `ConditionTask` class calls the `countCond()` method from the `ConditionBase` class. Given that this violation was detected in version *i*, it is trivial to find out whether it was fixed or not in the following versions (*i+1*, *i+2* ...). This allows us to compute a violation's lifecycle. For example, consider the following lifecycle for a hypothetical violation P:

P's lifecycle: v1 v2 v9 v10 v11

Analyzing the violations' lifecycle for all the studied versions, I were able to compute:

- introduced violations per version;
- fixed violations per version;
- architectural debt – the difference between introduced and fixed violations per version;

- amount of recurring violations;
- degree of recurrence – the amount of times that a violation reappears in the system.

As we can see, the violation id also allows us to identify not only the method that causes the violation, but also its class, package and module, once this hierarchy is defined in the high-level module view. Using this information, I also measured:

- the amount of classes with violations;
- the amount of violations per class.

3.2.5 Replication Package

The architecture module views for Ant, ArgoUML, Lucene and SweetHome3D were obtained from Bittencourt's Ph.D. dissertation [49]. I provide these models, raw and processed data, and the scripts used to obtain the results of this study in the URL: <http://code.google.com/p/on-the-nature-dataset/wiki/ReplicabilityOfTheStudy>.

3.3 Results

In this section, I present and analyze the results of the experiment in face of the questions raised during the experimental design. I address each question separately in both quantitative and qualitative perspectives. The quantitative analysis is based on the interpretation of the collected data whereas the qualitative analysis is derived from manual inspection of the repositories and the developers' public mailing list.

3.3.1 Addressing RQ1: How does the gap between code and architecture evolve over time?

In order to answer RQ1, the first step was to identify and count the amount introduced and fixed violations per version. Figure 3.1 shows the data collected for the four selected subjects. Each point in the figure represents either the number of introduced violations or the number of fixed violations (vertical axis) for a given version (horizontal axis).

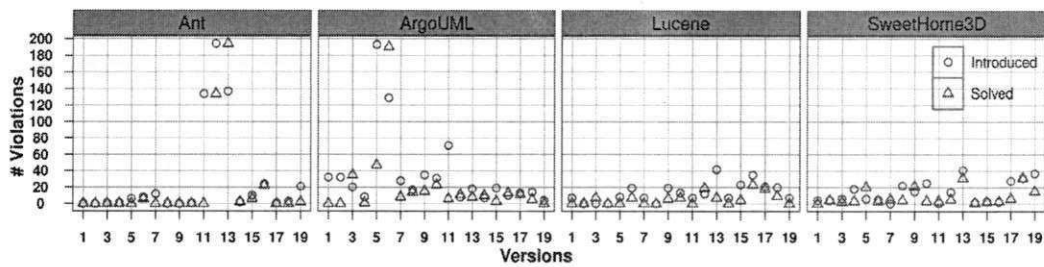


Figure 3.1: Introduced and Fixed Violations per Version

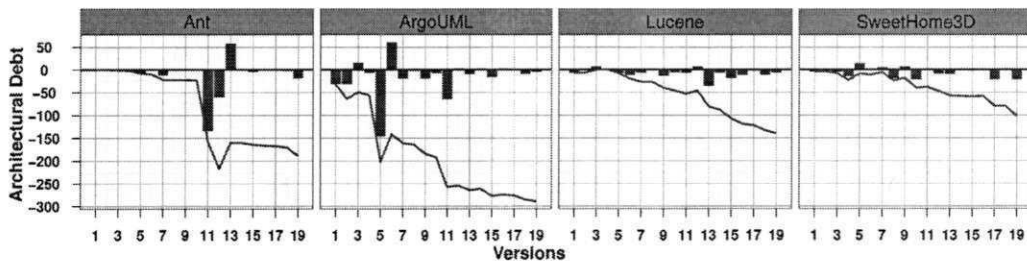


Figure 3.2: Architectural debt per version. The line represents the Cumulative Architectural Debt.

The second step was to observe how the architectural debt behaves over time. Given a software version, I define architectural debt as the difference between the amount of fixed and introduced violations. Figure 3.2 shows, as vertical bars, the architectural debt for each version. In addition, the line represents the cumulative architectural debt as software evolves.

Considering the amount of introduced violations per version shown in Figure 3.1, we can observe, in most versions, that few violations were introduced. The same occurs with the amount of fixed violations. It is worth noting that both for Ant and ArgoUML, the data collected shows that only a few versions introduce and solve a large number of violations. One can also notice that this occurs in consecutive versions, meaning that violations introduced in one version are usually fixed in the following version. As a consequence, the cumulative architectural debt increases in one version followed by a reduction in the following version (see versions 11, 12, and 13 for Ant, and versions 5 and 6 for ArgoUML in Figure 3.2).

Analysis

To conduct the qualitative analysis, I performed a manual inspection focused on adjacent versions in the repository. I conducted a more detailed inspection in versions with a large number of introduced and removed violations. For instance, versions 11, 12, and 13 for Ant, versions 5 and 6 for ArgoUML, and version 13 for both Lucene and SweetHome3D.

In fact, the qualitative analysis revealed a major restructuring period in Ant and ArgoUML. In ArgoUML, the analysis of the discussions between the developers during this period was quite enlightening. First, I detected that one developer performed a major change and communicated it to the rest of the development team, as we can see by his transcribed message: *“Yesterday I removed the old directory (org.argouml.uml.profile) and modified Argo code to use the code in the new directory (org.argouml.profile).”* This important change in the code introduced several violations since new classes with forbidden relationships were added into the (org.argouml.profile) package.

As Figure 3.1 shows, a significant number of architectural violations were fixed in version 6 of ArgoUML. The qualitative analysis revealed that two major changes in the code were responsible for this. Again, analyzing the mailing list and the commit messages (revision 12,455), I first found that one developer moved a class to its correct module, as can be seen by his transcribed message: *“ProgressMonitor does not belong in the GUI subsystem...Move the ProgressMonitor into its own subsystem.”* This change, combined with the removal of a cyclic dependence between two modules of ArgoUML (revision 12,407), decreased the cumulative architectural debt.

At last, I found a message from an important ArgoUML developer that summarizes the whole period of restructuring: *“I think it is time to start planning a 0.25.4 release to get all this together. I hope you agree.”*

Looking at Ant data shown in Figure 3.1 I identified a major restructuring period during versions 11, 12, and 13. An interesting fact that first caught attention is that the number of fixed violations in version 12 is identical to the introduced in version 11 (its previous version). Analyzing the repository and performing the diff between versions 10, 11, 12, and 13, I could visualize and understand what happened during this period. The restructuring was conducted as follows:

- 10 - 11: Class `FileUtils$3` was added to the project. This explains the large number of violations introduced.
- 11 - 12: The same class (`FileUtils$3`) was removed from the project and its code was moved to another class. This explains the identical number of introduced and fixed violations between these two consecutive versions and the large number of introduced violations as well.
- 12 - 13: Class `ProjectHelper` was restructured. The change comprised splitting its code in six other classes, which explains the large number of introduced and fixed violations.

Considering Lucene, various classes were added to the system in version 13 (revision 1,048,879). To be more precise, 26 classes were added to `util.automaton.fst` package, which is part of the `util` module. These classes have forbidden method calls to the `store` module. As a result, 42 architectural violations were introduced in the system.

Finally, in `SweetHome3D`, three classes were removed and their code were moved to `AppletApplicationClass`, which explains the number of introduced and fixed violations in version 13 (revision 2,210).

Another important aspect to point out is that, for all systems, developers usually perform perfective maintenance that aims to solve architectural violations. Moreover, when I analyze in each version the difference between fixed and introduced violations, the numbers suggest that, in most cases, the problem of architectural deviation is feasible to solve. Figure 3.3 shows the boxplot of the absolute value of the difference between the number of fixed and introduced violations per version. As we can see, in all systems, tackling up to the third quartile of violations seems to be feasible in a period of two weeks. However, as software evolves and the problem is not properly faced, the cumulative architectural debt, shown by the line in Figure 3.2, grows and the code tends to increasingly diverge from the intended architecture.⁶

⁶It is important to say that, due to the few restructuring moments, this function is not monotonically decreasing.

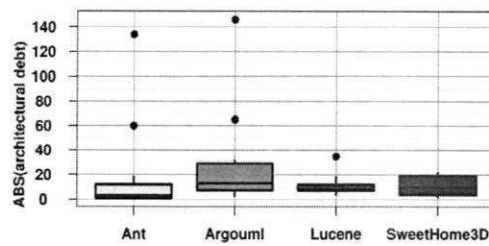


Figure 3.3: Quantiles for the architectural debt

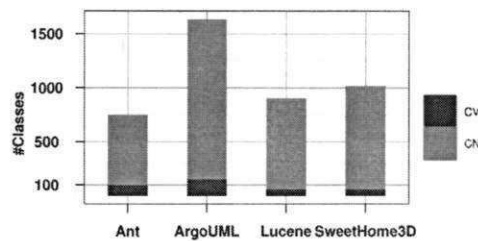


Figure 3.4: Distribution of violations per system. CV = classes with violations and CN = classes with no violations.

3.3.2 Addressing RQ2: Are the violations equally spread over the design entities or they concentrate on a few ones?

To answer RQ2, I collected data considering different relations between classes and violations. First, I investigated the ratio between classes with violations and the total of classes. Figure 3.4 shows the amount of classes with and without violations per system in version 1. It is important to mention that, in the experiment, I also looked at the other versions and found that the mean of classes with violations over time is 11%, 9%, 6% and 5% for Ant, ArgoUML, Lucene and SweetHome3D, respectively. Second, I analyzed the distribution of violations per class. Figure 3.5 shows the histogram of classes per violation. In each plot, the horizontal axis represents the amount of violations whereas the vertical axis stands for the frequency of classes. As we can see, few classes have many violations, while most classes contain very few violations. Again, these data regard only version 1 of each system. However, I have considered all the versions and found no significant variation among them.

The histograms suggest that the classes in the distribution tail are responsible for most of the violations. Figure 3.6 shows the cumulative proportion of violations per class, ordered by

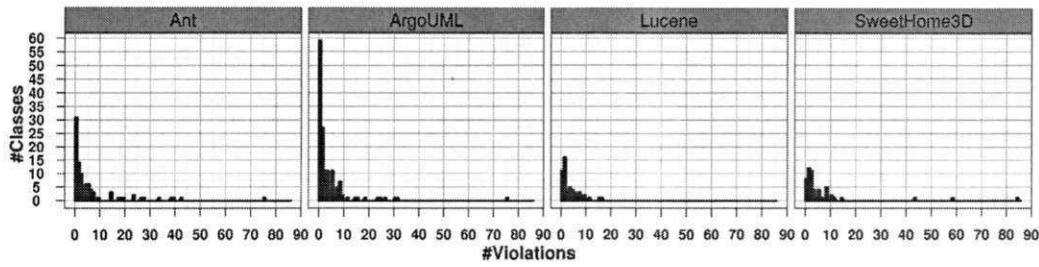


Figure 3.5: Frequency of classes per amount of violations (Version 1)

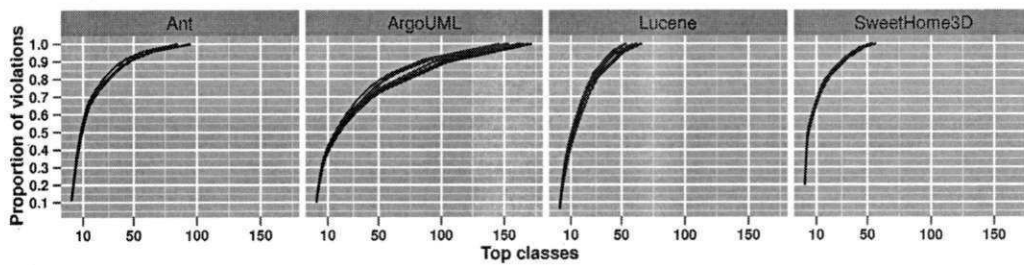


Figure 3.6: Distribution of violations per class

classes with most violations. Each curve in each plot represents one version. Figure 3.6 confirms the idea that few classes are responsible for most violations. Moreover, this behavior repeats in time, since the curves are very close to each other.

One last important aspect analyzed to answer RQ2 is shown in Table 3.2. For each system, the table presents: i) the mean proportion of violations caused by the Top-10 classes; ii) the number of different classes that appeared at least once in the Top-10 classes in the studied period (DC); iii) the mean ratio between DC and the total number of classes.

