UM SISTEMA DE RECOMENDAÇÃO PARA REPARAÇÃO DE EROSÃO ARQUITETURAL DE SOFTWARE

RICARDO TERRA NUNES BUENO VILLELA

UM SISTEMA DE RECOMENDAÇÃO PARA REPARAÇÃO DE EROSÃO ARQUITETURAL DE SOFTWARE

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Orientador: Roberto da Silva Bigonha
Coorientador: Marco Túlio de Oliveira Valente

Belo Horizonte
Setembro de 2013

RICARDO TERRA NUNES BUENO VILLELA

A RECOMMENDATION SYSTEM FOR REPAIRING SOFTWARE ARCHITECTURE EROSION

Thesis presented to the Graduate Program in Computer Science of the Universidade Federal de Minas Gerais. Departamento de Ciência da Computação. in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: Roberto da Silva Bigonha
Co-Advisor: Marco Túlio de Oliveira Valente

Belo Horizonte
September 2013

Villela, Ricardo Terra Nunes Bueno

V735r A recommendation system for repairing software architecture erosion / Ricardo Terra Nunes Bueno Villela. — Belo Horizonte, 2013
xxv, 104 f. : il. ; 29cm

Tese (doutorado) — Universidade Federal de Minas Gerais. Departamento de Ciência da Computação.

Orientador: Roberto da Silva Bigonha
Coorientador: Marco Túlio de Oliveira Valente

1. Computação – Teses. 2. Engenharia de software – Teses. 3. Software – Arquitetura – Teses. 4. Sistemas de recomendação – Teses. I. Orientador. II. Coorientador. III. Título.

CDU 519.6*32 (043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

A recommendation system for repairing software architecture erosion

RICARDO TERRA NUNES BUENO VILLELA

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. ROBERTO DA SILVA BIGONHA - Orientador
Departamento de Ciência da Computação - UFMG

PROF. MARCO TÚLIO DE OLIVEIRA VALENTE - Coorientador
Departamento de Ciência da Computação - UFMG

PROF. DALTON DARIO SEREY GUERRERO
Departamento de Sistemas e Computação - UFCG

PROF. EDUARDO MAGNO LAGES FIGUEIREDO
Departamento de Ciência da Computação - UFMG

PROF. PAULO HENRIQUE MONTEIRO BORBA
Departamento de Informática - UFPE

PROF. KRZYSZTOF CZARNECKI
ECE - University of Waterloo

Belo Horizonte, 27 de setembro de 2013.

This thesis is dedicated to Ryta, who has always been supporting me.

Acknowledgments

This work would not have been possible without the support of many people.

I thank **God** to provide me with intellect and persistence to reach a Ph.D. degree.

I thank my whole **family**—especially Maurício, Vânia, Lorena, Jéssica e Patrícia—for having always supported me.

I thank my **girlfriend** Carol for always being by my side.

I thank my **advisors** R. S. Bigonha and M. T. Valente for being abundantly helpful and offered invaluable assistance, support, and guidance.

I thank my **co-advisor** K. Czarnecki for giving me the opportunity to work under his supervision for a year.

I thank the **welcomer** L. Passos for receiving me as a brother in his home in Waterloo. As well as T. Bartolomei and C. Vargas for taking care of me like a brother.

I thank the **encouragers** L. Ishitani and M. A. J. Song for writing the recommendation letters that contributed to my acceptance into this Ph.D. program.

I thank the **program officers** R. Vieira, S. Borges, and S. L. Santos for the attention and patience over these years.

I thank the **software architects** G. Dafé (BrTCom) and R. Garcia (Geplanes) for the valuable collaboration in the case studies.

I thank the **members of the research groups** (e.g., LabSoft, GSD, and ASERG)—especially C. Couto and J. Brunet—for pursuing collaborative research.

I'd like to express my gratitude to the **members of my thesis defense**—D. Serey (UFCG), E. Figueiredo (UFMG), K. Czarnecki (UWaterloo), and P. Borba (UFPE).

It has been slightly over four years, two different countries, and a bunch of conferences. Therefore, like the approach proposed in this thesis, I cannot claim that this list of acknowledgments is complete, which is far ahead of my objective.

"The formulation of a problem is often more essential than its solution, which may be merely a matter of mathematical or experimental skill."

(Einstein, Albert)

Resumo

Erosão arquitetural é um problema recorrente enfrentado por arquitetos de software. Embora um grande número de técnicas de conformidade arquitetural tenham sido propostas para detectar violações arquiteturais (por exemplo, modelos de reflexão, matrizes de dependência estrutural, linguagens de consulta em código fonte, linguagens de restrição, linguagens de descrição arquitetural e testes de desenho), a tarefa de reparação de violações arquiteturais ainda não tem o suporte adequado. Como consequência direta, desenvolvedores costumam corrigir violações arquiteturais de forma *ad hoc* e ainda sem o auxílio de ferramentas no nível arquitetural. Isso pode fazer com que desenvolvedores gastem um tempo considerável para descobrir como reparar as violações arquiteturais ou mesmo introduzam novas violações ao reparar violações existentes.

Diante disso, esta tese propõe um sistema de recomendação de reparação arquitetural que provê diretrizes de correção para desenvolvedores e arquitetos encarregados de reverter processos de erosão arquitetural. Formalizou-se um catálogo de recomendações de reparação para corrigir violações detectadas por abordagens estáticas de conformidade arquitetural; elaborou-se uma heurística para determinar o módulo correto para entidades de código-fonte; desenvolveu-se uma ferramenta – chamada `ArchFix` – que automatiza as recomendações propostas; e avaliou-se a aplicabilidade da abordagem em um sistema de médio porte e em um sistema de grande porte. Para o primeiro sistema – um sistema de gestão estratégica de 21 KLOC – a abordagem proposta indicou reparações corretas para 31 das 41 violações detectadas pelo processo de conformidade arquitetural. Para o segundo sistema – um sistema de atendimento a clientes de 728 KLOC utilizado por uma empresa de telecomunicações nacional – a abordagem proposta sugeriu recomendações corretas de acordo com o arquiteto do sistema para 632 das 787 violações. Além disso, os arquitetos apontaram 80% dessas recomendações como tendo complexidade moderada ou alta.

Palavras-chave: Erosão arquitetural, reparação arquitetural, sistema de recomendação.

Abstract

Architectural erosion is a recurrent problem faced by software architects. Although several architecture conformance techniques have been proposed to detect architectural violations (e.g., reflexion models, dependency structure matrices, source code query languages, constraint languages, architecture description languages, and design tests), less research effort has been dedicated to the task of repairing such violations. As a consequence, developers usually perform the repairing task in ad hoc ways, without tool support at the architectural level. This fact may lead developers to spend a long time to discover how to repair the architectural violations and even to introduce new violations when repairing one.

In view of such circumstances, this thesis proposes an architectural repair recommendation system that provides fixing guidelines for developers and maintainers when tackling architectural erosion. We have formalized a catalog of repairing recommendations to repair violations raised by static architecture conformance checking approaches; we have elaborated a suitable module heuristic to determine the correct module for source code entities; we have designed a tool—called `ArchFix`—that triggers the proposed recommendations; and we have evaluated the application of our approach in two industrial-strength systems. For the first system—a 21 KLOC open-source strategic management system—our approach indicated correct repairing recommendations for 31 out of 41 violations detected as the result of an architecture conformance process. For the second system—a 728 KLOC customer care system used by a major telecommunication company—our approach triggered correct recommendations for 632 out of 787 violations, as asserted by the system's architect. Moreover, the architects scored 80% of these recommendations as having moderate or major complexity.

Keywords: Architectural erosion, architectural repair, recommendation system.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

In this chapter, we state the problem and present our motivation (Section 1.1). We then provide an overview of our proposed approach (Section 1.2). Finally, we present the outline of the thesis (Section 1.3) and our publications (Section 1.4).

1.1 Problem

Software architecture erosion is one of the most evident manifestations of software aging [76, 78, 25]. The phenomenon designates the progressive gap normally observed between two architectures: the *intended architecture* defined during the architectural design phase and the *concrete architecture* defined by the current implementation of the software system [111, 48, 55]. Causes of architectural erosion include deadline pressures, conflicting requirements, miscommunication, developers' unawareness, and the lack of an explicit correspondence between architectural and programming language abstractions. Regardless the causes, when the erosion is neglected over long periods, it may reduce the concrete architecture to a small set of strongly-coupled and weakly-cohesive components, whose maintenance and evolution become increasingly more difficult and costly [88, 25].

To tackle the erosion process, the first task is *to check whether the concrete architecture conforms to the intended one* [50, 77, 26, 25]. More specifically, the goal of an architecture conformance process is to reveal the implementation decisions that denote architectural violations, i.e., the concrete statements, expressions, or declarations in the source code that do not match the constraints imposed by the intended architecture. For this purpose, several architecture conformance techniques have been proposed, including reflexion models [68], dependency structures matrices [96], source

code query languages [23], constraint languages [102, 44, 27], architecture description languages [2], and design tests [16].

After the conformance phase, the next task is *to replace the detected violations with implementation decisions consistent with the intended architecture*. However, this reengineering effort is usually a non-trivial and time-consuming task because software erosion is mostly a silent process that accumulates over years. For example, Knodel et al. described their experience of applying an architecture conformance process to a product line in the domain of portable measurement devices [47]. As a result, they identified almost 5,000 architectural divergences in three products of this product line. In a previous work [102], we described our own experience in applying conformance techniques to a human-resource management system. In this process, we were able to detect more than 2,200 architectural violations. As a last example, Sarkar et al. reported their experience in remodularizing a large banking application [88]. Reconstructing the original architecture of this system demanded 2,100 person-days just for coding and testing.

However, despite of its relevance and in contrast to the variety of techniques available for architecture conformance, the task of fixing architectural violations is usually performed in an ad hoc way. The only employed tools are often the automatic refactorings provided by today's IDEs or simple program analysis tools, such as those that extract function-call information [88]. However, their application is too generic in most scenarios. As a first example, assume that a developer has created an object of type `Product` in a module where the creation of this object is not allowed. To fix this violation, one potential solution consists in applying a *Replace Constructor with a Factory Method* refactoring [32]. However, most developers do not have a complete understanding of the system and, therefore, they may not know about the factory or the exact factory method to call in this situation. As a second example, assume that a particular database query is performed outside a Data Access Object (DAO) [33]. To fix this violation, developers may apply the *Extract Method* and then the *Move Method* refactoring [32]. However, they may require considerable time to determine the particular DAO the query must be moved to. Therefore, *we claim that the task of repairing architectural violations should not be addressed in ad hoc ways because architecture repair is as important as architecture conformance checking in order to reverse software architecture erosion.*

1.2 An Overview of the Proposed Approach

As stated in the previous section, the task of repairing architectural violations is non-trivial, time-consuming, and usually performed in ad hoc ways without adequate tool support at the architecture level. In order to address the lack of support for removing architectural violations, this thesis describes a solution based on recommendation system principles that provides repairing guidelines for developers and maintainers when fixing violations in the module architecture view of object-oriented systems.

As illustrated in Figure 1.1, considering a set of architectural violations raised by a static architecture conformance tool, the proposed recommendation system—called `ArchFix`—provides repairing recommendations to guide the process of fixing each detected violation. For illustration purposes, we reconsider the motivating examples of the previous section. In the first example, when a developer creates an object of type `Product` in a module where the creation of this object is not allowed, our system not only suggests the application of a *Replace Constructor by a Factory Method*, but it also complements this information with the name of the Factory method that should be called in this particular context. In the second example, when a developer mistakenly implements a database operation outside a DAO, our system not only suggests the application of the *Extract Method* and *Move Method* refactorings, but it also indicates the most suitable target class where the extracted method should be moved to.

Figure 1.1: Proposed architectural repair recommendation system

First, we have formalized a catalog of repairing recommendations for addressing architectural violations, including violations due to divergences and absences. This set of 32 recommendations emerged after an in-depth investigation of possible fixes for more than 2,200 architectural violations we detected in a previously evaluated system [102]. Second, we have elaborated a suitable module heuristic to determine the correct module for source code entities based on their structural similarity. Third, we have designed a tool—called `ArchFix`—that implements our approach and hence provides recommendations for architectural violations in Java systems. Fourth, we have evaluated the application of our approach in two industrial-strength systems. For the first system—a 21 KLOC open-source strategic management system—our approach indicated correct repairing recommendations for 75% of the detected violations. For the second system—a 728 KLOC customer care system used by a major telecommunication

company—our approach triggered correct recommendations for 80% of the violations, as endorsed by the architects. Furthermore, the architects scored 80% of these recommendations as having *moderate* or *major* complexity [106, 98, 99, 104, 105, 77].

Such a solution may represent a promising approach, i.e., it may represent a significant improvement to the state of the practice in architecture repair. On the other hand, it is worthwhile to mention that, by definition, an approach based on recommendations does *not* have the ambition to provide a fully sound and complete solution to remove architectural violations, which is certainly a task ahead of the state of the art in reengineering tools. In fact, even a bug-free implementation for typical refactorings, i.e., refactorings whose scope are limited to a few classes, has proved to be a complex task [95, 93].

1.3 Outline of the Thesis

We organized the remainder of this work as follows:

- **Chapter 2** covers background work related to our research, such as architectural models, architecture conformance approaches, refactoring techniques, remodularization methods, and recommendation systems;

- **Chapter 3** provides a specification of the proposed architectural repair recommendation system, including the description of a subset of recommendations, underlying algorithms, and the design of the `ArchFix` tool;

- **Chapter 4** introduces the Qualitas.*class* Corpus and provides empirical evidences supporting the implementation decisions related to our suitable module heuristic. This heuristic is used by our approach to trigger recommendations that may involve moving methods or classes;

- **Chapter 5** evaluates our approach by presenting and discussing results on applying the repairing recommendations in two real-world systems; and

- **Chapter 6** presents the final considerations of this thesis, including the contributions, limitations, and future work.

1.4 Publications

This thesis generated the following publications and therefore contains material from them:

- Reference #106: Terra, R., Valente, M. T., Czarnecki, K., and Bigonha, R. S. (2013c). A recommendation system for repairing violations detected by static architecture conformance checking. *Software: Practice and Experience*, pages 1–28.

- Reference #98: Terra, R., Brunet, J., Miranda, L. F., Valente, M. T., Serey, D., Castilho, D., and Bigonha, R. S. (2013a). Measuring the structural similarity between source code entities. In *25th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 753–758.

- Reference #99: Terra, R., Miranda, L. F., Valente, M. T., and Bigonha, R. S. (2013b). Qualitas.class Corpus: A compiled version of the Qualitas Corpus. *Software Engineering Notes*, 38(5):1–4.

- Reference #104: Terra, R., Valente, M. T., Bigonha, R. S., and Czarnecki, K. (2012a). DCLfix: A recommendation system for repairing architectural violations. In *III Brazilian Conference on Software: Theory and Practice (CBSoft), Tools Session*, pages 1–6. (2^{nd} best tool)

- Reference #105: Terra, R., Valente, M. T., Czarnecki, K., and Bigonha, R. S. (2012b). Recommending refactorings to reverse software architecture erosion. In *16th European Conference on Software Maintenance and Reengineering (CSMR), Early Research Achievements Track*, pages 335–340.

- Reference #77: Passos, L., Terra, R., Diniz, R., Valente, M. T., and Mendonça, N. (2010). Static architecture-conformance checking: An illustrative overview. *IEEE Software*, 27(5):82–89.

Chapter 2

Background

In this chapter, we discuss background work related to our thesis. First, Section 2.1 describes different *Architectural Models*, including the one this thesis is based on. Second, Section 2.2 introduces *Architecture Conformance* approaches, which are related to our work in terms of providing the input of our approach. Third, Section 2.3 presents *Refactoring* techniques, since our approach also relies on primitive refactorings to repair architectural violations. Fourth, Section 2.4 discusses *Remodularization* approaches because they represent potential alternatives to architecture degradation. Fifth, Section 2.5 provides an overview on *Recommendation Systems*, since our approach is based on recommendation principles. Finally, Section 2.6 concludes this chapter with a general discussion.

2.1 Architectural Models

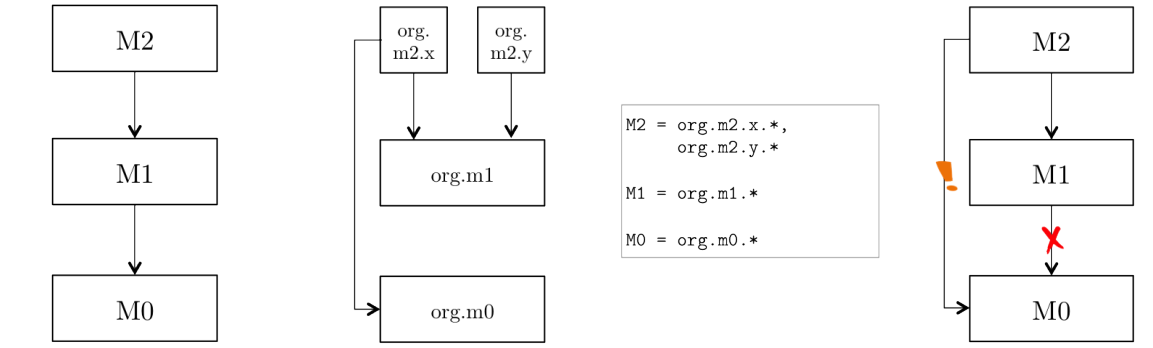
Kruchten defines software architecture using the following five concurrent views: *logical view* (which defines the conceptual decomposition of the system), *process view* (which partitions the software in independent tasks), *physical view* (which maps software tasks to hardware elements), *development view* (which partitions a system into physical modules or subsystems), and *use case view* (which defines the functions provided to the users by listing use cases) [53]. Each view addresses concerns of interest to different stakeholders. Particularly, our work is centered on the *development view* (a.k.a. *module view*), which describes the software's static organization in its development environment. It concerns low-level design decisions, patterns, and best practices. In practice, there are studies that refer to the *development view* as a high-level software model [68]. Similarly, there are studies that ascribe the violations tackled by our approach as due to *design erosion* rather than *architectural erosion* [111, 16].

2.2 Architecture Conformance

Over the past decade, several techniques have been proposed to deal with the architecture erosion problem [77, 79]. Since our approach works on the violations detected as the result of a static architecture conformance process, an overview on architecture conformance techniques is relevant. This section therefore presents an overview of the following state-of-the-art techniques:

Reflexion Models (RM): The reflexion model technique was initially proposed by Murphy et al. [68, 69]. The main idea is to compare two models—representing a high-level and a low-level view of the target system [68, 69]. As a result, the technique highlights the detected differences in terms of *divergences* and *absences* in what the authors called a *reflexion model*. Basically, a divergence occurs when a relation not prescribed by the high-level model exists in the source code, whereas an absence occurs when a relation prescribed by the high-level model does not exist in the source code. There are commercial tools based on reflexion model principles [48, 55, 50, 49], besides several works extending the original reflexion model to support, for example, continuous conformance checking [11, 10], hierarchical structures [52], behavioral design [1], and software variants [51, 34].

As an illustrative example, we rely on Fraunhofer IESE's `SAVE` (Software Architecture Visualization and Evaluation) tool.[1] The architect must first build a high-level model that captures the intended architecture of the system (Figure 2.1a). Such model includes the main components of the system and the relations between them (e.g., calls, creates, and inherits). In this example, we assume a strictly layered system with modules $\mathtt{M}_2$, $\mathtt{M}_1$, and $\mathtt{M}_0$ (where $\mathtt{M}_0$ represents the module in the lowest level of the hierarchy). Hence, in this system, only $\mathtt{M}_i$ must use services provided by module $\mathtt{M}_{i-1}$, $i > 0$.



(a) High-level model (b) Source code model (c) Mapping (d) Reflexion model

Figure 2.1: RM approach

[1] http://www.fc-md.umd.edu/save

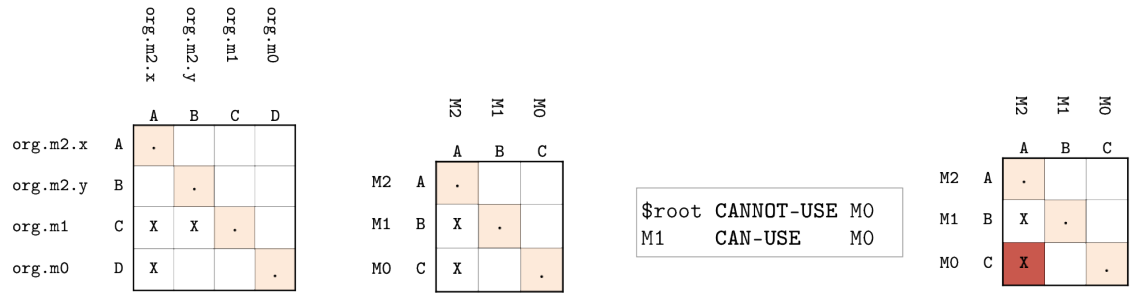
SAVE provides the automatic extraction of the source code model, i.e., the implemented architecture of the system (Figure 2.1b). Next, the architect must define a declarative mapping between the source code model and the high-level model (Figure 2.1c). In our example, module M2 contains all classes from packages org.m2.x and org.m2.y, module M1 corresponds to package org.m1, and module M0 corresponds to package org.m0. Finally, SAVE automatically computes the reflexion model (Figure 2.1d). In our example, a divergence was found because module M2 is directly using services provided by module M0 (as indicated by an exclamation mark). In addition, an absence was also found since module M1 is not using the services provided by module M0 (as indicated by a cross mark).

Dependency Structure Matrices (DSMs): The concept of DSM was originally proposed by Baldwin and Clark to demonstrate the importance of modular design principles in the hardware industry [7]. Afterwards, Sullivan et al. argued that DSM could also be used in the software industry [96]. DSMs provide a scalable view of the established dependencies among classes of a system [87, 96]. A DSM is a weighted square matrix whose both rows and columns denote classes from an object-oriented system. An x in row A and column B of a DSM denotes that class (or module) B depends on class (or module) A, i.e., B has explicit references to syntactic elements of A. Another possibility is to represent in cell (A,B) the number of references that B contains to A.

As an illustrative example, we rely on a DSM as computed by Lattix Inc's Dependency Manager (LDM) tool.[2] LDM includes a simple language to declare design rules that the target system implementation must follow. Basically, design rules have two forms: A can-use B and A cannot-use B, indicating that classes in the set A can (or cannot) depend on classes in B. Violations in design rules are automatically detected by LDM and visually represented in the extracted DSM. We assume again the aforementioned strictly layered system. Figure 2.2a illustrates the DSM automatically extracted from the source code. As can be observed, package org.m2.x depends on packages org.m1 and org.m0, and package org.m2.y depends on package org.m1. In order to improve the visualization, LDM supports grouping and renaming of packages. Figure 2.2b illustrates the extracted DSM after packages org.m2.x and org.m2.y have been grouped in a module named M2, and packages org.m1 and org.m0 have been renamed to M1 and M0, respectively.

In order to check architectural conformance, architects can define design rules. For example, Figure 2.2c shows a design rule that allows only M1 to depend on M0.