Analysis

The distribution of violations among classes revealed different issues. First, analysis of Figure 3.4 suggests that a small proportion of the whole system is responsible for the architectural violations. In fact, in the worst scenario (ArgoUML), at most 11% of the classes contain forbidden architectural relationships. Still, this proportion means, in this case, that

Table 3.2: Top-10 data. DC = Number of different classes in Top-10 during the studied period.

Subject	Top-10 proportion	DC	DC / Total
Ant	54%	12	1.8%
ArgoUML	40%	17	2.7%
Lucene	45%	16	6.2%
SweetHome3D	66%	12	2.9%

186 classes are responsible for architectural violations, which leads us to believe that such a large number of classes inhibits developers to cope with the problem. For this reason it is important to analyze the distribution of violations inside the classes with violations. Analysis reveals that very few classes contain a large number of violations (Figure 3.5). Therefore, the analysis of the violations inside the classes suggests the existence of a small number of classes responsible for a large proportion of the violations, since few classes in the distribution tail are responsible for a large proportion of the violations (Figure 3.6).

One important concern during the experiment was to restrict the analysis to a smaller group of Top-10 violating classes. If the Top-10 classes do not vary much in time, maintaining conformance might be easier, since the scope of architectural problems would be confined to a small number of classes. Confirming my intuition, Table 3.2 shows that the number of different classes (DC) that appear at least once in the Top-10 group is small. Moreover, it represents a rather small proportion of the total number of classes. In summary, the critical core comprises a small number of classes and does not change much in time.

Results from the qualitative analysis point that a core of classes is critical not only for the large number of violations that they contain, but also for two aspects: i) their role in the architecture; and ii) how restructuring changes in the core have a high impact on the amount of fixed relations.

In this context, regarding the role of the critical core, I found, for ArgoUML and SweetHome3D, that classes inside the `gui` and `swing` modules correspond to the majority of Top-10 classes. For example, I found that six classes in ArgoUML Top-10 group are responsible for the graphical interface. In fact, the class with most violations in ArgoUML is `ProjectBrowser`, which is part of the `gui` component. The large number of violations

occurs because `ProjectBrowser` is a presentation class and therefore needs information about various business logic objects, which leads to non-allowed coupling with various classes.

Another valuable information that reinforces my observation about the critical core is that, although I have analyzed ArgoUML data for 2007, since 2002 the `ProjectBrowser` class is an architectural concern to the development team, as the discussion below shows:

1. Developer A: "I just refactored `ProjectBrowser` to take out the construction of the `MenuBar`. Are there any objections against this?" (2002-10-10)
2. Developer B: "Take out the references to the project. Let the project be managed by another singleton class. Decouple `Main` and `ProjectBrowser`." (2002-10-12)
3. Developer C: "Refactor suggestion for `ProjectBrowser`: Take out anything to do with current themes and place this in its own singleton class." (2002-10-12)

Due to their application domain, `Lucene` and `Ant` do not have graphical interface modules. However, analyzing violations in these two projects, I found that the `util` module of both applications is critical. That happens because an `util` abstraction receives objects from many different classes to perform its actions.

Regarding changes in the critical core, I first hypothesized that the number of fixed relations is highly impacted by corrective changes in the Top-10 classes. Then, I analyzed peaks of fixed violations and identified these changed classes. Not surprisingly, for all the systems, peaks of fixed violations were caused by changes in the classes on the distribution tail. In other words, by changes in the critical core.

3.3.3 Addressing RQ3: Once violations are fixed in a given version, do they appear again in future versions?

In order to answer RQ3, I identified the number of recurring violations (RV) and their proportion over the total of fixed violations (RV / fixed). Besides that, I counted the amount of times that a violation reappears in the system (degree of recurrence). After that, I identified the statistical modal value of the degree of recurrence (MDR) considering the recurring vio-

lations. I use the modal value instead of mean or median because it is the most representative descriptive statistic of the data. Table 3.3 summarizes the data collected for each system.

Table 3.3: Recurring violations data. RV = #Recurring Violations and MDR = The modal value of the degree of recurrence.

Subject	RV	Total Fixed	RV/Fixed	MDR
Ant	343	366	94%	1
ArgoUML	44	400	11%	1
Lucene	21	107	20%	1
SweetHome3D	37	162	23%	4

Analysis

The quantitative analysis revealed that all the systems have recurring violations. The high number of recurring violations for Ant is explained by the rollback occurred during versions 11 and 12, as I have previously identified. Yet, recurring violations represent a meaningful number when compared to the total of fixed violations.

For all systems, analyzing the recurring violations' lifecycle, I found that a small number of violations were not definitely fixed during the studied period. For example, for Ant, 9 of the 343 violations were not fixed. It is worth saying that, due to the limitation in the timeline, I cannot assure that these violations were definitely removed from the system.

It is worth noting that, except for SweetHome3D, the modal value of degree of recurrence (MDR) is 1. It means that most of the recurring violations were fixed and appeared only once again.

One common approach to regard a problem as relevant is to identify whether it was addressed by the development team in earlier versions [50; 51]. What is clear when analyzing recurring data is that a meaningful proportion of architectural violations that were fixed earlier, i.e., relevant violations, are likely to reappear in the future.

It is not simple to assert why violations reappear over time. For example, analyzing Ant data, I observed that one of the factors that might cause this is a mistaken restructuring activity followed by a series of correction steps. Besides that, another aspect to take into account is the degree of knowledge of a developer in a specific area of the code. Unfortunately, due

to limitation of the data, it is not possible to identify the authors who were responsible for the introduced violations.

In summary, the number of recurring violations suggests that the problem exists and it cannot be ignored during the software evolution. This kind of behavior may indicate, for example, mistaken restructuring changes and the lack of architectural awareness by some developers [16].

3.4 Discussion

3.4.1 Do not live with broken windows

Through this exploratory study, I could gather some interesting insights about architectural drift. In particular, although the data does not empirically demonstrate such a conclusion, I believe that architectural drift seems to be related to the “Do not live with broken windows” [52] principle. Hunt and Thomas used this metaphor to highlight the importance of not letting small problems unrepaired in the code. In the context of architectural debt, the results suggest that the gap between code and architecture is tractable when violations are checked and then fixed in a short period (e.g., bi-weekly). However, as software evolves and small problems are not properly faced, tackling architectural drift can become unfeasible, as can be seen in Table 3.4. The table shows the total number of violations in version 19. This number regards the violations introduced and not fixed during all the systems’ lifecycle. That is to say, it represents the overall gap, not only the one observed during the studied timeline.

Table 3.4: Total number of violations in version 19

Subject	#Violations
Ant	637
ArgoUML	641
Lucene	305
SweetHome3D	429

3.4.2 Human factors

Qualitative analysis performed in this study revealed valuable information about: i) how developers deal with architectural issues; and ii) how changes in the code impact architectural drift. Although quantitative analysis reveals important facts about architecture erosion, I found that other sources of information improve the understanding about it. For example, developers' mailing list records gave detailed descriptions of architectural discussions and decisions. In summary, analyzing architectural issues involves observing not only source code and models, but also the human factors involved.

3.4.3 Critical core first

Changes in the critical core can produce great impact on architectural debt. For instance, Figure 3.7 shows one of the peaks of fixed violations in ArgoUML. As we can see, the number of violations of the classes in the distribution tail (critical core) had significantly decreased, i.e., they were moved to the left in the distribution.

Based on the results, it is possible to state that by addressing the critical core, developers can concentrate on the largest part of the violations, while having to deal with a small number of entities. However, it is important to make clear that this does not imply that less work has to be done.

3.4.4 Recurring violations

In the study, I also found that some violations are fixed, but reappear in future versions of the system. Analyzing this issue helps to reveal recurring architecture problems. These problems may be caused by several reasons, of which I highlight two: lack of architectural awareness [16] and lack of conceptual integrity [17]. In a nutshell, the former refers to the awareness of a developer about several aspects, including architectural decisions, while the latter refers to the uniformity of a mental model that the developers have about the architecture. I put these two terms in perspective because, if the recurrence was caused by the same developer that fixed it earlier, it may suggest that this developer is losing architectural awareness over time. On the other hand, if its recurrence was caused by a different developer, it may suggest lack of conceptual integrity.

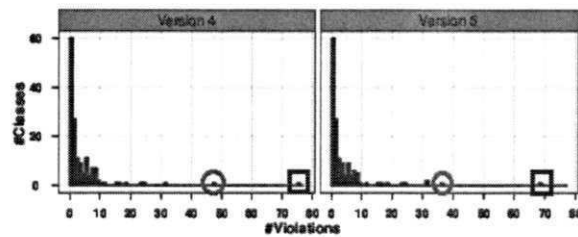


Figure 3.7: Peak of ArgoUML Fixed violations. Classes in the distribution tail were restructured.

3.5 Threats to Validity

It is important to state some aspects that might influence the observations. In this context, the main threat is the architecture module views that I have extracted from the subjects. Despite the fact that they were based on systems' architectural documentation, I still have to validate them with the architects and developers. However, in the qualitative analysis of the peaks of introduced and fixed violations, I manually inspected two sources of information to support the quantitative data: the commits and discussions in the developers' mailing list. Through this analysis, I did not find inconsistencies between the observations and what really happened in the systems, which leads us to believe that the architectural module views seem to be consistent.

Another important aspect regarding the architecture module views is that I assume that the architectural decisions remain stable during the studied period. This might affect the observations because a change in the module view of a system during the period of the experiment could generate different results. Nevertheless, as far as I could observe, the architectural decisions remained stable for all the studied systems.

It is also worth mentioning the renaming problem, i.e., given that a violation *id* is based on entity names, if this name changes, I consider the violation as fixed. I performed a qualitative analysis of the amount of renamed entities and found that, in only one of the subjects, namely ArgoUML, one of the top ten classes was renamed. Regardless of this exception, the approach was still correct in considering the violation fixed, because the renaming was caused by moving the class to its appropriate package. Nonetheless, I do realize that this is a particular case. Hence, I understand that further analysis is required, to account for name

changes.

Finally, results cannot be generalized to contexts different from the subject systems. I tried to reduce external validity threats, though, by choosing popular and long-life systems in an industrial-strength language (Java).

3.6 Related Work

Lehman have built an initial body of knowledge on software evolution establishing, among other concepts, the so called Lehman's laws [18]. In this context, several studies have been performed to analyze software in an evolutionary perspective. Godfrey and Tu [53], for example, investigated the growth of the Linux operating system kernel over time. The authors examined 96 kernel versions measuring their size in terms of the distribution package, LOC, number of functions, variables and Macros. As an important observation, the authors found that all measures revealed that development releases grow at a super-linear rate over time, contrasting Lehman's hypothesis of an inverse square growth rate [54; 55]. Gall et al. [56] performed a similar work on a large telecom switching system (TSS). As a major result, the authors found divergences between the whole system growth and its subsystems.

Some of the works in the software evolution area specifically focus on the architectural evolution over time. For example, van Gurp et al. [21] analyzed two case studies in order to investigate the common causes for design erosion, how the stakeholders identify this scenario and what are the common activities performed to address design erosion.

Hassaine et al. [57] proposed a quantitative approach to study the evolution of the architecture of object-oriented systems. The authors conceived a representation of an architecture based on classes and their relationships and used this representation to measure architectural decay by comparing with a subsequent program architecture. Hassaine and colleagues use the term architectural decay to refer to the deviation of the actual architecture from the original design. This study is related to this thesis because I both analyze architectural decay/drift over the time. However, instead of assuming that the first version of a system is the intended architecture, I use explicit architectural models extracted from the systems' documentation, which reveals that the inconsistencies are in fact architectural violations.

Another work closely related is a case study performed by Rosik et al. [45]. The researchers assessed the architectural drift during the *de novo, in vivo* development of a commercial system, named DAP. The authors describe their experience in applying conformance checking during software evolution and developers' actions in face of the feedback given by the process of conformance checking. As a result, they identified that the analyzed system diverged from the intended architecture and that developers tend to keep a number of violations unfixed. According to the authors, in most cases this is due to the risk of the changes, which comprise a number of restructuring activities. The work of Rosik et al. is similar in the sense that it aims to analyze the evolution of the gap between code and architecture. However, it is important to clarify some differences. First, I have analyzed four mature and architecturally stable systems, while they performed architectural conformance checking in one system since the beginning of its development. Moreover, my focus was not restricted only to architectural drift, but I also observed the location of the architectural violations and their lifecycle to respectively identify critical cores and recurring architectural problems. On the other hand, the results of this thesis corroborate with theirs in that implementation of a system tends to diverge from its intended architecture.