[2] http://www.lattix.com



(a) Extracted DSM (b) DSM after grouping (c) Design rules (d) Checked DSM

Figure 2.2: DSM approach

First, this rule specifies that `$root`, which denotes all types of the system, cannot access services provided by module `M0`. Next, an exception to the previous rule is defined, specifying that classes in module `M1` are able to use services of `M0`. As a result, `LDM` highlights the existing dependencies from `M2` to `M0` as potential violations (Figure 2.2d). In contrast to `SAVE`, `LDM` does not provide means to detect absences.

Source Code Query Languages (SCQLs): SCQLs are usually employed to automate a broad range of software development tasks, such as checking coding conventions, searching for bugs, computing software metrics, and detecting refactoring opportunities [23]. In this section, we present how to apply a particular SCQL—Semmle's `.QL`[3]—to find potential architectural violations. As a concrete example, De Schutter successfully used Semmle's `.QL` to implement automated architectural reviews in a Belgian electronic communications company [24].

`.QL` adopts an SQL-like syntax, which makes its query constructs familiar to most software developers. Therefore, in order to check conformance, the architect is supposed to write queries for each architectural constraint. Assuming again the aforementioned strictly layered system, the architect can write the following query to detect unauthorized accesses to `M0`:

```
1  from RefType ref, RefType m0
2  where
3      ref.fromSource()
4      and not (ref.getPackage().getName().matches("org.m0")
5          or ref.getPackage().getName().matches("org.m1"))
6      and m0.getPackage().getName().matches("org.m0")
7      and depends(ref, m0)
8  select ref as Type,
9      "Architectural Violation: " + ref.getQualifiedName()
10     +  " uses " + m0.getQualifiedName() as Violation
```

[3]http://semmle.com

In .QL queries, RefType represents any type for which references can be established in the source code. RefType has functions such as getPackage() (that returns the package where the type has been declared) and predicates such as fromSource() (that checks whether the target type is part of the current project). The previous query checks whether there is a type ref in the current project that is not part of the module M0 or M1 (lines 4-5) and that depends on a type of module M0 (lines 6-7). The query returns the type ref that contains the architectural violation and its description (lines 8-10). Despite the expressiveness of .QL, we consider SAVE and LDM more suitable to architecture conformance purposes. On the other hand, SCQLs have a broader scope and hence they can be used for many other software engineering tasks.

Constraint Languages: The rationale behind constraint languages is to provide architects with means to restrict the spectrum of structural dependencies. In this section, we describe DCL (Dependency Constraint Language), a domain-specific language that supports the definition of structural constraints between modules [102, 100, 101, 91, 103]. Nevertheless, there is a wide range of constraint languages. For instance, SCL (Structural Constraint Language) [44], its predecessor FCL (Framework Constraint Language) [45], and LogEn [27] are first-order logic-based languages for specifying structural design constraints.

DCL provides constraints to capture both divergences and absences [68, 77]. To capture divergences, DCL allows architects to specify that dependencies *only can*, *can only* or *cannot* be established by specified modules. In addition, to capture absences, architects can specify that particular dependencies *must* be present in the source code. Moreover, DCL supports a fine-grained model for the specification of structural dependencies common in object-oriented systems, which can be generic (depend) or more specific (e.g., access, declare, create, extend, etc.). However, since DCL relies on static analysis techniques, it is not able to handle dynamic information or dependencies generated using reflection. A complete description of DCL is found at [102].

In order to check architectural conformance, the architect specifies the DCL constraints, which are continuously enforced by the dclcheck tool.[4] Assuming again the aforementioned strictly layered system, the architect could specify the following constraints:

```
1 module M0: org.m0.*
2 module M1: org.m1.*
3
4 only M1 can-depend M0
5 M1 must-depend M0
```

[4] www.dclsuite.org

First, we define the modules. Modules `M0` and `M1` include respectively all classes from packages `org.m0` and `org.m1` (lines 1–2). To capture divergences, a constraint states that only `M1` can establish dependencies with `M0` (line 4). On the other hand, to capture absences, another constraint requires that classes from `M1` depend on `M0` (line 5).

Other Techniques: Architecture Description Language (ADL), such as Darwin [60], Rapide [56, 57], Wright [3], and ACME [37], represent alternatives that can be used to provide architectural conformance by construction [63]. Such languages allow developers to express the architectural behavior and the structure of software systems in an abstract and declarative language. Code generation tools can then be used to map architectural descriptions to source code in a given programming language. Nevertheless, ADLs require the use of specific architecture-based development tools and compilers, in order to keep the generated code synchronized with the architectural specification. A variant of this approach—such as `ArchJava` [2]—advocates the extension of current programming languages with architectural modeling constructs, which in practice requires developers to use a new programming language.

In a novel line of research, Brunet et al. introduced the concept of design tests, which are test-like programs that automatically check whether the current implementation conforms to a specific design rule [15, 16]. In contrast to DCL, where constraints are defined using a DSL, design rules are implemented as Java code in traditional JUnit test cases. As another approach to architecture conformance, Maffort et al. proposed a data mining approach for architecture conformance based on a combination of static and historical source code analysis that frees architects from specifying the architectural constraints [58, 59]. For this purpose, data mining techniques automatically extract architectural patterns and their tool—called `ArchLint`—uses these patterns to indicate both absences and divergences in source-code based architectures.

2.3 Refactoring

The understanding of refactoring techniques as well their complexity and shortcomings is relevant to our study, since our approach also relies on well-known refactorings to repair architectural violations. This section first presents the basics of refactoring, i.e., concepts, catalogs, goals, etc. Next, it discusses challenges to formalize and therefore to automatically perform refactorings. Finally, it provides an overview of research related to identifying refactoring opportunities.

Refactoring Basics: The concept of refactoring was first coined by Opdyke [75] and has been consolidated by Fowler [32] who has cataloged a collection of program transformations to improve the structural design of systems, i.e., *refactorings*. Refactoring is the process of changing a software system without modifying its external behavior in order to improve the design and understanding of the code. Refactorings are usually guided by *bad smells*, i.e., certain structures in the code or even intuitive signals that indicate the need for refactoring [32, 46]. As an example, the *Extract Method* refactoring turns a code fragment into a new method whose name explains its purpose. Some bad smells might motivate this refactoring, e.g., too long method or duplicated code. As another example, the *Move Method* refactoring moves a method f from a class C to another class C' because f depends more on C' than on its current class C (*Feature Envy*).

Kerievsky proposes a set of coarse-grained refactorings that focus on improving the structure of systems with design patterns by applying sequences of well-known refactorings [46]. For example, he proposes the *Move Creation Knowledge to Factory* refactoring when data and code used to instantiate a class is spread across numerous classes (a bad smell). In this case, the author suggests the creation of a factory class to contain the factory methods and then update all direct instantiations to call the factory method. Analogously in this thesis, we intend to rely as much as possible on traditional refactorings to repair code that violates the existing system design.

By using four large data sets, Murphy-Hill et al. cast doubt on several previously stated assumptions about how programmers refactor the code, while supporting others [71]. They found that, for instance, programmers usually do not indicate refactoring activities in commit logs; refactoring tools are underused; and different refactorings are performed with and without tools. As one of the most relevant results related to this study, although they found that refactoring is indeed commonly practiced, manual refactoring is performed much more often than with a tool. Only very simple refactorings showed the opposite tendency, e.g., *Rename Method* and *Rename Type*.

Complexity: Although the understanding of refactorings is straightforward, the task of formalizing and automating them is not trivial [90, 12, 95, 108, 112, 75, 94, 93]. The mechanics behind refactorings is usually specified in natural language and does not cover all possible scenarios and preconditions. In fact, even a bug-free implementation for typical refactorings—i.e., refactorings whose scope are limited to few classes—has proved to be a complex task [95].

Steimann and Thies demonstrated that most refactoring tools, such as the ones provided by contemporary IDEs, are flawed in terms of maintaining accessibility mod-

ifiers (public, private, etc.) [95]. They argue that the problem is not caused by negligence of the programmers who implemented the tools, but due to the high complexity of object-oriented programming languages. In order to address accessibility problems, they formalized accessibility in Java as a set of constraints rules and provided a framework to check whether a refactoring affects the accessibility of program elements.

Whilst many refactorings have been proposed, Verbaere et al. argue that it would be desirable for programmers to script their own refactorings [112]. For this purpose, they proposed JunGL—a domain-specific language for refactoring—which can be seen as a hybrid of a functional language and a logic query language. For example, the authors were able to implement the well-known *Rename Variable* and *Extract Method* refactorings. Nevertheless, as a limitation, they have not regarded two major features of object-oriented languages: inheritance and accessibility.

Schäfer et al. described a new approach to specify how the refactorings should work using the concepts of dependency preservation, language extensions, and microrefactorings [89]. Because refactoring implementations are commonly complex, hard to understand and maintain, their main goal is to provide modular specifications that are precise enough to serve as the basis of a reimplementation of existing refactorings. Indeed, they re-implemented all Eclipse refactorings and they noticed an improvement on size and clarity of the implementation.

Since it is known that mainstream refactoring engines contain critical bugs, Soares et al. proposed a technique to test Java refactoring engines [94, 93]. Basically, the refactoring under test is applied to hundreds of programs—exhaustively generated by a Java program generator— in order to detect behavioral changes. Their evaluation on 29 refactorings in Eclipse, NetBeans and the JastAdd Refactoring Tools identified 57 bugs related to compilation errors and 63 bugs related to behavioral changes.

In short, the aforementioned studies indicate problems and limitations on automating refactorings. For this reason, we do not intend to address such problems, but to delegate the repairing execution tasks to state-of-the-practice tools, such as the automatic refactorings provided by today's IDEs.

Refactoring Opportunities: Refactorings are usually motivated by bad smells, which are implementation structures that negatively affect system's quality properties, such as understandability, testability, extensibility, and reusability. Fowler has first introduced this concept and proposed an initial set of code bad smells [32]. Thenceforward, many studies were conducted in order to identify refactoring opportunities based on a subset of existing bad smells or on a set of new ones [46, 84, 35, 36, 108, 109, 86, 74, 114].

Kerievsky proposed a set of refactorings that focus on design patterns [46]. On the one side, some of the proposed refactorings are motivated by existing bad smells, e.g., a too long method is a bad smell that may motivate a *Replace Conditional Logic with Strategy* refactoring. On the other side, other refactorings are motivated by new bad smells, e.g., the lack of information hiding—a new bad smell named *Indecent Exposure*—may motivate an *Encapsulate Classes with Factory* refactoring. Roock and Lippert proposed a set of complex refactorings, i.e., large restructurings that takes longer than a day and change significant parts of a system [84]. They mainly presented refactorings of API and relational database refactorings. More important, they identified several *architectural bad smells*—i.e., bad smells that occur at a higher level of the design of a system—and indicated techniques to detect and to avoid them.

In the same line, Garcia et al. described four representative architectural smells—namely Connector Envy, Scattered Functionality, Ambiguous Interfaces, and Extraneous Connector—that emerged from the re-engineering of two large industrial systems [35, 36]. As an example, an *Ambiguous Interface* is an interface that offers only a single, general entry-point into a component. As the system evolves, ambiguous interfaces reduce the system's analyzability and understandability because developers have to inspect the component implementation to be aware of its provided services.

Tsantalis and Chatzigeorgiou proposed a semi-automatic approach to identify Move Method refactoring opportunities [108]. Their general goal is to tackle coupling and cohesion anomalies manifested in the form of the *Feature Envy* bad smell [32]. Similarly to our approach, they employed the notion of a similarity between an entity (attribute or method) and a class. For each method of the system, `JDeodorant` suggests the most appropriate class to be moved to—which is the class that has lower Jaccard distance and satisfies particular preconditions. JDeodorant has been recently extended to also identify *Extract Method* refactorings using an adaptation of program slicing techniques [109] and *Extract Class* refactorings [31].

In the same line, Sales et al. proposed a similar approach also based on the notion of similarity [86, 85]. However, while `JDeodorant` relies on the set of accesses to attributes and methods, `JMove` relies on the set of types that the method establishes dependency with. Their evaluation on thirteen open-source systems indicated that `JMove` was 49% more effective to detect misplaced methods than `JDeodorant`.

O'Keeffe and Ó Cinnéide proposed a search-based software maintenance tool that relies on search algorithms, such as Hill Climbing and Simulated Annealing, to suggest six inheritance-related refactorings [74]. Based on the Quality Model for Object-Oriented Design (QMOOD) [8], they evaluated their approach in two packages of the

`spec.benchmarks` and demonstrated that some programs can be automatically refactored to improve quality in terms of flexibility, reusability, and understandability.

Wong et al. proposed an approach that detects modularity violations through algorithms that compare how components should change together according to their modular structure (structural coupling) with how components actually change together as revealed in version histories (change coupling) [114]. They evaluated their tool—called `Clio`—in 15 releases of Hadoop Common and 10 releases of Eclipse JDT. As a result, they identified hundreds of violations that represent design problems or that were refactored in later versions.

In general terms, the ultimate goal of the aforementioned tools is to suggest refactorings that improve the internal quality of the code—for example, in terms of coupling and cohesion. On the other hand, the repairing recommendation system we propose in this thesis aims to help developers to handle violations exposed as the result of an architecture conformance process.

2.4 Remodularization

Remodularization approaches have been proposed as a potential solution for software aging [76, 78, 25, 111]. Remodularization is a process that changes the modular design of the system for the purposes of adaptation, evolution, or correction.[5] As a matter of fact, architectural erosion is one of the most evident manifestations of software aging. When architecture erosion is neglected over the years, it may reduce the architecture to a set of strongly-coupled and weakly-cohesive components [13]. At a certain point, architecture conformance and refactoring techniques present themselves mostly ineffective and a complete remodularization can be the only solution [42]. Therefore, this section describes some of the most relevant remodularization approaches reported in the literature.

Rama and Patel analyzed several remodularization efforts in order to define recurring modularization operators [80]. More specifically, they formalized the following operators: module decomposition/union, file/function/data transfer, and function to API promotion. Their case study on Linux shows that some operators are continuously applied as the software evolves and therefore they argue that remodularization is not necessarily a one shot process. However, Rama and Patel do not provide tool support

[5] Remodularization does not imply the system behavior should be changed. In fact, in many situations it can be very convenient to change the modular design of a system without changing its external behavior. In this sense, it is quite similar to a refactoring. Perhaps, one could call this an architectural refactoring or perhaps a large scale refactoring.

for applying the proposed operators. Particularly in this thesis, we intend to design an approach that provides tool support and that can also be continuously applied as the system evolves.

Moghadam et al. proposed a remodularization approach that automatically refactors the source code of a system towards a desired design, provided in the format of a UML class diagram [65]. The proposed approach automatically compares the current and desired models and expresses the *design differences* as a set of *detected refactorings* [115]. Although their evaluation indicated a high degree of accuracy, it was conducted in a very restricted context and hence the results may not be as good in real scenarios. Therefore, we intend to evaluate our architectural repair recommendation system using real-world systems in order to avoid similar complaint. Although we would not be either able to extrapolate our results to other systems, we can argue that our evaluation was conducted in a real development scenario.

Bourquin and Keller argue that bad smells and code metrics should be complemented with the analysis of architectural violations to spot opportunities for high-impact refactorings, which are refactorings with a strong impact on the quality of the system's architecture [14]. They report an experience of applying an iterative refactoring process that uses—besides traditional code metrics and well-known bad smells—an architectural conformance tool called Sotograph [9].

Hierarchical clustering is another technique commonly proposed to evaluate alternative software decompositions [5, 64]. However, the effectiveness of clustering in reengineering tasks is often challenged. For example, Glorie et al. reported an experiment in which clustering and formal concept analyses have failed to produce an acceptable partitioning of a monolithic medical imaging application [39]. As another example, using Eclipse as case study, Anquetil et al. reported that restructurings manually performed by developers do not necessarily improve modularity in terms of cohesion/coupling/cyclic dependencies [4]. Regardless the reasons, this finding undermines the validity of clustering techniques that aims to minimize coupling and maximize cohesion. For this reason, we do not intend to rely on traditional metrics in this thesis, but on structural similarity among source code entities to recommend move refactorings.

On the other side, there are studies providing evidence that remodularization is not trivial and should be prevented as far as possible [88, 41]. For example, Sarkar et al. described a process to reengineer a monolithic banking application, which has grown from 2.5 to 25 MLOC and was presenting several maintenance and evolution problems [88]. The root causes of these problems include the fact that the code was not organized by functional domains, the system did not have a layered architecture, and developers commonly mix presentation and business logic in the code. More im-

portant, reconstructing the original architecture of this banking application demanded 2,100 person-days. As another example, High and Sutton described the modularization approach adopted to reengineer Xenon, a large system that had outgrown its initial design [41]. The authors indicated design erosion as the most relevant signal of decay. More important, they employed tools for communication and enforcement—such as SVN commit hooks, FindBugs[6], and Structure101[7]—to prevent architecture decay and consequently to avoid the need of further remodularizations. Besides providing evidence of the complexity behind remodularizations, these studies motivate the development of new approaches and techniques—such as the one proposed in this thesis—that aim to prevent remodularization processes.

Chern and De Volter claimed that the static-dynamic coupling—i.e., the degree to which changes in a program's modular structure imply changes in its dynamic behavior—is a major obstacle to software remodularization [18]. To alleviate this kind of coupling, they proposed a new language, called SubjectJ, which allows class implementations to be split across different files.

In conclusion, most of the difficulties faced during remodularizations are caused by the accumulation of the architectural erosion process over the years. As an evidence, Silva and Balasubramaniam claim that *architecture restoration* strategies should complement *architecture conformance* strategies in order to control architecture erosion [25]. For this reason, our goal in this thesis is to propose an approach that prevents large restructurings by providing suggestions to repair a violation as soon as it is detected.

2.5 Recommendation System

Recommendation Systems for Software Engineering (RSSEs) is an emerging research area [82]. An RSSE is a software application that provides potential valuable information for a software engineering task in a given context. These systems combine computer science and engineering methods to proactively trigger suggestions that meet developers' particular needs. From reusing code (e.g., `CodeBroker` [116]) to recommending existing artifacts for development tasks (e.g., Hipikat [21]), RSSEs help developers in a wide range of activities. This section presents some RSSEs with the central objective to demonstrate the feasibility of a solution based on recommendation system principles with focus on software architecture repair.

Murphy-Hill et al. proposed seven principles for the design of code smell detectors [70]. For example, a code smell detector cannot block the programmer (*unob-*

[6]http://findbugs.sourceforge.net
[7]http://www.headwaysoftware.com

trusiveness) and must explain why the smell exists (*expressiveness*). Particularly in this thesis, we intend to contemplate these principles in the design of our architectural repair recommendation system.

Some recommendation systems aim to raise the effectiveness and efficiency of software quality assurance tasks. As an example, Nagappan et al. proposed an approach called `Suade` that provides suggestions for software investigation [72]. `Suade` collects input data from several sources (bug database, version history, and source code), maps historical failures to existing entities, and then use a combination of metrics to best predict the failure probability of new entities. As another example, Zimmermann et al. proposed an approach called `eRose` that identifies program artifacts that are usually changed together [117]. For instance, when developers need to add a new preference to the Eclipse IDE and therefore have to change the `fKeys[]` field and the `initDefaults()` method, `eRose` would recommend changing also the `Plugin.properties` file, because, according to previous software repository mining results, they are always changed together. In contrast, instead of detecting potential issues, our goal is to assist developers on repairing architectural violations already detected by a conformance technique.

Several recommendation systems have also been proposed to assist developers in using frameworks and APIs. McMillan et al. developed a recommendation system to provide source code examples to developers by querying against API calls and their official documentations [62]. On the other hand, Ye et al. proposed an approach that frees the developers from making queries [116]. They designed a reuse-conducive development environment called `CodeBroker` that analyzes comments and the signature of the method under development to detect similarities to class library elements in order to help developers implement the described functionality. Similarly, Holmes et al. proposed a recommendation system, called `Strathcona` [43], which locates relevant code in an example repository matching the structure of the code under development. The tool also forms the query automatically based on the *structural context*. As a differentiating factor from other approaches, the repository of examples is automatically extracted from existing applications. Thummalapenta et al. proposed an approach, called `PARSEWeb`, that takes queries of the form "*Source Object Type* $\rightarrow$ *Destination Object Query*" and suggests method invocation sequences that can take the *Source* object type as input and result in the *Destination* object type. However, the tool relies on code search engines, instead of a local repository [107]. Montandon et al. proposed a platform—called APIMiner—that instruments the standard Java-based API documentation format with concrete examples of usage [66]. The examples provided by APIMiner are summarized by a static slicing algorithm.

Dagenais et al. claim that finding suitable replacements for program elements deleted as part of a framework evolution can be a challenging task. Therefore, they proposed an approach called `SemDiff` that recommends replacement methods for adapting code to a new framework version, i.e., the tool finds suitable replacements for framework elements that are accessed by a client program but removed as part of the framework's evolution [22]. Analogously, we intend to provide suitable replacements for implementation decisions that denote violations in the planned architecture of a software system.

RSSEs usually present little or no information about the consequences of the recommended changes. An exception is the system proposed by Muşlu et al. to inform developers on the consequences of code transformations [67]. They built an Eclipse plug-in—called `QuickFixScout`—that augments the Quick Fix dialog by adding the number of compilation errors that remain after each proposal's application (a technique they called speculative analysis). Their experiments demonstrate that developers can remove compilation-errors 10% faster when using their tool. Although their technique originally focuses on compilation errors, we intend to reuse it to prioritize our repairing recommendations when more than one can be triggered to fix an architectural violation.

In spite of the fact that several recommendation systems have been proposed to assist developers in many software engineering tasks, there is no system whose precise goal is to help developers in tackling the architectural erosion process. In practice, to repair architectural violations, developers rely solely on their expertise to determine the most appropriate repair action, which may be a time-consuming task. In this thesis, we intend to address this shortcoming through a recommendation system that provides repairing guidelines for developers and maintainers when fixing architectural violations.

2.6 Final Remarks

This chapter provided the background necessary to the fully understanding of the architectural repair solution proposed in this thesis. In Section 2.1, we described different *Architectural Models*, including the one this thesis is based on. According to the software architecture definition by Kruchten [53], our work is centered on the *development view*, which describes the software's static organization in its development environment. This view concerns low-level design decisions, patterns, and best practices.

In Section 2.2, we presented several architecture conformance checking approaches, which is related to our work in terms of detecting violations. Due to the

existence of a large number of conformance approaches, our goal in this thesis is to design an approach generic enough to be integrated to existing techniques and tools.

In Section 2.3, we emphasized the importance of refactorings to improve the design of software systems. We focused on the relevance of bad smells to spot opportunities for refactoring. In this thesis, we intend to identify recurrent architectural violation repair tasks to define a catalog of architectural repairing recommendations.

In Section 2.4, we described several remodularization tools and techniques. We reported studies that show that remodularizations are usually performed when architecture erosion is neglected over the years and consequently architecture conformance and refactoring techniques present themselves mostly ineffective.

Finally, in Section 2.5, we presented several Recommendation Systems for Software Engineering. Despite the fact that RSSEs can assist developers in a variety of software engineering tasks, we have not found recommendation systems whose precise goal is to help developers in tackling the architectural erosion process.

In the next chapter, we present a specification of the architectural repair recommendation system proposed in this thesis, including the underlying algorithms, a detailed description of a subset of the proposed recommendations, and the design of the `ArchFix` tool.