Wermelinger et al. [58] proposed an architectural evolution assessment framework based on quality metrics, laws, principles, and guidelines to address important questions about the architecture behavior over time. They focus on assessing architecture by analyzing quality principles, such as the Acyclic Dependency Principle and the Open Close Principle. We, however, did not evaluate the architecture through quality metrics. I were more concerned on how far it is from the intended one.

3.7 Summary

This chapter addressed the lack of knowledge on the evolutionary nature of architectural violations. I focused my effort on investigating the architectural drift over time, the location of the violations and their lifecycle. In particular, I addressed three main research questions:

- How does the gap between code and architecture evolve over time?
- Are the violations equally spread over the design entities or they concentrate on a few

ones?

- Once violations are fixed in a given version, do they appear again in future versions?

In order to provide answers to the aforementioned questions, I conducted a longitudinal and exploratory study. I performed conformance checking on four widely known open-source systems for which I have architectural models. In total, I analyzed more than 3,000 violations. From the quantitative and qualitative analysis, I observed that, in face of the questions raised during the study design: i) the number of architectural violations, in the long term, is continuously growing; ii) in all studied systems there is a critical core and this core does not change much over time; and iii) some violations are fixed, but reappear over time.

Chapter 4

An Empirical Study of Architectural Rules and Violations¹

4.1 Contextualization

Developers, designers, architects and other stakeholders often invest significant effort towards creating (and maintaining) good software architecture for their products. These investments are meant to pay-off over the lifetime of a software product through improved reusability, better adaptability for new features, and other benefits [59; 60]. However, should development deviate from the architecture, the long term pay-offs can be diminished, replaced with technical debts instead. To help ensure intended characteristics of a software architecture are carried into implementation, practitioners and researchers have developed a number of approaches and tools to help implementations retain consistency with their architectures. ArchJava [61], Domain Specific Languages (DSL) [62; 63; 64], and consistency checkers [1; 65] are but some tools and approaches that address this architecture deviation problem.

Although these kinds of approaches have been available for many years, little is still known about how the approaches work in practice. The studies that have been conducted have focused on checking architectural rules that are limited to expressing statements about how modules should or should not access each other (e.g., [66; 7; 46]). Furthermore, little is known about the relevance of the violations reported by conformance checking approaches.

In this chapter, I report on an empirical study conducted to more broadly understand

¹The content of this chapter is under revision (minor revision) at IEEE Software.

what rules about architecture developers want to and do express, the ways in which implementations violate expressed rules and how developers view gaps that occur between an implementation and its intended architecture. In contrast to earlier studies, the cases I consider include a broader set of rules, going beyond rules just about access (e.g., module A *must not* access module B) to also include rules about type hierarchies (e.g., class C may not be subclassed) and about object instantiation (e.g., class D *must not* be instantiated).

The empirical study involved three subject software systems: the open-source Eclipse integrated development environment,² a closed-source distributed file system called BeeFS [67], and a closed-source web inquiry management system for the Federal Police of Brazil called e-Pol. For 5 years, the developers of Eclipse have expressed architectural rules about constraints on Eclipse plugins. I analyzed existing official releases reports of violations of these rules that have occurred over 19 versions of Eclipse. For BeeFS and e-Pol, I interviewed developers to collect, express and verify rules. For each system, I quantified and analyzed the rules expressed and interviewed developers about the use of such rules.

This chapter presents the following contributions:

- a characterization of architectural rules and their violations that occur in practice,
- a quantitative and qualitative data on the relevance of architectural violations and how developers deal with differences between an implementation and its intended architecture, and
- a characterization of reasons that lead developers to commit violating code.

I begin by detailing the design of the study (Section 4.2) before presenting (Section 4.3) and discussing the results (Section 4.5). Then, I present the threats to validity (Section 4.4). After that, I continue by reviewing existing work in expressing and checking architectural rules (Section 4.6). Finally, I conclude with summary of the chapter (Section 4.7).

4.2 Study Design

Four research questions drove the quantitative and qualitative investigations into the practice of using architectural rules:

²www.eclipse.org, verified 7/9/13.

Table 4.1: Subjects

Project	#Packages	#Classes	#LOC
Eclipse Release 4.3	546	8909	4.4 MLOC
BeeFS	51	258	25 KLOC
e-Pol	41	599	57 KLOC

RQ1 : What kinds of architectural rules are expressed?

RQ2 : What kinds of architectural violations occur?

RQ3 : Which architectural violations are relevant to developers?

RQ4 : Why do developers commit violating code?

4.2.1 Subjects

To help answer these questions, I investigated the three Java systems: Eclipse, BeeFS and e-Pol.

Eclipse is an open-source platform and development environment whose architecture relies on the concept of plug-ins. The Eclipse developers have automated an approach to structural architectural rule checking. For the last 5 years, Eclipse included 48 plugins in its architectural rule conformance checking process. 32 of these 48 plug-ins do not appear in all releases. In this study, I focus on the 16 plug-ins that were included in every release over these 5 years and provide valuable historical information about the Eclipse platform plug-ins. As we can see in Table 4.1, these 16 plug-ins represent 4.4 million LOC organized in 546 packages and 8909 classes. I analyzed information from different versions of Eclipse; as information is discussed I specify the particular versions considered.

BeeFS is a closed-source distributed file system that harnesses the free disk space of machines already deployed in a local network. The system uses a hybrid architecture that follows a client-server approach for serving metadata and managing file replicas, and a peer-to-peer architecture for serving data. The project has been under development for 2 years by 9 developers and contains 25 KLOC organized in 51 packages and 258 classes (Table 4.1). The version of the software is dated June 5, 2013.

e-Pol is a closed-source web-based system which main goal is to automate tasks performed by Brazilian Federal Police, such as managing data on police investigations. The

project has been under development for 4 years by 12 developers and contains 57 KLOC organized in 41 packages and 599 classes (Table 4.1). The version of the software is dated April 6, 2013.

In addition to having access to the source code of each of these systems and the results of architectural rule checks, I also interacted with 24 developers, each of whom was involved with one of these three projects. Seven of the developers (29%) were associated with the Eclipse project; these developers had worked with the Eclipse project for between 4 and 12 years. Five of these developers have been committers since the beginning of the Eclipse project and three are part of the Eclipse Architecture Council, which is responsible for the development and maintenance of the Eclipse Platform architecture. Seven of the 24 (29%) developers were associated with the BeeFS project and had between one and two years experience with the project. Ten of the 24 developers (42%) were associated with the e-Pol project, each had between one and four years experience with the project.

4.2.2 Data Collection Procedures and Analysis

Figure 4.1 provides an overview of the data I collected and analyzed.³ Next, I describe the data collected and analysis procedures used for each of the four research questions in turn.



Figure 4.1: Data sources

Due to the projects heterogeneity, I applied mixed methods approach [68], collecting data from different sources for triangulation (Figure 4.1). Based on the research questions, I was mainly interested in data on **architectural rules**, **architectural violations**, **the relevance** and **the causes** of architectural violations. Next I describe how I gathered data from these different sources and how I analyzed this data to answer the research questions.

³All study data is available at www.dsc.ufcg.edu.br/~jarthur/icse2014

Architectural Rules

In the case of Eclipse, the developers had implemented an approach for checking architectural rules and had been expressing and checking such rules for 5 years independent of this study. For the other two systems, BeeFS and e-Pol, I interviewed developers to collect architectural rules.

Eclipse. In the Eclipse development, an API consists of public and well-documented packages, interfaces, classes, methods and fields. Every API is documented to express what it is supposed to do and how it is intended to be used. As part of expressing how an Eclipse APIs is intended to be used, the developers of Eclipse use the Plugin Development environment (PDE) / API Tool, a mechanism that provides Java annotations to restrict access to an API.⁴ Using this tool, Eclipse developers can express structural architectural rules that restrict extension of a class (`@noextend`), that restrict implementation of an interface (`@noimplement`), that restrict object creation (`@noinstantiate`), that restrict overriding of a method (`@nooverride`), and that ensure no use (`@noreference`). Besides these five restrictions, the PDE/API tool also ensures no references from external clients to a package with “internal” in its name, because they are not API elements and, for this reason, they are likely to change without official support to existing clients.

I was able to leverage these existing annotations in the code as the architectural rules for the Eclipse plug-ins included in the study.

BeeFS and e-Pol. Neither BeeFS or e-Pol were using an approach to express or check architectural rules. However, both systems were willing to participate in the study. To seed the architectural rules for each of these systems, I performed on-site interviews with one to two software developers from each project at their respective workplaces. Since the beginning of their projects, these developers are responsible for the architectural decisions involving the structure of the code to be implemented. I started these interviews by discussing the concept of structural architectural rules and asking each developer to describe instances of these rules in the context of their project. For BeeFS, I carried out two sessions of approximately one hour with two of its developers. For e-Pol, I conducted a one hour and thirty minute session to collect the rules. During the sessions, architectural rules were described informally using text, and box and arrow diagrams so that the language for explicitly expressing the architec-

⁴<http://www.eclipse.org/pde/pde-api-tools/>, verified 8/9/13

tural rules would not be an obstacle. An example of rule that was declared is: “MODEL *must not depend on* SESSION”. The second step was to map the design entities involved in the rules to the source-code entities, such as mapping MODEL to the EPOL.MODEL package.

For each project, after I collected the rules and mapped the design entities involved, I applied a technique called member checking [69] to gather feedback on the collected architectural rules. I achieved this by transcribing the rules from the whiteboard to a collaborative editing document and asking the developers to validate them. During this activity, there were only updates to the mappings.

To perform automatic conformance checking, I expressed the collected architectural rules as Design Tests [70], which automatically checks whether an implementation conforms to an architectural rule. Design tests can be written to express a wide variety of structural architectural rules including all of those expressible by the Eclipse PDE / API tool.

Architectural Violations

In addition to gathering data about the structural architectural rules, I also gathered data about how the architectural rules were violated within the implementations of the software projects.

Eclipse. I collected Eclipse architectural violations from the official PDE / API Tools' Verification Reports available in the public web sites for each release. Reports detailing architectural violations for the last 19 Eclipse releases (from 3.4 to 4.3) were available, totaling 5 years of historical data. Here is an example of a violation from a report: `JAVASOURCEVIEWER illegally extends PROJECTIONVIEWER`.

BeeFS and e-Pol. Once I composed the design tests to express BeeFS and e-Pol rules, I collected architectural violations for these two projects by executing the design tests on the last version of each system. The checker provided similar output to the Eclipse PDE / API tool output, for example: `MAIN.CONDITIONTASK.EXECUTE() illegally references GUI.CONDITIONBASE.COUNTCOND()`.

Architectural Violations' Relevance

To study how developers perceive architectural violations that occurred, I gathered both quantitative and qualitative data.

To study the relevance of an information, Schamber et al. [71] state that a study should go further than quantitative evaluation. It is a fact that relevance of an information involves how developers perceive it during software development. It is also a fact that qualitative research is suitable to answer and raise knowledge on questions of that nature. For this reason, besides quantitative evaluation, I applied qualitative research methods to better understand the relevance of architectural violations and to provide strong and reliable observations of development teams' behavior regarding architectural issues.

Eclipse. In the case of Eclipse, I had multi-version data about architectural violations. The first step to gather data about the relevance of Eclipse's architectural violations was to determine when violations that were introduced were fixed.

For each violation, I computed the violations' lifecycle. For example, let us suppose that a violation dv is present in the release 3.4 and 3.4.1, but not in the subsequent releases. Then, the lifecycle of this violation is:

dv 's lifecycle: 3.4 3.4.1

Given a violation's lifecycle, it is trivial to determine whether and when it was fixed. In the example above, the violation was fixed in release 3.4.2. Hence, through the analysis of violations' lifecycle for all the studied releases, I was able to compute the amount of fixed violations per release.

To get more in-depth information about the violations, I selected two moments of the Eclipse development history in which several violations were fixed: 3.5 and 4.3 releases. I then conducted a qualitative analysis on each one of these violations by performing the following steps:

- inspecting bug reports and commit messages related to the design entities involved in the violation in order to uncover changes and discussions regarding these entities,
- inspecting the source code repository to understand the changes performed to fix the violations, and
- initiating discussions on the developers' mailing lists to uncover the reasons for fixing some and not other violations.