Chapter 3

The Proposed Recommendation System

Chapter 2 provided the background for the full understanding of this thesis. This chapter then describes our approach: an architectural repair recommendation system that provides repairing guidelines for developers and maintainers when fixing architectural violations.

We organized this chapter as follows. Section 3.1 provides an overview of the proposed approach. Section 3.2 provides definitions for central concepts needed to follow our approach, such as *architectural model*, *architectural violations*, *architecturally defective code*, and *repairing recommendations*. Section 3.3 presents a specification of the proposed architectural repair recommendation system, including the description of a subset of recommendations, underlying algorithms, and similarity functions. Section 3.4 describes the design and implementation of the `ArchFix` tool. Finally, Section 3.5 provides a critical discussion about the proposed approach and Section 3.6 concludes with a general discussion.

3.1 Overview

As outlined in Chapter 2, although several architecture conformance techniques have been proposed to detect architectural violations (e.g., reflexion models [68], dependency structure matrices [87], source code query languages [23], constraint languages [102], architecture description languages [2], and design tests [16]), the task of repairing architectural violations is still conducted without the adequate support. More specifically, developers usually perform the repairing task in ad hoc ways, without tool support at the architectural level. As a consequence, developers may spend a long time discovering

how to repair architectural violations and they can even introduce new violations when repairing one. Thereupon, we argue that the task of repairing architectural violations should not be addressed in an ad hoc way because *architecture repair*—although not trivial—is as important as *architecture conformance checking*.

In view of such circumstances, we propose an architectural repair recommendation system whose main purpose is to provide repairing guidelines for developers and maintainers when fixing violations in the module architecture view of object-oriented systems. As illustrated in Figure 3.1, considering a set of architectural violations raised by a static architecture conformance tool, the proposed recommendation engine—called `ArchFix`—provides repairing recommendations to guide the process of fixing each detected violation.

Figure 3.1: Proposed approach

For example, when a class implemented in a wrong module is detected, `ArchFix` may suggest a *Move Method* refactoring to fix the violation, including a suggestion of a target class. As another example, when misusing the programming to interfaces principle [33], our approach may suggest the replacement of a type with one of its most appropriate supertypes. As a last example, when bypassing layers, our recommendation system may suggest the replacement of an unauthorized call with a call to a delegate method. A complete list of the proposed recommendations is presented in Section 3.3.3.

3.2 Basic Concepts

In this section, we provide definitions for the following fundamental concepts employed in this thesis: *recommendation system*, *architectural model*, *architectural violation*, *architecturally defective code*, and *repairing recommendation*.

Recommendation System: A *recommendation system* is a software system that provides potentially valuable information in a given context [82, 81, 17]. As mentioned in Section 2.5 (page 18), in the particular context of software engineering, recommendation systems can recommend, for example, relevant source code fragments to help developers to use frameworks and APIs (e.g., Strathcona [43]), software artifacts that should be changed together (e.g., eRose [117]), and replacement methods for adapting

code to a new library version (e.g., SemDiff [22]). In this thesis, we propose a novel recommendation system, called `ArchFix`, which provides repairing recommendations to fix architectural violations. The proposed system is defined by the following function:

Arch. Model `x` *Arch. Violation* `x` *Arch. Defective Code* $\longrightarrow$ *Repairing Recommendation*

In short, our system receives as input the intended architectural model (*Arch. Model*), the description of the violation (*Arch. Violation*), and the piece of source code that raised the architectural violation (*Arch. Defective Code*). `ArchFix` then returns a *Repairing Recommendation* that might be useful to fix such violation. In the remainder of this section, we discuss such elements in more detail.

Architectural Model: Kruchten defines software architecture using five concurrent *views*, each one addressing a specific set of concerns of interest to different stakeholders [53]. Particularly, our work is centered on the *development view* (a.k.a. *module view*), which describes the software's static organization in its development environment. It concerns low-level design decisions, patterns, and best practices. From this viewpoint, an object-oriented software architecture is defined by a set of modules and their interactions, where we consider a module as a set of classes [102]. Therefore, we model relations at the level of classes. More specifically, a dependency $(\texttt{A}, \texttt{dep}, \texttt{B})$ is established whenever a class `A` uses services provided by a target class `B`. We consider that dependencies can be established using the following types of common operations in object-oriented languages (i.e., these types are possible values of the `dep` field): calling methods or attributes (`access`), declaring variables (`declare`), creating objects (`create`), extending classes (`extend`), implementing interfaces (`implement`), throwing exceptions (`throw`), or using annotations (`useannotation`). For example, the relations $(\texttt{A}, \texttt{create}, \texttt{B})$ and $(\texttt{A}, \texttt{access}, \texttt{B})$ indicate that class `A` creates and calls methods of an object of type `B`, respectively.

The *architectural model* considered by `ArchFix` is expressed in terms of architectural constraints, which are formalized as follows:

$$Module_1 \quad [\texttt{cannot}|\texttt{must}]{-}dep \quad Module_2$$

where *dep* denotes the dependency type, i.e., *dep* can be `access`, `declare`, `create`, etc. As an illustrative example, assume that $\texttt{M}_\texttt{A}$ and $\texttt{M}_\texttt{B}$ are modules, i.e., sets of classes. A constraint in the form $\texttt{M}_\texttt{A}$ `cannot-dep` $\texttt{M}_\texttt{B}$ indicates that types from module $\texttt{M}_\texttt{A}$ *cannot* establish a dependency of the `dep` kind with types from module $\texttt{M}_\texttt{B}$, e.g., `ViewLayer cannot-access ModelLayer`. On the other hand, a constraint in the form

$\texttt{M}_\texttt{A}$ `must-dep` $\texttt{M}_\texttt{B}$ indicates that types from module $\texttt{M}_\texttt{A}$ *must* establish a dependency of the `dep` kind with types from module $\texttt{M}_\texttt{B}$, e.g., `DTO must-implement Serializable`.

Architectural Violation: Basically, there are two types of violations in the static architecture of software systems: *divergences* (when an existing dependency in the source code violates the architectural model) and *absences* (when the source code does not establish a dependency that is prescribed by the architectural model) [68, 50, 77]. In our approach, *divergences* occur when architectural constraints of type `cannot` are not respected by the source code. Conversely, absences occur when architectural constraints of type `must` are not respected.

We consider that an *architectural violation* is defined by a tuple $[\texttt{A}, \texttt{viol\_type}, \texttt{dep}, \texttt{B}]$, where $(\texttt{A}, \texttt{dep}, \texttt{B})$ is the dependency that caused the violation and `viol_type` indicates whether the violation is due to a divergence or an absence. To increase readability, when expressing a value for `viol_type` we use the words `cannot` and `must` to denote divergences and absences, respectively. For example, a violation $[\texttt{ProductView}, \texttt{cannot}, \texttt{access}, \texttt{ProductModel}]$ denotes a divergence where a class `ProductView` *is* accessing an object of type `ProductModel`. As another example, a violation $[\texttt{ProductDTO}, \texttt{must}, \texttt{implement}, \texttt{Serializable}]$ denotes an absence when the class `ProductDTO` *is not* implementing the interface `Serializable`.

Formal Definition: Assume that $\texttt{M}_\texttt{A}$ and $\texttt{M}_\texttt{B}$ are modules, `A` and `B` are classes, and that `dep` denotes a dependency type, i.e., `dep` can be `access`, `declare`, `create`, etc. A violation of a constraint in the form $\texttt{M}_\texttt{A}$ `cannot-dep` $\texttt{M}_\texttt{B}$ happens whenever

$$\exists \texttt{A}\, \exists \texttt{B}\, [\, \texttt{A} \in \texttt{M}_\texttt{A} \ \wedge \ \texttt{B} \in \texttt{M}_\texttt{B} \ \wedge \ \texttt{establishes}(\texttt{A}, \texttt{dep}, \texttt{B})\,]$$

where the predicate `establishes` checks whether there is a dependency of type `dep` from `A` to `B`. Conversely, a violation of a constraint in the form $\texttt{M}_\texttt{A}$ `must-dep` $\texttt{M}_\texttt{B}$ happens whenever

$$\exists \texttt{A}\, \nexists \texttt{B}\, [\, \texttt{A} \in \texttt{M}_\texttt{A} \ \wedge \ \texttt{B} \in \texttt{M}_\texttt{B} \ \wedge \ \texttt{establishes}(\texttt{A}, \texttt{dep}, \texttt{B})\,]$$

Architecturally Defective Code: An architectural violation, as defined before, statically occurs in a piece of code. In our recommendation system, we consider that the code responsible for a violation (i.e., the *architecturally defective code*) follows one of the code templates defined in Table 3.1. As an example, the template `new B(exp)` cov-

Table 3.1: Code templates

Template	Interpretation
`class A`	Class implementation
`class A derive B`	Class that extends or implements type `B`
`@B class A`	Class with an annotation of type `B`
`g (p){ S }`	Implementation of a method `g`, with formal parameters `p` and body `S` (`S` is matched in its maximal form; also in the following code templates)
`@B g (p){ S }`	Implementation of a method `g`, with annotation `@B`, formal parameters `p`, and body `S`
`g (B b){ S }`	Implementation of a method `g`, with a formal parameter of type `B`, and body `S`
`g (p){ T v = exp_b }`	Implementation of a method `g`, with formal parameters `p` and whose body declares a local variable `v` of type `T` initialized with `exp_b` (which returns an object of type `B`, a subtype of `T`)
`g (p){ return new B(exp) }`	Implementation of a method `g`, with formal parameters `p`, returning an object of type `B` created using parameters `exp`
`g (p) throws B { S }`	Implementation of a method `g`, with formal parameters `p`, and body `S` that can throw an exception of type `B`
`B b; S`	Declaration of a variable `b` of type `B` followed by statements `S`
`B b = exp; S`	Declaration of a variable `b` of type `B`, initialized with `exp` and followed by statements `S`
`try { S } catch (B b) { S'}`	Implementation of a *try-catch* block with protected body `S` containing a clause for exceptions of type `B` with handling body `S'`
`b.f(exp)`	Invocation of a method `f` using the target object `b` and actual parameters `exp`
`new B(exp)`	Instantiation of an object of type `B` using parameters `exp`

ers instantiations of a given class `B`. As another example, consider the architectural constraint `UI cannot−access Bar` and the class `Screen` presented in Code 3.1.

```
1 public class Screen {
2     private Bar bar;
3     ...
4     public void init() {
5         ...
6         bar.foo(13,'b');
7     }
8 }
```

Code 3.1: Code template matching example

Assuming that `Screen` is located in module `UI`, a violation [`Screen`, `cannot`, `access`, `Bar`] is raised by the call `bar.foo(13,'b')` located at line 6. This call matches the template `b.f(exp)` with `b` bound to the target object `bar`, `f` bound to the method `foo`, and `exp` bound to the list of expressions [`13`, '`b`'].

Repairing Recommendation: In our approach, the recommendations consist in a repair activity to fix an architectural violation. In fact, a *repairing recommendation* is a sequence of repairing operations, using the functions described in Table 3.2. Although most repairing functions are familiar, Appendix A provides a detailed description of the following particular functions: `inline`, `promote_param`, and `unwrap_return`. Particularly, we may suggest transformations that may change the semantics/behavior of the system and may also not be well-formedness preserving. An in-depth discussion on the behavior preservation of our repairing recommendations is presented in Section 3.5.

Table 3.2: Repairing functions

Function	Description	Type
`extract(stm)`	Applies an *Extract Method* refactoring [32] with the set of statements `stm`	*ct*
`inline(exp, v, S)`	Inlines `exp` in the uses of variable `v` in the block of code `S` (see Appendix A)	*ct*
`move(f, M)`	Moves method `f` to the most suitable class in module `M`, i.e., applies a *Move Method* refactoring [32]	*mv*
`move(C, M)`	Moves class `C` to module `M`, i.e., applies a *Move Class* refactoring [32]	*mv*
`promote_param(f, v, exp)`	Promotes variable `v` to a formal parameter of method `f`; `exp` is used as the additional argument in the calls to `f` (see Appendix A)	*ct*
`replace(`S_1, S_2`)`	Replaces block of code S_1 with S_2	*ct*
`remove(S)`	Removes block of code `S`, which is equivalent to `replace(S,` ϕ`)`	*ct*
`remove_catch(Ex, S)`	Removes the catch clause for exception `Ex` from the *try-catch* block `S`	*ct*
`unwrap_return(f, T, exp)`	Modifies the return type of method `f` to the type of `exp` and moves the instantiations of the wrapper type `T` to the respective call sites (see Appendix A)	*ct*

As reported in Table 3.2, the recommendations include two types of transformations:

- (*ct*) stands for a *code transformation*, which is a repair action more common to handle divergences. For instance, suppose that a given module should not use `List` objects, but it can use `Collection`. In this case, function `replace([List], [Collection])` locally replaces a declaration of the unauthorized type `List` with its supertype `Collection`. As another example, we can mention a repair action that promotes a local variable to a formal parameter. In this case, function `promote_param(f, v, exp)` locally modifies the method and globally adjusts call sites of the respective method.

- (*mv*) stands for a transformation that requires *moving* code, which may be useful to handle both divergences and absences, as will be discussed in Sec-

tion 3.3.5. As an example, suppose that a class `ProductReport` is implemented in module `Controller`, while the architectural model prescribes that all reports must be implemented in module `Report`. In this case, function `move(ProductReport, Report)` may indicate the correct repair action to handle this particular violation.

3.3 Architectural Repairing Recommendations

This section defines the architectural repairing recommendations triggered by `ArchFix`. We start by explaining how the recommendations have emerged and then we define and illustrate our recommendations.

3.3.1 Training System

The proposed recommendations emerged after an in-depth investigation of possible fixes for more than 2,200 violations we detected in a previously evaluated system called SGP (our training set system, which is a system different from the systems used for evaluation in this thesis) [102]. The choice of SGP was motivated by the following facts: (i) it is a large web system with around 240 KLOC, (ii) its architecture reflects the one commonly used in Java-based web systems, and (iii) the system was facing a serious architectural erosion process, i.e., we detected 2,241 architectural violations.

We considered this set of violations as our *training set* to define our recommendations. We analyzed and generalized the solution (repairing task) employed by the system architect for each violation. More important, we also documented short preconditions to minimize potential side effects of the repair actions, such as compilation errors or new violations. As a result, we reached a catalog of 32 architectural repairing recommendations. We hypothesized that many of the recommendations would be useful in other systems (a discussion on the completeness of our repairing recommendations is presented in Section 3.5). Our evaluation confirms this hypothesis; however, we acknowledge that the catalog is likely incomplete.

3.3.2 Syntax and Auxiliary Functions

Assuming the definitions and concepts outlined in Section 3.2, the architectural repairing recommendations triggered by our system are defined using the following syntax:

```
[A, viol_type, dep, B]
code_template  ⟹  recommendation,  if preconditions
```

This recommendation syntax is interpreted as follows: whenever the indicated violation [A, viol_type, dep, B] is found in a piece of code that matches the code_template and the preconditions hold, the recommendation is triggered (i.e., suggested to the user).

A *recommendation* consists of one or more *repairing* functions, as previously defined in Table 3.2. Table 3.3 lists a set of auxiliary functions used to define the preconditions. Some of such functions define a simple source code query language, including functions such as call_sites(f), which returns the call sites of a given method f, and super(T), which returns the supertypes of a given type T. Other functions are slightly more complex, such as can(T_1, dep, T_2), which checks whether the establishment of a dependency (T_1, dep, T_2) raises an architectural violation, and typecheck(stm), which checks whether a piece of code compiles without type errors. Appendix B provides a detailed description of the following non-trivial functions: delegate, factory, gen_decl, and gen_factory.

Table 3.3: Auxiliary functions

Function	Description
call_sites(f)	Returns the statements that call method f
can(T_1, dep, T_2)	Checks whether a dependency of the dep type from type T_1 to type T_2 does not raise any violation, i.e., respects the architectural model
has_catch(Ex, S)	Checks whether there is a *catch* clause for exception Ex in the *try-catch* block S
delegate(f)	Searches for a delegate method for f, i.e., a method that just encapsulates a call to f (see Appendix B)
equals_sig(f_1, f_2)	Checks whether methods f_1 and f_2 have the same signature
factory(C, exp)	Searches for a factory method for class C, accepting exp as input (see Appendix B)
gen_decl(f)	Declares a variable of the type C, which defines the method f, to be the target of a call to the method f (see Appendix B)
gen_factory(C, exp)	Generates a factory for class C, accepting exp as parameter (see Appendix B)
override(C_1, C_2)	Checks whether class C_1 overrides a method defined in superclass C_2
sub(T)	Returns the subtypes of type T
suitable_module(E)	Returns the most suitable module for a source code entity E (see Section 3.3.5)
super(T)	Returns the supertypes of type T
target(A)	Returns the target entity of an annotation A, which can be a *type* or a *method*
type(v)	Returns the type of variable v
typecheck(stm)	Checks whether code stm type checks (the context is implicitly assumed)
user_code()	Returns a block of code with only a TODO comment, which the user must fill

3.3.3 Recommendations

Table 3.4 specifies the recommendations using the aforementioned syntax and functions. The table formalizes recommendations to address both divergences

(recs. $D1$ to $D24$) and absences (recs. $A1$ to $A8$). In the specifications, we consider that $\texttt{M}(\texttt{A})$—or just $\texttt{M}_\texttt{A}$ for the sake of simplicity—is a total function that returns the module of a given class `A`. We also consider that $\overline{\texttt{M}_\texttt{A}}$ is the complement of the module returned by $\texttt{M}_\texttt{A}$, i.e., all classes in the project under analysis except those in $\texttt{M}_\texttt{A}$.

As further described in Section 3.3.4, when multiple recommendations match a given violation, we sort the potential recommendations according to the number of architectural violations that remain after their application (a technique known as speculative analysis [67]). In other words, the recommendation that repairs the largest number of architectural violations takes the highest priority.

To provide an overview of our architectural repairing recommendations, we describe next a small subset of our recommendations:

***D*2**: Replace the unauthorized type `B` with one of its subtypes $\texttt{B}'$. As an example, developers when implementing web-based systems using GWT (Google Web Toolkit) should avoid the use of generic types (e.g., `java.util.Collection`) on GWT interfaces to reduce the size of the generated JavaScript code. Therefore, whenever possible, they should rely on more specialized types (e.g., `java.util.ArrayList` instead of `java.util.Collection`).

***D*9**: Remove a call to a given method `f` when no class in the system can access the class where `f` is implemented. This recommendation is particularly useful when developers access methods whose usage is restricted. For instance, developers tend to establish dependencies with the Java API `System` class (e.g., by calling `System.out.println`) as a practice of rudimentary debugging. Nevertheless, these calls must be removed, especially in web-based systems.

***D*11**: Replace a `new` operator with a call to the `get` method of a Factory `FB`. It addresses the situation where developers—due to unawareness or forgetfulness—create directly objects of classes that have a well-defined factory.

***D*16**: Remove the `throws` clause and insert a `try−catch` block around the body of the method to handle a given exception internally. In this particular case, the developers must provide the code that handles the exception, as required by the auxiliary function `user_code`.

***D*20**: Move a class `A` to a most suitable module `M`. It is particularly useful when developers mistakenly implement a class in the wrong module, e.g., a `ProductReport` class in the `View` layer.

Table 3.4: Repairing Recommendations

[A, cannot, declare, B]		
B b; S	$\Longrightarrow$ replace([B], [B′]), if B′ ∈ super(B) ∧ typecheck([B′ b ; S]) ∧ B′ ∉ M_B	$D1$
B b; S	$\Longrightarrow$ replace([B], [B′]), if B′ ∈ sub(B) ∧ typecheck([B′ b ; S]) ∧ B′ ∉ M_B	$D2$
B b = exp; S	$\Longrightarrow$ propagate([exp], b, [S]), if can(A, access, B)	$D3$
g (B b) { S }	$\Longrightarrow$ remove([B b]), if typecheck([g(){ S }])	$D4$
try { S } catch (B b) { S′ }	$\Longrightarrow$ replace([B], [B′]), if B′ ∈ super(B) ∧ typecheck([try { S } catch (B b) { S′ }]) ∧ B′ ∉ M_B	$D5$
[A, cannot, access, B]		
b.f	$\Longrightarrow$ replace([b.f], [D; c.g]), if g = delegate(f) ∧ ⟨D,c⟩ = gen_decl(g) ∧ type(c) ∉ M_B	$D6$
b.f	$\Longrightarrow$ replace([b.f], [D; c.g]), if equals_sig(f, g) ∧ ⟨D,c⟩ = gen_decl(g) ∧ type(c) ∉ M_B	$D7$
b.f	$\Longrightarrow$ g = extract([b.f]), move(g, M), if M = suitable_module(g) ∧ can(A, access, M)	$D8$
b.f	$\Longrightarrow$ remove([b.f]), if $\overline{M_A} = \emptyset$	$D9$
g (p){ T v = exp_b }	$\Longrightarrow$ promote_param(g, v, [exp_b]), if ∀C ∈ call_sites(g), can(C, access, B)	$D10$
[A, cannot, create, B]		
new B(exp)	$\Longrightarrow$ replace([new B(exp)], [FB.getB(exp)]), if FB = factory(B, [exp]) ∧ can(A, access, FB)	$D11$
new B(exp)	$\Longrightarrow$ replace([new B(exp)], [null]), if $\overline{M_A} = \emptyset$	$D12$
new B(exp)	$\Longrightarrow$ replace([new B(exp)], [FB.getB(exp)]), if FB = gen_factory(B, [exp]) ∧ can(A, access, FB)	$D13$
g (p){return new B(exp)}	$\Longrightarrow$ unwrap_return(g, B, [exp]), if ∀C ∈ call_sites(g), can(C, create, B)	$D14$
[A, cannot, throw, B]		
g (p) throws B { S }	$\Longrightarrow$ remove([throws B]), if typecheck([g (p) { S }])	$D15$
g (p) throws B { S }	$\Longrightarrow$ remove([throws B]), replace([S], [try {S} catch(B b) {S′}]), if can(A, declare, B) ∧ S′ = user_code()	$D16$
g (p) throws B { S }	$\Longrightarrow$ replace([B], [B′]), if B′ ∈ super(B) ∧ B′ ∉ M_B	$D17$
g (p) throws B { S }	$\Longrightarrow$ move(g, M), if M = suitable_module(g) ∧ M ≠ M_A	$D18$
[A, cannot, derive, B]		
class A derive B	$\Longrightarrow$ replace([B], [B′]), if B′ ∈ super(B) ∧ typecheck([A derive B′]) ∧ ¬override(B, B′) ∧ B′ ∉ M_B	$D19$
class A derive B	$\Longrightarrow$ move(A, M), if M = suitable_module(A) ∧ can(A, derive, B)	$D20$
[A, cannot, useannotation, B]		
@B class A	$\Longrightarrow$ move(A, M), if M = suitable_module(A) ∧ M ≠ M_A	$D21$
@B class A	$\Longrightarrow$ remove([@B]) if M_A = suitable_module(A)	$D22$
@B g (p){ S }	$\Longrightarrow$ move(g, M), if M = suitable_module(g) ∧ M ≠ M_A	$D23$
@B g (p){ S }	$\Longrightarrow$ remove([@B]) if M_A ≠ suitable_module(A)	$D24$
[A, must, throw, B]		
g (p){ S }	$\Longrightarrow$ replace([g (p){ S }] , [g (p) throws B { S }]), remove_catch(B, S) if has_catch(B, S)	$A1$
g (p){ S }	$\Longrightarrow$ move(g, M) if M = suitable_module(g) ∧ M ≠ M_A	$A2$
[A, must, derive, B]		
class A	$\Longrightarrow$ replace([A], [A derive B]), if M_A = suitable_module(A) ∧ typecheck([class A derive B])	$A3$
class A	$\Longrightarrow$ move(A, M), if M = suitable_module(A) ∧ M ≠ M_A	$A4$
[A, must, useannotation, B]		
class A	$\Longrightarrow$ move(A, M), if M = suitable_module(A) ∧ M ≠ M_A ∧ target(B) = type	$A5$
class A	$\Longrightarrow$ replace([class A], [@B class A]), if M_A = suitable_module(A) ∧ target(B) = type	$A6$
g (p){ S }	$\Longrightarrow$ move(g, M), if M = suitable_module(A) ∧ M ≠ M_A ∧ target(B) = method	$A7$
g (p){ S }	$\Longrightarrow$ replace([g(p)], [@B g(p)]), if M_A = suitable_module(g) ∧ target(B) = method	$A8$

A3: Make class `A` extend or implement `B`. It addresses the situation where developers have failed to derive from the base types of the module. For instance, an `Entity` class must implement `Serializable` to provide persistence. However, assume that a given entity class `Product` does not implement `Serializable`. Because `Entity` classes rely extensively on the same types, the `suitable_module` function will likely infer that `Product` is indeed in its correct module and therefore must implement `Serializable`.