These steps provided not only the code changes between releases, which were used to explain some quantitative data, but also provided data about architectural discussions and decisions of the development team, which were used to confirm some of the findings.

BeeFS and e-Pol. For these two systems, I presented the detected architectural violations to the software developers and asked them to classify each violation into an exception to the rule or actual violation. Whenever a developer classified a violation, I asked the following questions:

- Exceptions
 - Why is this violation an exception to the rule?
- Actual violations
 - Is this violation critical?
 - Would you fix it? Why? When?

By following these steps, I was able to quantify the amount of architectural violations that are **exceptions**, **actual violations** and **critical violations**. Moreover, I could collect qualitative data to explain why an architectural violation is an exception or critical.

Causes of Architectural Violations

To understand what development actions might lead to violations occurring, I collected data from: *i*) semi-structured interviews performed with developers; and *ii*) discussions that we conducted on developers' mailing lists. To do focus these interviews and discussions, I selected particular architectural violations and, for each one, I asked: "In your opinion, what are the reasons that led the developer to introduce this particular inconsistency?"

I collected responses for 10 violations for Eclipse, 13 for BeeFS and 8 for e-Pol. For Eclipse, I did not have sufficient access to the developers to collect data about all violations. For this reason, I randomly selected violations that were intentionally fixed during Eclipse source-code evolution and gradually presented the architectural violations in order to discuss them.

For BeeFS and e-Pol, I selected architectural violations that were considered critical by the developers. I selected 13 and 8 architectural violations for BeeFS and ePol because this is

a representative amount that covers the spectrum of classes that are involved in the violations. For example, if there are 10 illegal access between class A and B, I randomly chose only one of these violations. In both cases, participants were allowed to inspect the code to provide more accurate information on the causes of architectural violations.

I coded the text collected from semi-structured interviews and discussions[72]. This process involves extracting from the responses small phrases or sentences that can be organized into categories. Categories were defined based on the vocabulary of the responses and previous knowledge and experience achieved through the related works. This coding resulted in five categories: **Unawareness**, which includes responses that mention developers' unawareness about the architectural rules; **Ease**, which includes responses that express that committing an architectural violation is easier than the other alternatives to implement a feature; **Misplaced design entity**, which is related to design entities that are placed in the wrong modules or packages; **Copy and paste programming**, which includes responses that relate the causes to reusing violating code; and **Time constrains**, which includes responses that relate the causes to deadlines and time pressure.

4.3 Results

I present the results in terms of the four research questions.

4.3.1 What kinds of architectural rules are expressed?

Across the snapshots of the three projects, I found 880 architectural rules. The vast majority of these rules were expressed by the developers for Eclipse (838 rules or 95% of the total) while developers for BeeFS and e-Pol specified 18 (2%) and 24 rules (3%), respectively. Based on the vocabulary used in a architectural rule and the purpose of the rule, I classified the 880 identified rules into three categories: *general restriction*, *hierarchy* and *object instantiation* rules. Table 4.2 shows the breakdown of the rules into these three categories across the projects. I use the *general restriction* category for rules that express some restriction of reference or use between two design entities without specifying a particular kind of dependency. For example, Eclipse and BeeFS developers wrote rules such as the following: “*external clients must not depend on INTERNAL package*”. This rule means that any kind

Table 4.2: Architectural rules expressed in each system, S , P and O are sets of design entities which for these systems were packages, classes, interfaces, methods or fields

Rules	Eclipse (Release 4.3)	BeeFS	e-Pol
<i>General restriction rules</i>			
S must not depend on P , except for O	226	10	21
S can only depend on P		3	2
<i>Hierarchical rules</i>			
Class or interface C must not be extended	392		
Interface I must not be implemented	156		
Classes that extend C can only be referenced by S		2	1
Method M must not be overridden	26		
<i>Object instantiation</i>			
Class C must not be instantiated, except for S	38	3	
<i>Total (880)</i>	838	18	24

of dependency (generalization, realization, method calls, field access etc) among external clients and classes within INTERNAL package is not allowed. 27% (226 of 838) of the rules for Eclipse, 72% (13 of 18) of the rules for BeeFS and 96% (23 of 24) of the rules for e-Pol fall into the general restriction category. Table 2 further breaks this category down into rules of the form “*must not depend on*” and “*can only depend on*”. In general, the former form was preferred. BeeFS and e-Pol developers opted for “*can only depend on*” rules because it was easier for them to describe the few allowed dependencies rather than a number of restrictions. I use the *hierarchy* category for rules that refer to restrictions on the use of the type hierarchy. For example, BeeFS developers expressed the following rule “*Classes that extend BASICHANDLER can only be referenced by class HONEYCOMB*”. 68% (574 of 838) of the rules for Eclipse fall into this category whereas a much smaller percentage of the rules in the other two projects were of this form: 11% (2 of 18) for BeeFS and 4% (1 of 24) for e-Pol. The Eclipse rules expressed referred to more specific aspects of the type hierarchy, includ-

ing restrictions on 392 classes and interfaces that must not be extended, 156 interfaces that must not be implemented and 26 methods that must not be overridden. The rules expressed for BeeFS and e-Pol restricted access to classes that are part of a given type hierarchy. For instance, e-Pol developer expressed the following rule: “*Classes that extends ACTION can only be referenced by VIEW*”.

By interviewing stakeholders I found two motivations to declare type hierarchy rules. First, an experienced Eclipse developer [Eclipse developer #1] explained that the rules of type @noextend, @noimplement and @nooverride are used to protect APIs from semantic changes, namely changes that override services. The Eclipse developers wanted to avoid changes that might affect several clients and changes that might cause misunderstandings about the functionality provided by a part of the API. Second, for the other two projects, the rules expressed related to hierarchy were about the conceptual model the developers were trying to maintain in the code. For instance, e-Pol classes that are responsible for managing the concept “actions” are classes that extend ACTION. One might argue that these classes could be all located in the same package and the rule could be expressed referencing the package instead of the hierarchy. However, the e-Pol developer mentioned that the package ACTIONS contains other classes that are not related to the concept of action. Hence, he opted for the rule: “*Classes that extends ACTION can only be referenced by VIEW*”. I saw similar rules for BeeFS.

The third category, *object instantiation*, refers to rules that restrict which classes can instantiate particular objects. This was the only type of rule not found across all three projects. 4.5% (38 of 838) of the Eclipse rules were of this form and 16% (3 of 18) of the BeeFS rules were of this form. An example of this form of rule can be found in the BeeFS project: “*Class REPLICATIONGROUP must not be instantiated, except for REPLICATIONGROUPFACTORY*”. In general, object instantiation rules were used by the developers to restrict the instantiation of objects to factory classes.

It is somewhat surprising the large percentage of hierarchy rules for Eclipse as much of the literature on architectural rule checking focuses on rules of the general restriction category. The analysis of the rules expressed leads us to the first observation about the use of structural architectural rules:

Observation 1. *Restricting all types of dependency is sometimes too strong for developers. In some scenarios, they allow usage, but control the extension of type hierarchy and, to a lesser degree, the instantiation of objects.*

Through the interviews with the developers and analysis of Javadoc and mailing list information, I found the developers wanted to express structural architectural rules to define the proper way of using a design entity. For example, one rule in Eclipse states: “TREEVIEWER [class] *must not be extended*”. One developer stated:

“As a past maintainer of that code, I can say that the intended way of using TREEVIEWER is to just use the class as is, without subclassing. There are many other supported ways through which its behavior can be customized. Subclassing is definitely not a supported way of using TREEVIEWER.” [Eclipse developer #2]

Eclipse rules that restricts object instantiation also reinforce this observation. For example, besides the rule “RENAMERESOURCEDESCRIPTOR *is not intended to be instantiated by clients*”, the Javadoc documentation explains the proper way of acquiring an instance of this class: “An instance of this refactoring descriptor may be obtained by calling REFACTORINGCONTRIBUTION.CREATEDESRIPTOR(). In the same way, BeeFS and e-Pol developers also described restriction rules to enforce a proper way using a design entity. For example, the two object instantiation BeeFS rules were expressed to guide developers instantiate objects through the proper factory classes.

These comments lead us to a second observation:

Observation 2. *Rules are not created to blame but to guide developers.*

One Eclipse developer’s reinforced this idea, saying:

“The rules are made for those developers who are not aware of what they are doing”. [Eclipse developer #1]

Table 4.3: Violations per project

Project	General	Hierarchy	Instantiation	Total
Eclipse	60 (45%)	70 (53%)	3 (2%)	133
BeeFS	137 (96%)	5 (4%)	-	142
e-Pol	245 (99.5%)	1 (0.5%)	-	246

4.3.2 What kinds of architectural violations occur?

Table 4.3 shows the number of violations that occurred for each system by category. Interestingly, Eclipse has the lowest ratio of violations to expressed rules of 133 : 838. The ratio of violations to expressed rules is much higher for the other two projects: 142 : 18 for BeeFS and 246 : 24 for e-Pol.

Consistent with other studies (e.g., [66], [73]) that have looked at the general restriction category of architectural rule violations, large percentages of the violations in the systems I studied are of this general form. The occurrence of these violations is not surprising for two reasons. First, the rules are often stated broadly, such as in the case of Eclipse where a rule states that no external packages are to reference an INTERNAL package. Second, as is this case with the example just given, the rules are often stated between high-level design entities, such as packages. If a class violates the rule, then all of the methods that violate the rule are often cited as violations as well. For example, continuing the same Eclipse example mentioned above, the average number of classes and interfaces within the 16 plugins we analyzed is 427, creating lots of opportunities for violations. All but one rule in BeeFS and e-Pol is of this very general form; the one specific rule in BeeFS refers to communication between two classes: *REPLICA must not depend on GROUP*. Violations of hierarchy rules occurred much more often in Eclipse than in the other two projects. Of the 70 hierarchy violations that occurred in Eclipse (see Table 4.3), 54 of the violations were because a class illegally extended another class and 16 violations were due to a class illegally implementing an interface. There were no violations for rules describing constraints on overriding methods. The higher percentage of violations of hierarchy rules for Eclipse is likely due to two factors. First, many more hierarchy rules were specified for Eclipse (574 rules) than for the other two systems (3 rules between the two systems). Second, the parts of Eclipse I analyzed contained classes that provide core functionality, such as the JFace UI toolkit, that are used by many other client plugins. Discussions with an Eclipse developer affirmed that

the plugins analyzed provide core features used by several clients. Very few, only 2% of Eclipse violations and no violations for BeeFS and e-Pol, were due to object instantiation rules. The three Eclipse violations were due to illegal instantiations of three classes: `RENAMEPLUGINPROCESSOR`, `MODELCHANGEDEVENT`, and `ASSERTIONFAILEDEXCEPTION`. I investigated these violations through discussions with developers from the Eclipse project. According to developer who committed code leading to the violation related to `RENAMEPLUGINPROCESSOR`,

“Most likely the API was not available when the PDE refactoring code was created.” [Eclipse developer #3]

I confirmed this hypothesis by investigating the history of the code. I was unable to contact the committers of the other two violations, but discussions with other developers of Eclipse indicated that the `ASSERTIONFAILEDEXCEPTION` case is acceptable as it is performed in a private method for logging purposes. Regarding to the violation related to `MODELCHANGEDEVENT` a developer said:

“[the instantiation] It is an attempt to trick the system...the framework should create these events, not this class.” [Eclipse developer #1]

The analysis of the kinds of violations that occur leads to the following observation:

Observation 3. *The most common kinds of violations to occur were those related to rules about general restrictions of access, even when those rules were not the most plentiful of the rules expressed.*

4.3.3 Which architectural violations are relevant to developers?

In previous work, others have considered an architectural violation as relevant if the violation is addressed by the development team in a later version of the system [50; 74]. I use this approach to classify the relevance of violations for two different snapshots of Eclipse: violations reported for version 3.42 and violations reported for 4.2.2. As Figure 4.2 shows, these are two points at which many violations were subsequently fixed, in the 3.5 and 4.3 releases respectively. Figure 4.2 shows the number of violations in the last

Table 4.4: Relevance of Violations

Project	Detected	Exceptions or Not addressed	Actual	Critical
Eclipse 3.4.2 – 3.5	372	238 (64%)	-	134 (36%)
Eclipse 4.2.2 – 4.3	229	226 (99%)	-	3 (1%)
BeeFS	142	101 (71%)	41	13 (9%)
e-Pol	246	167 (68%)	79	46 (19%)

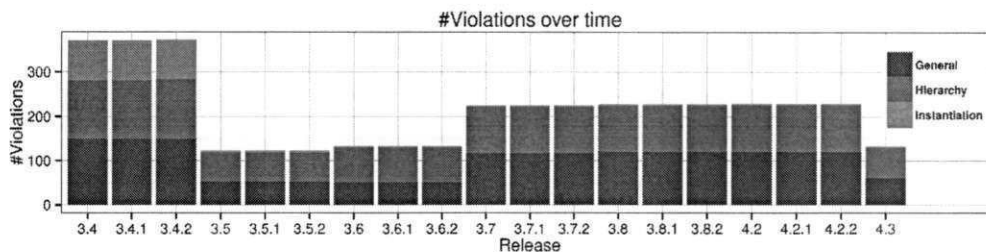


Figure 4.2: Amount of violations per Eclipse release over time

19 official releases of Eclipse. For the other two projects, BeeFS and e-Pol, I was able to investigate the relevance of violations directly with each system's respective developers.

Table 4.4 shows a classification of detected violations for the two snapshots of Eclipse development, BeeFS and e-Pol. The exceptions column in the table refers to violations the developers accept as exceptions to the rules or were not addressed by the development team (Eclipse); these are cases where the rule expressed is too general. The actual column represents violations the developers consider as relevant. I did not have sufficient access to the Eclipse developers to review each violation as to whether it was actual with them and so have left those cells blank in the table. For BeeFS and e-Pol, the critical column is the number of actual violations the developers stated compromise the structure of the code and which must be addressed as soon as possible. For Eclipse, I categorized as critical those violations that were deliberately removed by a refactoring activity.

As we can see on Table 4.4, over half of all violations are exceptions to the structural architectural rule. This leads us to the following observation:

Observation 4. *The majority of architectural violations detected are not relevant. Either*

they are included as exceptions to the rules or not addressed during software development.

By considering how violations were addressed by developers through analysis of historical data about the project (in the case of Eclipse) and through interviews with the developers (in the cases of BeeFS and e-Pol), it is possible to gain insight into the situations where violations lead to action and situations where violations are accepted. The insights I describe below are based on analysis of each violation in each system. I draw out specific examples to characterize the trends found through the analysis.

As stated above, most violations in the general restrictions category are addressed by adding exceptions to the rules to allow particular code to violate a general rule. 53 (35%) of the 151 general restriction violations were fixed for Eclipse 3.5 through the addition of exceptions to three rules. Through discussion of the rule exceptions on the developers' mailing list, I found the main reason to use exceptions in the PDE / API tool is when closely related plugins must interact to implement desired functionality. One senior developer commented as follows:

"There are often internal clients of the API, even within the same plugin, which are willing to update when the API changes and therefore can afford to extend or reference APIs in illegal ways." [Eclipse developer #4]

All of the cases I analyzed that involved adding exceptions to the rules involve commits that changes the apifilters file – a configuration file to add exceptions to the rules. I found similar actions taken in the BeeFS and e-Pol projects. However, the reasons for using exceptions to take action differed from the Eclipse project. In the BeeFS project, one half of the violations in this category (53 of 101) were considered exceptions because a logging class was invoking methods in an internal package. The developers determined the logging class required special access to the classes in the internal package. Similarly, in e-Pol, 81 of the violations were related to test classes that needed special access to the classes under test. A second case of using exceptions in e-Pol was due to coupling between MODEL and UTIL packages; this case was deemed acceptable by the developers as the classes in the MODEL package required the UTIL functionality.

When violations of the general restriction category are not considered exceptional, action was taken by developers to eliminate the unwanted dependencies. I found three cases in

Eclipse 4.3 where violating dependencies were removed due to subsequent changes to the code. The BeeFS developers classified 13 of 41 (32%) actual general restriction violations as critical requiring action to correct the violation. In describing one of these critical changes, the developer noted there was a FIXME tag in the code that needed to be addressed and that a branch to refactor the code had been created. When discussing the 46 critical (of 79 or 58%) violations in e-Pol, the developer explained he would be discussing it with the team as soon as possible as the violations were impacting separation of concerns between model, action and data objects. In fact, developers performed a major refactoring activity after the discussion. According to e-Pol developer, the development team spent 183 hours to solve the problem.

Interestingly, the majority of violations related to hierarchy (all in the two versions of Eclipse), were fixed by changing the code or the rules rather than adding exceptions. Of the 133 hierarchy violations detected in the version leading up to Eclipse 3.5, 77 (58%) were addressed by the development team. Of these 77 addressed hierarchy violations, 45 (58%) were fixed due to deliberate refactoring activities, 24 (31%) were addressed through exceptions to rules and 3 (4%) were addressed by removing a rule. I could not determine what happened to the 5 (7%) remaining violations because the classes involved were renamed or removed from the code.

As an example of refactoring, bug #193529⁵ describes a change to the code to move a method that was being accessed by subclassing to a new class to avoid violating a stated architectural rule. As an example of exceptions to the rules, exceptions were added to allow plugins to have friend status, as in “*SWT API *Listener types are allowed to extend non-API type SWTEVENTLISTENER*”. As an example of removing a rule, I found in the documentation of an involved class a statement that “*SELECTMARKERRULEACTION is allowed to be subclassed since 3.5*”.

Intentional development activities to deal with violations related to instantiation rules were only found for the version Eclipse 3.5. Of the 88 violations detected in the version leading up to Eclipse 3.5, 84 were fixed in version 3.5. 83 of these 88 violations were solved through a similar refactoring, changing all violating classes to delegate the instantiation to a factory instead of directly calling the constructor of the restricted class. The remaining

⁵https://bugs.eclipse.org/bugs/show_bug.cgi?id=193529, verified 09/11/2013.

violation was also fixed intentionally in the code. I did not find any evidence of the removal of rules or the use of exceptions for instantiation related violations.

This analysis leads to a fifth observation:

Observation 5. *Despite many violations remain unfixed, developers consider some violations to be relevant and fix them by either evolving rules or refactoring the code to reflect intended architecture.*

4.3.4 Why do developers commit violating code?

To better understand why violations might occur, I interacted with developers on each project. For Eclipse, I presented architectural violations on mailing lists relevant to the developers and asked for details on what development actions might lead to violations occurring. For BeeFS and e-Pol, I interviewed 9 and 10 developers respectively.

I gathered 137 responses from interviews and discussion snippets regarding reasons that lead developers to commit violating code. Iteratively, I coded [72] these responses to develop categories to explain the reasons. Through this process, I developed five categories: *Ease*, *Lack of Awareness*, *Time Constraints*, *Code Misplacement*, and *Copy and Paste Programming*. Table 4.5 shows the occurrences of each category in the collected data regarding all the systems. Some responses provide data that can be coded to more than one category. For example, one developer explained: “*Probably the developer didn’t know about this [rule] and was easier for him or her just access the exception instead of make some refactoring.*”

Table 4.5: Causes of architectural violations

Category	Occurrences
Ease	68
Unawareness	25
Time constrains	23
Misplaced design entity	18
Copy and paste programming	14

Ease

For all systems, the ease of implementation was the main cause of violating code. One comment from a BeeFS developer summarizes this situation:

“We were adding a feature and this feature asked for a modification in FILESYSTEM interface. However, It was not easy to come up with this modification so that we added a coupling with the concrete class.”

Through the Eclipse developers’ mailing list, I discussed a similar case in which there were 14 illegal uses of subclassing the TREEVIEWER class; this class is meant to be used through delegation. One of the Eclipse developers explained why developers were using inheritance:

“Cases where the API is not sufficiently flexible, and the only way to get the degree of customization required is via illegal API use.”[Eclipse developer #4]

In other words, subclassing made it easier to get control of some aspects of the API.

The same developer pointed out that violations “*may point to problematic API that needs more flexibility*”. However, evolving an API is not straightforward. As a BeeFS developer said, “*it is not easy to come up with an API change*” or, in Eclipse case, “*It could also be that these “illegal” subclasses are themselves exposed as API to clients, in which case it might be impossible to get rid of the subclass relationship without potentially breaking clients*”. [Eclipse developer #2]

Lack of Awareness

Similarly to the *Ease* category, developers mentioned lack of awareness about the rules as an explanation for mismatches between intended and implemented architecture. Lack of awareness can occur because rules are not automatically checked. When presented with one of the architectural violations, a developer on Eclipse said the violating code was committed before checking was performed with the API tool [Eclipse developer #5]. Before the tool, the restrictions were just comments in the Javadoc. Similarly, with BeeFS, before this study, the rules were discussed but not automatically checked. According to a BeeFS developer, one of the rules was not discussed with developers as much as others and as a result, he was

not surprised to see violations of the rule in the code. The e-Pol developers, after seeing the benefits of checking the rules suggested it would be helpful to restrict commits to only code that is in conformance with the rules.

Time Constraints

Only for the BeeFS and e-Pol projects were there responses mentioning deadlines pressure as the cause of violating code. One e-Pol developer explained that due to changes in the requirements of clients close to a deadline, the project had to change a framework used to manage graphical interface and, for this reason, developers “*were not really concerned about architectural issues. [we] needed to get things done*”. Similarly, when I presented a violation to BeeFS developer he explained: “*This is critical. By the way, this code was produced right after a deadline.*”

Misplaced Code

According to developers, in the same way that several violations may indicate changes to the architectural rules, they are also symptoms that design entities should be re-located. In other words, when analyzing violations, I found cases in which developers argue that the coupling considered illegal should exist, but one of the design entities involved are misplaced. This was particularly expressed by BeeFS and e-Pol developers. For example, 7 e-Pol developers agreed on the fact that the access to TASK class is only illegal because the class is misplaced, not because the coupling should not exist. Moreover, e-Pol developer commented: “*This is a violation because the class TASK is in the wrong package. It should be in MODEL.TASK package*”.

Copy and paste programming

Developers of the three studied systems refer to copy and paste as a cause of architectural violations. By copy and paste, they mean not only literally reproducing the code from other classes, but also writing new code following the ideas of a violating one. For example, one e-Pol developer used the following words to explain the causes of a violation: “*Because of the framework change, we were just following what an experienced developer produced*

before". Similarly, one BeeFS developer reinforced that once a project has code with violation, it is likely to have more similar violations because developers usually base their implementation in existing code. When analyzing a violation in a class that, according to the team, has a lot of architectural issues, the developer commented: "QUEENBEE has a lot of problems. I think the developer had seen old code doing the same inconsistency". In this same vein, an Eclipse developer comment summarizes explanations given into this category "It's a **common pattern** to have UI bundles reference internals of Core bundles in the same namespace". [Eclipse developer #6]

4.4 Threats to Validity

A number of the choices I made in the study could affect the results. I review these choices below.

External Validity. I gathered and analyzed data from three diverse systems. This diversity helps to improve the likelihood that the results may characterize a broad set of projects and systems but given the small sample size there is a risk that the results are specific to the situations considered. I believe the in-depth characterizations I report provide an empirical basis for formulating more specific hypotheses that can be tested on a wider range of systems. As an example, the data I present suggests type hierarchy rules are relevant to developers; a hypotheses could be formed to test across a broader set of systems related to these specific rule forms.

Internal Validity. The kinds of rules expressed by the developers for BeeFS and e-Pol might have been influenced by the fact that I have introduced the concept of conformance checking and gathered the rules through interviews with developers. I tried to mitigate the effects of the first threat by discussing only general aspects about structural rules, instead of giving concrete examples of rules. I tried to mitigate the effects of the second threat by not interfering while the developers were describing the rules. The inclusion in the study of the Eclipse case also helps to balance any effects of the introduction of conformance checking to the other two projects as the Eclipse developers had been expressing and checking architectural rules for that last five years independent of this study.

Another threat to the data I collected is the selection of developers to interview. One

might argue that interviewing seven developers from a project as large as Eclipse is far too small of a sample. By interviewing developers that have been with the project for at least four years and who, in several cases, serve on the Eclipse Architecture Council, I believe I have found developers who are highly vested in the issue of benefiting from architecture. Similarly, for BeeFS and e-Pol, I interviewed only developers with at least one year of experience with their respective projects.