Chapter 5 provides concrete examples on the use of the proposed recommendations in two real-world systems. Furthermore, a detailed description of all architectural repairing recommendations is available in Appendix C.

3.3.4 Algorithm

In order to provide an operational description for our approach, Algorithm 1 presents the steps followed by `ArchFix` to trigger architectural repairing recommendations. First, function `getRecommendations` returns a list with the possible recommendations for a particular violation (line 1). For each recommendation $r_i \in potentialRecs$, we verify whether the architecturally defective code matches the code template and whether the preconditions are satisfied (lines 3–7). In the positive cases, we insert r_i in the resulting list (line 5). Prior to return the selected recommendations (line 9), we sort them according to the number of architectural violations that remain in the system after their application, i.e., we employ a technique known as speculative analysis [67] (line 8).[1]

Algorithm 1 Recommendation Algorithm

Input: Architectural model (*arch*), violation (*viol*), and defective code (*code*)
Output: List of repairing recommendations (*result*)

1: $potentialRecs \leftarrow$ `getRecommendations`$(viol)$
2: $result \leftarrow \phi$
3: **for each** r_i ***in*** $potentialRecs$ **do**
4: **if** (`match`$(code,\ r_i.code\_template)$ and `satisfy`$(\langle code, viol, arch\rangle, r_i.preconditions)$) **then**
5: $result \leftarrow result + r_i.recommendation$
6: **end if**
7: **end for**
8: $result \leftarrow$ `prioritize`$(result)$ ▷ by the # of remaining violations
9: **return** $result$

[1] Although we decided to consider the remaining violations globally in the system, it is straightforward to consider the remaining violations more locally, i.e., in the respective method or class.

For instance, assume a violation [A, cannot, derive, B]. Consequently, the list of potential recommendations is [D19, D20]. Assume also that such violation matches the code template and satisfies the preconditions of both recommendations. However, suppose that if the maintainer accepts the recommendation $D19$, three architectural violations would still remain in the system. On the other hand, if the maintainer accepts the recommendation $D20$, only one violation will remain. In this case, our algorithm returns the priority-sorted list of recommendations [D20, D19]. In case of tie, we rely on the pre-defined order (i.e., [D19, D20]), which was defined considering the complexity of the recommended repair actions. For instance, it is simpler to change the superclass of a class (D19) than to move the entire class to another package (D20).

3.3.5 Module Suitability

In Table 3.4, a number of 14 out of 32 recommendations (e.g., $D18$, $D20$, $D21$, $A4$, etc.) include a suggestion to move methods or classes to more *suitable* modules, as computed by the `suitable_module` function. In essence, this function considers the structural similarity among source code entities to make a recommendation. In order to measure this similarity, we rely on the *Relative Matching* coefficient [83], which is a statistical measure for the similarity between two sets. To calculate this coefficient, we assume that a given source code entity (method, class, or package) is represented by the set of types it establishes dependency with. This assumption is based on the fact that our recommendations are proposed to remove divergences or absences in the structural dependencies established between the modules of object-oriented software architectures. Furthermore, Chapter 4 describes an empirical study that supports this assumption.

The *Relative Matching* similarity between source code entities i and j is defined by:

$$S(i,j) = \frac{a + \sqrt{ad}}{a + b + c + d + \sqrt{ad}}$$

where a denotes the number of common types on both entities, b denotes the number of types on entity i only, c denotes the number of types on entity j only, and d denotes the number of types on neither of the entities. In this way, this coefficient considers a set of similarity properties such as no mismatch, minimum match, no match, complete match, and maximum match, as detailed in Section 4.2.1.

To determine the most suitable module for a *method or class* e_i, the `suitable_module` function returns the respective *class or package* m that provides the following maximal value:

$$\max_{\forall m} \; S(\, Deps(e_i), Deps(m)\,)$$

where $Deps(E)$ is the function that returns the set of types that the source code entity E establishes dependency with.

Illustrative example: Using a simple web-based e-commerce system developed for academic purposes called `MyWebMarket` [105], we implemented `ProductReport`—a class responsible for report generation—in module `Controller`. Particularly in this system, the architectural model prescribes that `Controller` classes must extend `ActionSupport`. However, suppose that `ProductReport` does not extend `ActionSupport` and hence a violation in the form [`ProductReport`, `must`, `extend`, `ActionSupport`] is raised by a static architecture conformance checking tool.

In this scenario, our recommendation system can trigger either recommendation $A3$ or $A4$, depending on the result of the `suitable_module` function. In order to determine the correct module for class `ProductReport`, the function returns the package m that provides the following maximal value:

$$\begin{aligned}\max \{ \\ S(\, Deps(\texttt{ProductReport}), Deps(\texttt{Report})\,) &= 0.328, \\ S(\, Deps(\texttt{ProductReport}), Deps(\texttt{Controller})\,) &= 0.154, \\ S(\, Deps(\texttt{ProductReport}), Deps(\texttt{DAO})\,) &= 0.142, \\ S(\, Deps(\texttt{ProductReport}), Deps(\texttt{Mail})\,) &= 0.088, \\ S(\, Deps(\texttt{ProductReport}), Deps(\texttt{DTO})\,) &= 0.085, \\ S(\, Deps(\texttt{ProductReport}), Deps(\texttt{Schedule})\,) &= 0.083, \\ S(\, Deps(\texttt{ProductReport}), Deps(\texttt{Ajax})\,) &= 0.000 \,\} = \texttt{Report}\end{aligned}$$

As the result, the suitable module heuristic indicates module `Report` as the most suitable one. Since `Report` is not the current module of the class `ProductReport`, our approach triggers recommendation $A4$—`move`(`ProductReport`, `Report`)—which is exactly the correct repair action to handle this particular violation.

3.4 The `ArchFix` Tool

We developed a prototype tool called `ArchFix` that implements our approach. In its current implementation, `ArchFix` relies on violations raised by the DCL lan-

guage [102, 100], which is a domain-specific language for defining structural constraints between modules in Java. Compared to other approaches, DCL relies on a simple syntax and supports *architecture compliance by construction*, in order to proactively prevent architecture decay [49]. For example, to capture divergences, DCL supports constraints expressing that dependencies *only can*, *can only*, or *cannot* be established by specified modules. In addition, to capture absences, it is possible to specify constraints to check that particular dependencies *must* be present in the source code.

More specifically, we implemented `ArchFix` as an extension of `DCLcheck` [102], an Eclipse-based conformance tool that checks architectural constraints defined in DCL. As illustrated in Figure 3.2, `ArchFix` exploits some preexisting data structures available in `DCLcheck`, such as the graph of existing dependencies, the defined architectural constraints, and the detected violations. Moreover, `ArchFix` also reuses some auxiliary functions from `DCLcheck`, e.g., a function that checks whether a type can establish a particular dependency with another type (function `can`, Table 3.3).

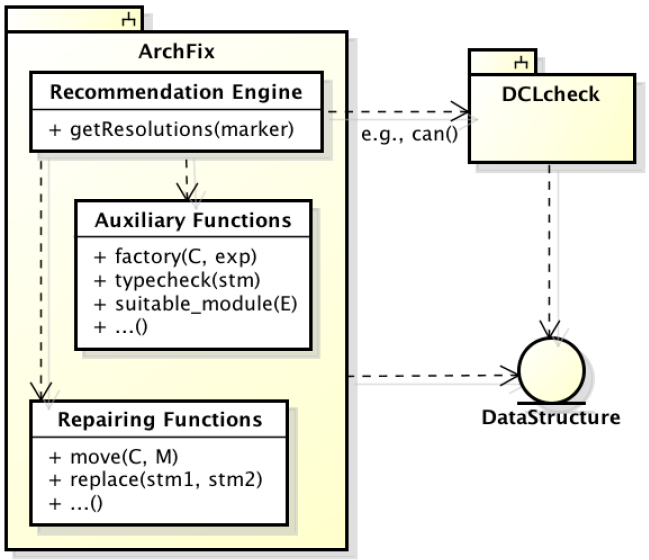


Figure 3.2: `ArchFix` architecture

`ArchFix` follows an architecture with three main modules:

- *Recommendation Engine*: Based on the specification shown in Table 3.4, this module is responsible for suggesting appropriate repairing recommendations for a given violation (when applicable). In fact, this module implements the Algorithm 1 presented in Section 3.3.4. In our current implementation, `ArchFix` was designed as a marker resolution because `DCLcheck` marks architectural violations

in the source code. In other words, `ArchFix` is invoked whenever the developers request *Quick Fixes* for *problem markers*[2] that represent architectural violations.

As illustrated in Figure 3.2, the module `Recommendation Engine` inspects the recommendations using services from modules `Auxiliary` and `Repairing Functions`. More specifically, the `Recommendation Engine` exports the public method `getResolutions` that receives a problem marker as input—which contains information on the architectural violation (e.g., the violated constraint and the architecturally defective code)—and then returns a list of potential repairing recommendations.

- *Auxiliary Functions*: This module implements the auxiliary functions listed in Table 3.3.

- *Repairing Functions*: This module is responsible for applying the repair actions listed in Table 3.2. In the current stage of its implementation, `ArchFix` is mostly a recommendation engine, i.e., the tool suggests actions to repair architectural violations. Nevertheless, `ArchFix` already supports some simple repair actions, such as replacing a type and removing annotations (which are only applied when the user accepts a given recommendation).

To illustrate `ArchFix`'s interface, assume an architectural constraint of the form `ControllerLayer cannot−declare HibernateDAO`. Assume also a class `ProductController` that declares a variable of a type `ProductHibernateDAO`, i.e., a class that introduces a violation in the form [`ProductController`, `cannot`, `declare`, `HibernateDAO`]. When the maintainer requests a recommendation to fix such violation, `ArchFix` indicates the most appropriate repair action (see Figure 3.3). The provided recommendation suggests replacing the declaration of the unauthorized type `ProductHibernateDAO` with its interface `IProductDAO` (which corresponds to recommendation $D1$ in Table 3.4). This recommendation is particularly useful to handle violations due to references to a concrete implementation of a service, instead to its general interface.

3.5 Discussion

This section includes a critical analysis about the architectural repair recommendation system proposed in this thesis. Our analysis is based on the following criteria:

[2]A *problem marker* (an object of type `org.eclipse.core.resources.problemmarker`) represents an error or warning listed in the *Problems* view of the Eclipse IDE.