Construct Validity. For BeeFS and e-Pol, I implemented the architectural rules expressed by developers as design tests. It is possible that I did not faithfully implement the architects intention. To address this possibility, I had the architects check each violation detected; this checking would likely have identified possible problems in rule expression. The architects did not report any such problems.

The observations could also be affected by relying on rules that consider only part of the architecture, that is the architectural rules do not describe the whole software architecture of the studied projects. However, I believe that I mitigate this threat due to the sample's size of architectural rules (880) collected for the three projects.

4.5 Discussion

The tools used to check architectural rules in this study limited the rules that could be expressed. I also wanted to investigate what developers wanted to express but could not. From interviews with the BeeFS and e-Pol developers and investigations of documentation in the Eclipse code, it appears it could be helpful to express patterns of intended use of design entities. For example, two of the BeeFS and e-Pol developers mentioned it would be helpful to check whether developers were following intended design patterns based on the use of design entities. In Eclipse, I found documentation in classes explaining how the classes should be used. As an example, CONTENTVIEWER includes documentation of the form:

“Implementing a concrete viewer typically involves the following steps: i) create SWT controls for viewer (in constructor) (optional), and ii) initialize SWT controls from input (inputChanged)...”

The architectural rules used, particular in the Eclipse project, focused on restricting coupling amongst entities. However, there is also the possibility of expressing mandatory cou-

pling between entities. For example, the documentation for the Eclipse class TREEVIEWER states that, “*Content providers for tree viewers must implement ITreeContentProvider interface*”. Extensions to architectural rule languages should consider this alternate form of expressing rules.

4.6 Related Work

There are two basic choices to try to ensure up-front architecture is carried through into the implementation of a system. One choice is to use development technologies that enforce constraints while implementation is occurring. An example of this approach can be found in ArchJava [61], which extends Java to express communication restrictions directly in the code, preventing architectural erosion. Alternatively, one can use an approach of periodically, and perhaps frequently, checking whether the implementation matches rules stated about the architecture. An example of this approach can be found in Software Reflexion Models that allows structural architectural models of a system to be compared against structure extracted from the implementation of the system [1].

In this study, I focus on the second approach in which architectural rules are stated and conformance of the implementation is checked against those rules. At least three existing studies consider the use of architectural rule conformance checking in practice. Murphy and Notkin report on the application of Software Reflexion Models to an experimental reengineering of Excel. This report provides one abstracted description of how structural architectural rules that capture access between modules was applied to an industrial scale system [7]. Knodel and colleagues describe their experience in regularly applying structural architectural rule conformance checking to 15 different projects over a two year period. Similar to the Excel case study, Knodel and colleagues study focused on structural architectural rules about access between modules. In the work presented in Chapter III, I also focused on structural architectural rules about access between modules but in an evolutionary perspective, providing quantitative evidences that implementation tends to diverge from the intended architecture and that few design are entities are responsible for most of the violations over the software history. In this study, I investigate the use of a broader set of architectural rules in which developers could also express rules about the type hierarchy being used and how objects in

the system were instantiated. To the best of my knowledge, there is no existing reports about this broader set of architectural rules.

Others have considered how the developers view the result of conformance checks or architectural erosion in general. Rosik and colleagues describe developers' actions when presented with the results of checking structural architectural rules [45]. The conformance checking they considered was also limited to access rules between modules. They found that developers tend to keep a number of violations unsolved, largely due to the risk of making changes to the code. The results echo this finding but the study also analyzes which kinds of violations were found to be acceptable to developers, which were critical and why the violations occurred in the first place.

Studies have also been undertaken to understand more generally how developers view architecture and why implementations deviate from architecture. Feilkas et al. [75] conducted a case study focusing on how outdated documentation is in comparison with software implementation. They observed that between 70% and 90% of deviations are caused by outdated documentation. The authors also explore why deviations between the intended architecture and code occur, finding that copy and paste programming is a typical reason of a violation. Also regarding the causes of architectural violations, Unphon and Dittrich also considered the causes of architectural violations [16]. They interviewed 15 developers to understand architecture practices in software organizations. Amongst other results, the authors found that due to lack of properly communication, developers tend to forget about the architectural decisions. Gulp et al. [76] conducted two qualitative study cases in which they investigated how developers identify and address architectural erosion. Besides lack of awareness, Gulp and colleagues also point to deadline pressure as a possible cause of architectural violation.

Putting these related works in perspective, besides a categorization of the architectural rules expressed, I not only measure the amount of architectural violations, but I also provide quantitative and qualitative data on their relevance. I also present results on how developers deal with architectural violations. Furthermore, this study confirms some of the causes of architectural violations explored by previous work and go further by providing a categorization of the reasons that lead developers to commit violating code.

4.7 Summary

For any given problem, there are typically multiple ways to express a solution to the problem in code. To try to ensure qualities related to flexibility, testability, maintainability and others, developers spend time on design. This time pays off if the implementation respects the intended architecture.

In this study, I investigated the use of architectural rules to express intended architecture and the use of checkers against implemented code to detect violations where the implementation varies from the intended architecture. The study involved two open-source systems, Eclipse and BeeFS, and one closed-source system, e-Pol. All of these systems have been under development for multiple years and involve multiple developers. By investigating the rules expressed and the violations that occurred through analysis of architectural checking reports, bug reports, code, and interviews and discussions with developers on the projects, I found that developers are concerned with checking the access to modules in the code, the use of the type hierarchy, and to a lesser extent, object instantiation. I found that developers do take action in response to violations but that the violations that persist as irrelevant tend to be related to access. The number of violations that persist are far fewer than the number of rules checked. The long use of a checker on Eclipse suggests that developers find value in the matching of implementation to intended architecture.

Chapter 5

Do Developers Discuss Design?¹

5.1 Contextualization

Open source developers share the majority of the information in a project in written form. Despite a plethora of mailing list archives, issues, commit information, and other resources associated with an open-source project, it is not usual to find a design document in project's archives. For example, I inspected *docs* folder, *wiki* pages, and main web sites of the top 90 popular projects in GitHub[77] and could not find any design documentation in 61 (68%) of them. Even considering those projects that have some documentation about their design, I could only find explicit technical artifacts (e.g. UML diagrams) in 7 (9%) projects.

Although no specific artifacts related to design can be detected in open-source projects, the other media used for communication, such as issues, commits' comments, and pull requests may include design concerns and discussions. To understand if design information is discussed and shared in these other forms in open source projects, I conducted an empirical study on 77 of the top popular projects in GitHub to provide quantitative evidence on how developers drive design discussions. Because developers usually approach structural aspects of design [14], such as communication constraints among classes, I focus this study on such aspects. In this context, I seek to investigate two questions:

- RQ1: To what extent do developers discuss design in open-source projects?

¹Parts of this chapter appeared in the Proceedings of the Working Conference on Mining Software Repositories (MSR 2014) - Mining Challenge Track.

- RQ2: Which developers discuss design?

To answer these questions, I developed the first contribution of this study – the development and evaluation of a prototype based on machine learning technique to automatically identify design discussions. Then, using this prototype, I provide quantitative evidence that, on average, 25% of the discussions in a project mention some design aspect and 26% of developers contribute to design discussions. In addition, I found that very few developers contribute to a broader range of design discussions in a project. I found a strong correlation (74%) between commits and design discussions contributions, suggesting that developers who contribute with more commits tend to discuss more about the design of the system. These two contributions may be useful for several purposes. For instance, one could use this information about which developers are involved in design discussions to drive structural refactorings to this small group of developers responsible for design. As another example, researchers can use the tool support to automatically uncover design rules.

This chapter is organized as follows. Section 5.2 describes the experimental design, including definitions about design discussions, dataset, and the procedures and measures employed. Section 5.3 shows results for the classifier and early results on analyzing design discussions. Section 5.4 discusses some important points related to the results as well as the relevance of this study. Section 5.5 briefly discusses related work, while Section 5.6 summarizes this chapter.

5.2 Study Design

In this study, I consider a discussion to be a set of comments on pull requests, commits, or issues. Because I was interested in discussions, I analyzed those pull requests, commits, and issues with more than one comment. Also, I consider a discussion to be about design if it contains at least one comment referring to some design concern. As said before, the study focuses on structural characteristics of a software design. As an example of such structural characteristic, developers usually discuss about avoiding coupling among unrelated classes or applying a specific design pattern to solve a design issue.

Hence, based on the literature in this area [30], the classification of design discussions in this work focuses on some particular topics, such as:

- coupling restrictions among design entities (e.g., “*you should not extend this class*”),
- decisions to expose or not an API (e.g., “*we should not expose this method to clients*”),
- structural refactoring (e.g., “*move this class to package presentation*”), and
- structural design patterns (e.g., “*Implement a factory to create messages for an optionally provided*”).

5.2.1 Data Set

Of the 90 projects present in the GHTorrent data set [77], I discarded 13 projects with less than 50 discussions. I chose the 77 projects with more than 50 discussions to work with a reasonable amount of data. The more discussions present in the projects, the more likely they have design discussions. Due to the fact that the interest was in the degree of design discussions, I made such decision. In addition, to simplify the analysis, I treated projects and their forks as one single project. In summary, the data set includes 77 projects and 102,122 discussions.

5.2.2 Methodology

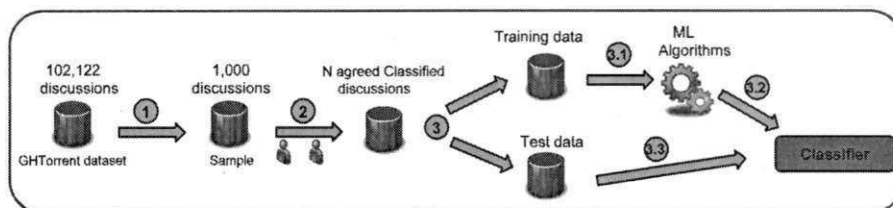


Figure 5.1: **Methodology applied to build the design discussion classifier.**

Building the Classifier

Figure 5.1 shows the steps conducted to build the design discussions classifier:

Step 1. I randomly selected 5 of the 77 projects. They are: *BitCoin*, *Akka*, *OpenFrameworks*, *Mono*, and *Twitter-Finagle*. Then, I randomly selected 200 discussions from each of these 5 projects, totaling 1,000 discussions.

Step 2. We (a collaborator in this work and I) classified the same set of 1,000 discussions separately. We tagged the discussions as design discussions or not. To avoid bias, before the classification, we did not specified or discussed any specific rules to classify the discussions. However, we stated that we would focus on structural design aspects. After the manual classification, we selected for training only the 967 discussions in which both classifications matched. 226 (23%) of these discussions refer to some design aspect, while 741 (77%) refer to other concerns related to software development.

Step 3. I used 10-fold cross validation methodology to train (steps 3.1 and 3.2) and evaluate (step 3.3) the classifier. That is, I randomly partitioned discussions into 10 equal size sets (96 discussions). Then, I used nine of these sets as training data and one of them as test data. I repeated the cross-validation process 10 times, using each one of the sets exactly once as test data. I use the mean of the 10 executions to produce an estimation of the classifier's accuracy. Using this method, I evaluated Naive Bayes and Decision Tree classifiers. I removed words from an English stoplist of common short words. As feature selectors to these classifiers, I used a combination of word frequency and bigrams. Besides the standard usage of word frequency, I also used bigrams because researchers have shown that these methods can significantly improve the results of text classification [78; 79]. For instance, in the context of this work, the bigram "exposes API" is more representative than the word "exposes" isolated or combined to other non-related word.

Answering Research Questions

After I have built confidence in the classifier, I rely on it to label all 102,122 discussions in the data set. Then, I analyzed the design discussions to answer the research questions. For the first question, I simply measured, for each project, the proportion of design discussions over all discussions. For the second question, I investigated design discussions and commits to determine:

- the ratio between the number of developers that contribute to design discussions and the number of committers in a project;
- the proportion of all design discussions in a project to which a developer has contributed, which I name *Coverage*. For instance, if a project has 10 design discussions

and a developer contributes to 5 of these discussions, the developer has 0.5 of coverage.

5.3 Results

After executing the 10-fold cross validation, the results show that Decision Tree outperforms the Naive Bayes method. The former achieved $94 \pm 1\%$ accuracy², while the latter achieved $86 \pm 3\%$. For this reason, I decided to use the Decision Tree classifier to automatically label the remaining discussions.

RQ1: To what extent do developers discuss design? Of the 102,122 discussions, the classifier labeled 25,123 (25%) as design discussions. As examples, it labeled as design concerns the following comments: “*I’d be surprised if this is the way to create RoutedActorRefs*” and “*We have the dependency issue that ActorSystem need to know about all extensions*”. Comments such as “*See code style guide. We use underscore style for variable names.*” were not labeled as design.

Figure 5.2(a) shows the proportions of design discussions per project. Following the overall proportion, $25 \pm 6\%$ of discussions within a project refer to some design aspect. As we can see by the flattened boxplot, this is a common pattern among projects. Also, this result reinforces the confidence in the classifier, once it is similar to the training data, which shows a proportion of 23% of design discussions.

RQ2: Which developers discuss design? In total, I analyzed data regarding 22,789 developers from the 77 studied projects. 8207 (36%) of these developers contribute to at least one design discussion, while 14,582 (64%) do not. The first step to answer this question was to investigate the proportion of developers that contribute to design discussions in a project. Figure 5.2(b) shows these results considering each project. A mean of $26 \pm 7\%$ of developers per project contribute to at least one comment regarding a design aspect. I inspected the projects with proportion above 30% (e.g., *Bitcoin, Django, Rails, Symfony*). These projects have a large number of committers and they are well known and established open source communities, which may explain the fact that more developers contribute to their design.

In a second step, to further investigate developers’ contribution, I measured the coverage of each developer. Figure 5.2(c) shows the coverage of developer per project. Each point

²The standard metric to evaluate classifiers, which stands for the percentage of instances correctly labeled.

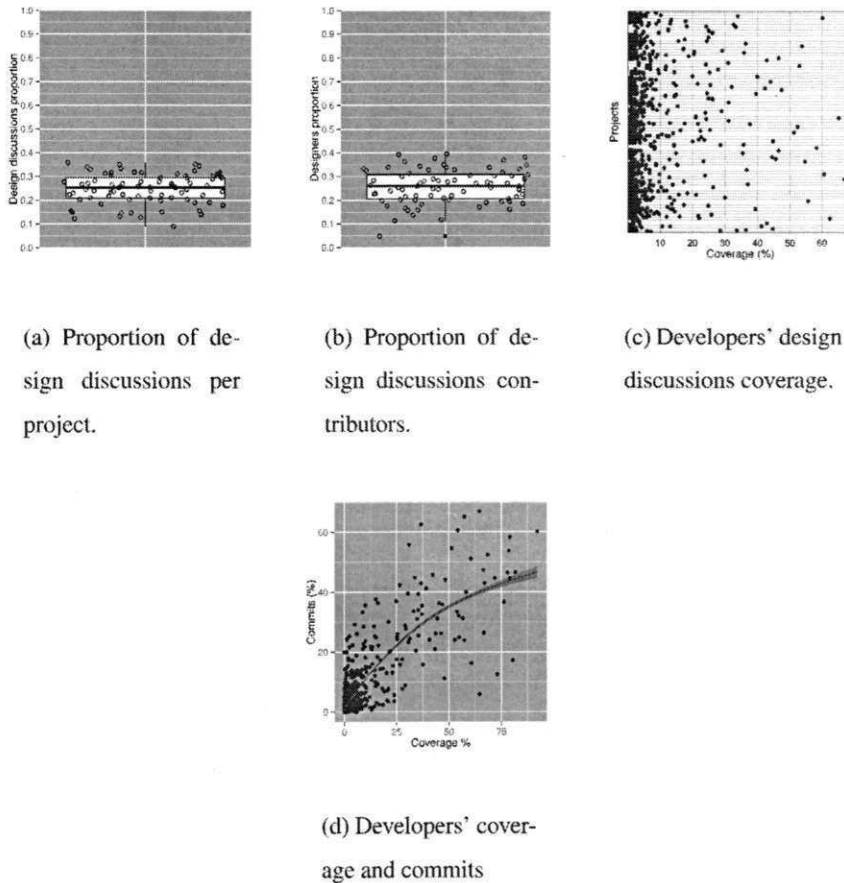


Figure 5.2: Empirical Results

in the graph represents a developer of a project in y axis. As we can see, the majority of developers contribute to less than 10% of design discussions. In fact, 99% of developers contribute to less than 15% of all design discussions in their respective projects. This results lead us to conclude that very few developers contribute to a broader range of design discussions, while most of the developers contribute to few design discussions.

Several factors might lead to the scenario in which very few developers contribute to a broad range of design discussions. This scenario suggests that these developers play a central role in their projects. I took a step forward to investigate one of the factors that might be correlated to developer's ability to discuss design in a broad range. To do so, I measured the relationship between the proportion of developers' commits and their respective coverage. Figure 5.2(d) plots the coverage against the percentage of commits of all developers studied.

The line represents the best fit for the data with 95% of confidence interval. I used Spearman's method and found a strong correlation (74%) between these two variables. As we can see, the developers with high level of Coverage are also the developers responsible for a high percentage of commits in their respective projects.

5.4 Discussion

Walking architecture. The results show that a very small number of developers have high levels of design discussions coverage. This result is aligned to a previous work that name this small set of developers as “walking architecture” [80]. The term refers to central developers who evaluate changes to code that affects design while at the same time update knowledge about design decisions. I argue that further work should invest in driving design issues to central developers. Through the classifier, researchers may use information about design discussions to build mechanisms to improve communication among developers. When developers discuss design often, they update their knowledge about the system and may achieve *Conceptual Integrity* — the uniformity of the understanding that the development team has about the software [17].

Developers' role. The high correlation between commits and design discussion coverage reveals that there is no clear separation between designers and developers role in these projects. Developers that discuss design in a broad range are the ones who most contribute to the code of the studied systems. The possible simple explanation for this scenario is the cumulative knowledge of code and design that these developers gain overtime. As time goes by, naturally these developers are responsible for the discussions, once they have a deeper knowledge about the system than the other committers.

Classifier's Performance. One possible drawback of using Decision Tree classifier is the performance issue. While the Naive Bayes 10-fold cross validation took only 9 seconds to finish, the Decision Tree validation took approximately 4 hours. This happens in the tree construction step of the algorithm, which takes a meaningful amount of time to build the branches and rules of the tree, since the combination of words and bigrams generates several tree nodes. However, it is only necessary to execute this process once, which pays-off its cost over time. For this reason, I decided to use Decision Tree classifier to label the remaining

discussions of the study.

Threats. Two threats might influence the results of the classifier. First, I trained the classifier with approximately 1% of all discussions analyzed. Second, only two researchers manually classified the discussions. Ideally, it would be better to have a broader range of researchers and practitioners classifying more discussions. However, I believe that we achieved a reasonable and reliable amount of training data. In addition, because the focus is on structural properties of design, the classifier may have missed discussions about other aspects related to design, such as dynamic and deployment concerns.

5.5 Related Work

To the best of my knowledge, this is the first work that quantitatively raises knowledge about design discussions in open-source projects and their distribution among developers. However, other researchers have investigated how developers deal with design and architecture concerns. Lange and Chaudron [81] interviewed 80 architects and observed that 66% of them employ UML diagrams to perform design activities. Cherubini et al. identified that developers usually externalize design decisions in temporary drawings that are lost over time [82]. Unphon and Dittrich conducted 15 interviews to qualitatively understand how developers drive architecture and design concerns in software companies [80]. Two of their results are closely related to this work. First, they observed the “walking architecture” phenomenon, whose existence seems to be empirically supported by the data I analyzed. Second, they observed that design/architecture documentation might not be used during software development due to the usage of other media. In this work, the data support that, for open-source projects, such media can be discussions in issues, pull requests, and commits. This last result is in conformance with Guzzi et al., which found that developers’ mailing list is not the main player in OSS project communication, as it also includes other channels such as the issue repository [83].

5.6 Summary

Developers need to maintain, verify, and discuss design during software development. In this chapter, I presented quantitative results indicating that developers address design through discussions in commits, issues and pull requests. I first built an automated classifier that employs machine learning to label discussions as design or not. I evaluated such classifier using 10-fold cross validation, achieving $94 \pm 1\%$ of accuracy. Then, using the classifier, I automatically labeled 102,122 discussions. The main observations about these discussions are: i) 25% of discussions in a project are about design; ii) 26% of developers contribute at least to one design discussion; iii) few developers contribute to a broad range of design discussions. In fact, 99% of developers contribute to less than 15% of design discussions; and iv) the very few developers who contribute to a broad range of design discussions are also the top committers in a project (correlation 74%).

Chapter 6

Conclusions

This thesis deals with the lack of knowledge about architectural erosion problem. In order to do so, I conducted three studies: i) a longitudinal and exploratory study on the nature of architectural violations (Chapter 3), ii) a study on the causes and relevance of architectural violations (Chapter 4), and iii) a study on design discussions in open-source projects (Chapter 5). This chapter summarizes my contributions to a body of knowledge of architectural erosion and includes future work that can be performed to increase even more this body of knowledge in order to help future researchers and practitioners in this area.

6.1 Contributions

In summary, the main results described in this thesis are:

Architectural erosion. One of the main contributions of this thesis is to approach the architectural erosion problem in a quantitative and evolutionary perspective. I have defined a metric (architectural debt) to capture the notion of architectural erosion taking into account the time dimension of software lifecycle.

Violations' location. Through an empirical study, this thesis shows that violations tend to be concentrated in a few design entities. For the studied projects, the top ten classes with more violations concentrate more than 40% of the violations.

Violations' lifecycle. The results also indicate that a meaningful amount of violations tend to be intermittent, which means that they are fixed and reappear in future versions of the software.

Architectural Rules. I also have shown that developers not only are concerned to general dependencies among entities, but they also express rules to control hierarchy and instantiation of objects.

Architectural violations relevance. Most of the violations are not relevant. However, I have shown that developers do perform architectural conformance checking and, more importantly, they consider some violations important to be fixed and perform refactorings to achieve this.

Causes of Architectural violations. Among other reasons, this thesis shows that violations are due to unawareness, time constraints, misplaced design entities, copy and paste programming, and the difficulties involved in following the architectural rules.

Design Discussions. An initial investigation on the presence of design discussions in open-source projects and the contributions of developers in such discussions. I found that on average 25% of the discussions in a project mention some design aspect. Moreover, 26% of the developers in a project contribute to design discussions. However, very few developers contribute to a broader range of design discussions. These few developers are also the top committers of the project.

These results contribute to provide a foundation to extend research into architectural conformance checking and provide a basis for more specific hypotheses about architectural rule expressibility and checking to be considered in future empirical studies.

6.2 Future Work

As future work, I intend to use the initial body of knowledge produced in this work to improve architectural maintenance and evolution tasks. For example, future work in this area includes the automated identification of critical cores. This will enable developers to focus

their efforts in the parts of the system that contains more violations. This could increase the number of fixed violations per maintenance activity.

Many of the violations detected for a system are either considered exceptions to a rule or are considered irrelevant by the developers in the sense that action is not taken to remove the violation. The use of exceptions to architectural rules suggests that 1) the language used to express rules may be insufficient in some ways, 2) the rules are for the majority of the cases but not all, 3) there is not sufficient time or motivation to fix a violation or 4) the architecture has changed and yet the rules have not evolved. From interviews with the developers, we did hear that the architectural rules are more of a guide than an absolute lending weight to the second suggested reason above. Further study should investigate which of the other cases might be valid reasons. If architectural rules need to be evolved, it might be that automated support to suggest when a rule is no longer valid might help with the use of checkers in practice. Similarly, automation to help detect when exceptions should be applied, perhaps through recognizing patterns of exceptions, might help ensure architectural rules can be stated simply and yet violations that are reported are ones on which action should be taken. Automation to suggest refactorings to bring code in-line with architectural rules might also be beneficial (e.g., [29]).

Another research track is to perform statistical studies in order to correlate the amount of introduced and solved violations with a number of other variables. For example, we are interested in finding out whether peaks of solved violations cause a positive impact in metrics such as coupling, cohesion, number of fixed bugs, and instability.

In this work, I did not organize design discussions in categories. As a main future work, I intend to achieve this. As we could observe, the subject of design discussions varies. For instance, some discussions are related to constraints involving classes and interfaces usage, while others to the suitability of design patterns to solve particular design issues. After this categorization, I intend to identify which design aspects attracts more attention from developers. This will require to analyze the distribution of developers per design discussion and identify the topic of such discussions. In a nutshell, I believe that such outcomes might assist in the prioritization of design issues, for example.

Bibliography

- [1] G. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: bridging the gap between source and high-level models," in *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pp. 18–28, ACM, 1995.
- [2] S. Duszynski, J. Knodel, and M. Lindvall, "Save: Software architecture visualization and evaluation," in *Proceedings of 13th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 323–324, IEEE, 2009.
- [3] R. Terra and M. Valente, "A dependency constraint language to manage object-oriented software architectures," *Software: Practice and Experience*, vol. 39, no. 12, pp. 1073–1094, 2009.
- [4] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," *SIGPLAN Not.*, 2005.
- [5] M. Godfrey and E. Lee, "Secrets from the monster: Extracting mozilla's software architecture," in *Proceedings of Second Symposium on Constructing Software Engineering Tools (CoSET'00)*, 2000.
- [6] I. S. <http://structure101.com/blog/2008/11/software-erosion-findbugs/>, "Last access in," Nov. 2012.
- [7] G. Murphy and D. Notkin, "Reengineering with reflexion models: A case study," *Computer*, vol. 30, no. 8, pp. 29–36, 1997.
- [8] D. Perry and A. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, pp. 40–52, 1992.
- [9] D. Budgen, *Software design*. Addison Wesley, 2003.

- [10] P. Clements, D. Garlan, R. Little, R. Nord, and J. Stafford, "Documenting software architectures: views and beyond," in *Proceedings of the 25th International Conference on Software Engineering*, pp. 740–741, IEEE, 2003.
- [11] D. Garlan and D. E. Perry, "Introduction to the special issue on software architecture," *IEEE Trans. Softw. Eng.*, 1995.
- [12] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Addison-Wesley Professional, 2003.
- [13] A. Jansen and J. Bosch, "Software architecture as a set of architectural design decisions," in *Proceedings of the 5th Working Conference on Software Architecture*, pp. 109–120, IEEE, 2005.
- [14] C. Hofmeister, R. Nord, and D. Soni, *Applied software architecture*. Addison-Wesley Professional, 2000.
- [15] M. Feilkas, D. Ratiu, and E. Jurgens, "The loss of architectural knowledge during system evolution: An industrial case study," in *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, 2009.
- [16] H. Unphon and Y. Dittrich, "Software architecture awareness in long-term software product evolution," *Journal of Systems and Software*, 2010.
- [17] F. Brooks, *The mythical man-month*, vol. 79. Addison-Wesley Reading, Mass, 1975.
- [18] M. Lehman, "Laws of software evolution revisited," *Software process technology*, 1996.
- [19] L. Hochstein and M. Lindvall, "Combating architectural degeneration: a survey," *Information and Software Technology*, vol. 47, no. 10, pp. 643–656, 2005.
- [20] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *Software Engineering, IEEE Transactions on*, 2001.
- [21] J. Van Gorp and J. Bosch, "Design erosion: problems and causes," *Journal of systems and software*, 2002.

- [22] D. Parnas, "Software aging," in *Proceedings of the 16th international conference on Software engineering*, pp. 279–287, IEEE Computer Society Press, 1994.
- [23] M. Dalgarno, "When good architecture goes bad," *METHODS & TOOLS*, p. 27, 2009.
- [24] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *Journal of Systems and Software*, 2012.
- [25] J. van Gurp, S. Brinkkemper, and J. Bosch, "Design preservation over subsequent releases of a software product: a case study of baan erp: Practice articles," *J. Softw. Maint. Evol.*, 2005.
- [26] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: connecting software architecture to implementation," in *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, IEEE, 2002.
- [27] J. Brunet, D. Serey, and J. Figueiredo, "Structural conformance checking with design tests: An evaluation of usability and scalability," in *27th International Conference on Software Maintenance*, pp. 143–152, IEEE, 2011.
- [28] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM Sigplan Notices*, 2004.
- [29] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha, "A recommendation system for repairing violations detected by static architecture conformance checking," *Software: Practice and Experience*, pp. 1–36, 2013.
- [30] C. Hoffmeister, *Applied software architecture*. Addison-Wesley Professional, 2000.
- [31] P. Bourque, R. Dupuis, A. Abran, J. Moore, and L. Tripp, "The Guide to the Software Engineering Body of Knowledge," *IEEE SOFTWARE*, pp. 35–44, 1999.
- [32] J. Knodel and D. Popescu, "A comparison of static architecture compliance checking approaches," in *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, IEEE Computer Society, S, vol. 12, 2007.
- [33] M. Fagan, "Design and code inspections to reduce errors in program development," *IBM Journal of Research and Development*, 1976.

- [34] D. L. Parnas and D. M. Weiss, "Active design reviews: principles and practices," in *Proceedings of the 8th international conference on Software Engineering*, 1985.
- [35] K. Beck, "Embracing change with extreme programming," *Computer*, vol. 32, no. 10, pp. 70–77, 1999.
- [36] K. Schwaber and M. Beedle, *Agile software development with Scrum*, vol. 18. Prentice Hall PTR Upper Saddle River, NJ, 2002.
- [37] M. Moriconi, X. Qian, and R. Riemenschneider, "Correct architecture refinement," *Software Engineering, IEEE Transactions on*, 1995.
- [38] C. Baldwin and K. Clark, *Design rules, Volume 1: The power of modularity*, vol. 1. MIT Press, 2000.
- [39] E. Gamma and K. Beck, "Junit," 2006.
- [40] G. Fairbanks, D. Garlan, and W. Scherlis, "Design fragments make using frameworks easier," *ACM SIGPLAN Notices*, 2006.
- [41] M. Baker, "The mozilla project: past and future," *Open sources*, vol. 2, pp. 3–20, 2005.
- [42] J. Bosch and M. Svahnberg, "Characterizing evolution in product line architectures," 1999.
- [43] A. Ant, "The apache ant project," 2010.
- [44] D. Bovet and M. Cesati, *Understanding the Linux kernel*. O'Reilly Media, Incorporated, 2005.
- [45] J. Rosik, A. Le Gear, J. Buckley, M. Babar, and D. Connolly, "Assessing architectural drift in commercial software development: a case study," *Software: Practice and Experience*, 2011.
- [46] J. Brunet, R. A. Bittencourt, D. Serey, and J. Figueiredo, "On the evolutionary nature of architectural violations," in *Proceedings of Working Conference on Reverse Engineering (WCRE)*, 2012.

- [47] M. Feilkas, D. Ratiu, and E. Jurgens, "The loss of architectural knowledge during system evolution: An industrial case study," in *Proceedings of ICPC'09*, IEEE, 2009.
- [48] G. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: Bridging the gap between source and high-level models," *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 4, pp. 18–28, 1995.
- [49] R. A. Bittencourt, *Enabling Static Architecture Conformance Checking of Evolving Software*. PhD thesis, 2012.
- [50] S. Kim and M. Ernst, "Which warnings should i fix first?," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ACM, 2007.
- [51] J. Araujo, S. Souza, and M. Valente, "Study on the relevance of the warnings reported by java bug-finding tools," *Software, IET*, 2011.
- [52] A. Hunt and D. Thomas, *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.
- [53] M. Godfrey and Q. Tu, "Evolution in open source software: a case study," in *Software Maintenance, International Conference on*, 2000.
- [54] M. Lehman, D. Perry, and J. Ramil, "Implications of evolution metrics on software maintenance," in *Software Maintenance, International Conference on*, 1998.
- [55] W. Turski, "Reference model for smooth growth of software systems," *IEEE Transactions on Software Engineering*, 1996.
- [56] H. Gall, M. Jazayeri, R. Klosch, and G. Trausmuth, "Software evolution observations based on product release history," in *Software Maintenance, International Conference on*, IEEE, 1997.
- [57] S. Hassaine, Y. Guéhéneuc, S. Hamel, and G. Antoniol, "ADvISE: Architectural decay in software evolution," in *Proceeding of the 16th European Conference on Software Maintenance and Reengineering*, 2012.

- [58] M. Wermelinger, Y. Yu, A. Lozano, and A. Capiluppi, "Assessing architectural evolution: a case study," *Empirical Software Engineering*, pp. 1–44, 2011.
- [59] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Elements of object-oriented software architecture." *Addison-Wesley*, vol. 9, p. 12, 1994.
- [60] P. Wolfgang, *Design patterns for object-oriented software development*. Reading, Mass.: Addison-Wesley, 1994.
- [61] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: connecting software architecture to implementation," in *Proceedings of the 24th International Conference on Software Engineering*, pp. 187–197, ACM New York, NY, USA, 2002.
- [62] R. Terra and M. T. Valente, "A dependency constraint language to manage object-oriented software architectures," *Softw. Pract. Exper.*, vol. 39, no. 12, pp. 1073–1094, 2009.
- [63] D. Hou and H. J. Hoover, "Using scl to specify and check design intent in source code," *IEEE Transactions on Soft. Engineering*, pp. 404–423, 2006.
- [64] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini, "Defining and continuous checking of structural program dependencies," in *Proc. of Int. Conf. on Soft. engineering*, pp. 391–400, 2008.
- [65] J. Brunet, D. Serey, and J. Figueiredo, "Structural conformance checking with design tests: An evaluation of usability and scalability," in *Proc. of Int. Conf. on Soft. Maintenance*, pp. 143–152, 2011.
- [66] J. Knodel, D. Muthig, U. Haury, and G. Meier, "Architecture compliance checking-experiences from successful technology transfer to industry," in *Proc. of European Conf. on Soft. Maintenance and Reengineering*, pp. 43–52, 2008.
- [67] T. E. Pereira, A. Soares, J. Silva, and F. Brasileiro, "Beefs: A cheaper and naturally scalable distributed file system for corporate environments," tech. rep., Technical report, LSD-UFCG, 2010.

- [68] J. W. Creswell, *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage, 2009.
- [69] F. Shull, J. Singer, and D. I. Sjöberg, *Guide to advanced empirical software engineering*. Springer, 2008.
- [70] J. Brunet, D. Guerrero, and J. Figueiredo, “Design tests: An approach to programmatically check your code against design rules,” in *Proc. of Int. Conf. on Soft. Engineering - NIER*, pp. 255–258, 2009.
- [71] L. Schamber, M. B. Eisenberg, and M. S. Nilan, “A re-examination of relevance: toward a dynamic, situational definition,” *Information processing & management*, pp. 755–776, 1990.
- [72] R. L. Gorden, “Basic interviewing skills,” 2006.
- [73] J. Brunet, R. Bittencourt, D. Serey, and J. Figueiredo, “On the evolutionary nature of architectural violations,” in *Proc. of Working Conf. on Reverse Engineering*, pp. 257–266, 2012.
- [74] J. E. M. Araújo, S. Souza, and M. T. Valente, “Study on the relevance of the warnings reported by java bug-finding tools,” *IET software*, pp. 366–374, 2011.
- [75] M. Feilkas, D. Ratiu, and E. Jürgens, “The loss of architectural knowledge during system evolution: An industrial case study,” in *Proc. of Int. Conf. on Program Comprehension*, pp. 188–197, 2009.
- [76] J. van Gurp, S. Brinkkemper, and J. Bosch, “Design preservation over subsequent releases of a software product: a case study of baan erp,” *Journal of Soft. Maintenance and Evolution: Research and Practice*, pp. 277–306, 2005.
- [77] G. Gousios, “The GHTorrent dataset and tool suite,” in *Proc. of Working Conference on Mining Soft. Repositories, MSR’13*, pp. 233–236, 2013.
- [78] J. Fürnkranz, “A study using n-gram features for text categorization,” *Austrian Research Institute for Artificial Intelligence*, pp. 1–10, 1998.

-
- [79] W. B. Cavnar, J. M. Trenkle, *et al.*, “N-gram-based text categorization,” *Ann Arbor MI*, pp. 161–175, 1994.
- [80] H. Unphon and Y. Dittrich, “Software architecture awareness in long-term software product evolution,” *Journal of Systems and Soft.*, pp. 2211–2226, 2010.
- [81] C. Lange, M. R. V. Chaudron, and J. Muskens, “In practice: Uml software architecture and design description,” *IEEE Soft.*, pp. 40–46, 2006.
- [82] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko, “Let’s go to the whiteboard: how and why software developers use drawings,” in *Proc. of SIGCHI conference on Human factors in computing systems*, pp. 557–566, 2007.
- [83] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. v. Deursen, “Communication in open source software development mailing lists,” in *Proc. of International Workshop on Mining Software Repositories*, pp. 277–286, 2013.