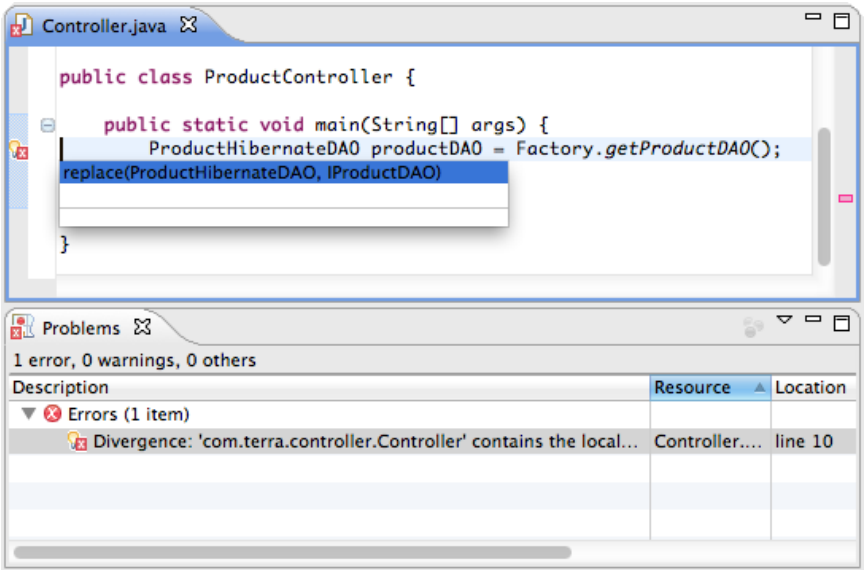


Figure 3.3: `ArchFix` interface

Completeness: Although our repairing recommendations have emerged from a web-based system (refer to Section 3.3.1) and our evaluation relies on two web-based systems (refer to Chapter 5), we claim that our approach is not limited to web-based systems. In fact, our approach is applicable in systems that are structured into modules and that specifies constraints on how these modules must interact [38, 33]. However, we agree that we need new case studies on different contexts to provide concrete evidences for such claim.

The current implementation of our approach—`ArchFix`—relies on architectural violations detected by DCL for the following reasons: (i) simplicity; (ii) the language is based on a fine-grained model for the specification of structural dependencies; and (iii) `DCLcheck` is an open-source conformance tool that provides an easy way to be extended and adapted. Nevertheless, we claim that it is straightforward to adapt our approach to other conformance techniques and tools.

Module Suitability: Even though the way that developers decide how to fix a violation might be more semantically oriented—i.e., considering the concerns implemented in the source code—we decide to rely on structurally oriented heuristics (refer to Section 3.3.5). This decision is based on the fact that our recommendations are proposed to repair divergences or absences in the structural dependencies established between the modules of object-oriented software architectures. However, we argue that semantic properties (as revealed, for example, by the source code vocabulary) might enhance the precision of our module suitability heuristic.

Behavior Preservation: Our repairing recommendations *cannot* always ensure behavior preservation (refer to Section 3.2). In practice, some architectural repair activities may lead to semantic changes. For instance, assume that a class `B` extends a class `A`. Assume also that `A` defines a method `f` and `B` does not override it. Therefore, when invoking `f` from an object of type `B`, it is actually invoked `A::f()`. Nevertheless, when the user accepts a recommendation to move another method `f` to `B` (e.g., rec. $D23$), the system behavior is changed. In this case, when invoking `f` from an object of type `B`, the moved method `B::f()` is called instead of `A::f()` as expected. In short, it is the developer's responsibility to carry out the recommendations without disturbing the system semantics. In fact, our approach does *not* actually apply the repair actions but suggests them to the user who decides to apply or not.

Finally, it is worth noting that: (i) our preconditions may *not* be complete. In some cases, they only provide minimal conditions to prevent the insertion of new violations and to reduce undesirable side effects; and (ii) our recommendation system was not designed to deal with concurrency issues. It means that the user must pay closer attention when our approach suggests moving or transforming synchronized code.

3.6 Final Remarks

Architectural erosion is a recurrent problem faced by software architects. However, a clear dichotomy is perceived in the techniques already proposed to tackle this problem. On the one hand, there are several approaches and commercial tools proposed to uncover architectural violations [68, 87, 23, 102, 2, 16]. On the other hand, the task of fixing the hundreds of violations raised by an architecture conformance process is normally conducted with limited tool support.

To address this shortcoming, we described a recommendation system that provides repairing guidelines for developers and maintainers when fixing architectural violations. To automate the task of triggering recommendations, we developed a prototype tool called `ArchFix` that implements our approach and provides recommendations for violations—divergences and absences—raised by static architectural conformance techniques. The `ArchFix` tool—including its source code—is publicly available at `github.com/rterrabh/DCL`.

In the next chapter, we provide the empirical evidences supporting the implementation decisions related to our suitable module heuristic, as computed by the `suitable_module` function (refer to Section 3.3.5). Beyond that, we introduce the Qualitas.*class* Corpus, which is the dataset the ensuing empirical study relies on.

Chapter 4

Evaluation of the Suitable Module Heuristic

From the repairing recommendations proposed in the previous chapter, fourteen recommendations (e.g., $D18$, $D20$, $D21$, $A4$, etc.) rely on the `suitable_module` function, which implements our heuristic based on structural similarity (refer to Section 3.3.5, page 34). However, the problem of determining whether methods or classes are well located is not trivial [98]. In this chapter, our goal is to provide empirical evidences for: (i) the choice of the *Relative Matching* coefficient; and (ii) the dependency set of a source code entity (e.g., a method or a class) contemplates only the types that it establishes dependency with, e.g., only a single [`ProductDAO`] instead of a pair [`access`, `ProductDAO`].

We organized this chapter as follows. Section 4.1 introduces the Qualitas.*class* Corpus, which represents the dataset our evaluation relies on. Section 4.2 describes an empirical study that compares several coefficients and strategies to identify which one is the most appropriate in determining where a source code entity should be located. Section 4.3 concludes this chapter with a general discussion.

4.1 The Qualitas.*class* Corpus

The Qualitas.*class* Corpus is a compiled version of the Qualitas Corpus proposed by Tempero et al. [97]. In short, it provides compiled Eclipse Java projects for the 111 systems included in the last release of the original corpus.[1] Table 4.1 provides an overview on our compiled corpus. It contains more than 18 million LOC, 200,000 compiled classes, and 1.5 million compiled methods. As another contribution, we have

[1]The current release is 20120401.

gathered a large amount of metrics data (such as measurements from size, coupling, and CK metrics) about the systems (see Section 4.1.2).

Table 4.1: Qualitas.*class* Corpus

Systems	111
Lines of Code (LOC)	18,548,026
Internal Projects (NOIP)	802
Packages (NOP)	16,509
Classes (NOCL)	202,052
Interfaces (NOI)	22,115
Methods (NOM)	1,464,893

The original Qualitas Corpus provides a valuable contribution for experimentation in software engineering. However, there are several scenarios—e.g., experiments that rely on Abstract Syntax Tree (AST) or bytecode, as the one reported in this chapter—in which researchers need to import and compile the source code. Since this task is not trivial in the case of systems with many external dependencies, we decided to address such effort by providing a compiled variant of the Qualitas Corpus.

4.1.1 Compilation Process

The corpus contains a collection of **systems**, each of which contains *one or more* **projects**. For instance, the AspectJ system is divided in four internal projects: `matcher`, `rt`, `tools`, and `weaver`. Thus, considering the 111 systems, the Qualitas.*class* Corpus has a total of 802 internal projects (including 461 projects from NetBeans).

Since the Qualitas Corpus provides us with the source code of the systems, our main work consisted in organizing the code in well-defined Java projects according to the following guidelines:

- In the case of outdated systems, we compiled their most recent version, instead of those in the original Qualitas Corpus.
- Source code distributions usually do not include third-party libraries, which are required by the compilation process. We hence searched for these libraries using Maven repositories[2] or version control systems.
- Some projects rely on particular libraries. For instance, JTOpen 7.8 relies on the IBM AS/400 library, which is not publicly available. In these cases, we manually created *stub* JAR files to simulate the internal structure of the libraries (e.g., `ibm_as400_stub.jar`).

[2]Maven repositories: search.maven.org and mvnrepository.com.

- Some projects presented compilation errors. In these cases, we fixed the error and explained our fixing procedure in a comment.[3] For instance, package `etc.testcases` on Ant 1.8.2 has some classes whose file name differs from the public class name. Therefore, we renamed the class names and inserted comments in the code to explain the changes.

4.1.2 Measurements

The Qualitas.*class* Corpus also includes the values of 23 source code metrics measured at the level of classes (detailed in Appendix D). In a summarized perspective, Figure 4.1 illustrates the distribution of the average value for a subset of metrics. Basically, each circle represents a system and the values in the vertical indicate the overall average for each metric.[4] As can be noticed, the corpus is very heterogeneous. For example, the MLOC (Method lines of code) metric ranges from 3.35 (fitlibraryforfitnesse) to 23.4 (jparse) and the overall average is 7.88 ± 2.7.

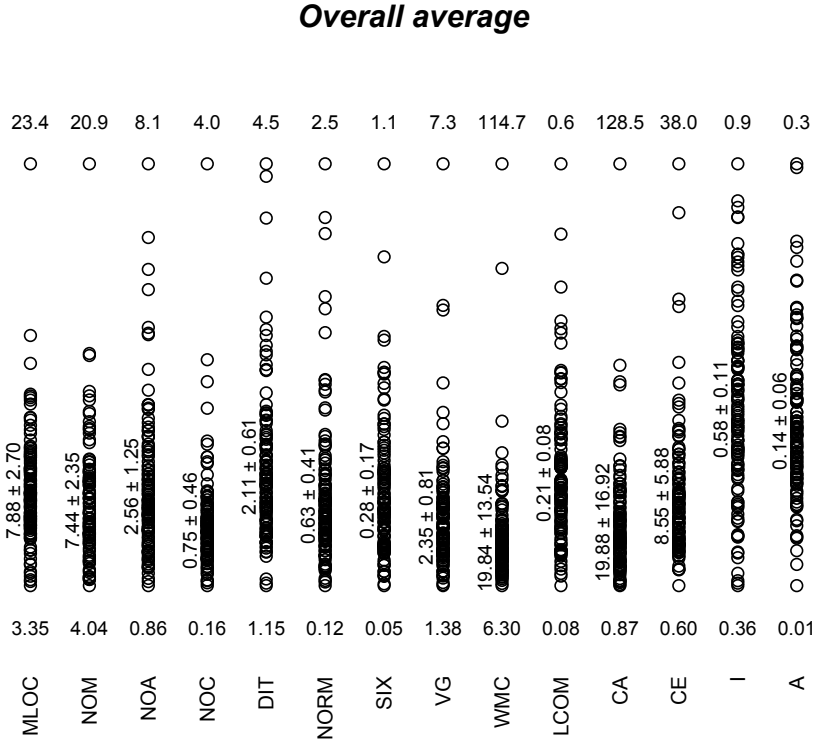


Figure 4.1: Distribution of the metric values in systems from the Qualitas.*class* Corpus (average values)

[3] Comments like `//Qualitas.class:...` to be easily identified.

[4] We used average (arithmetic mean) for illustrative purposes, but it is worth noting that the data do not follow a normal distribution.

4.2 Empirical Study on the Module Suitability

Similarity coefficient measures the degree of correspondence between two entities according to an established criterion. This concept is widely used in software engineering for several different purposes, such as identification of refactoring opportunities [108, 109, 92] and system remodularizations [73, 30]. Particularly in this thesis, we employ the similarity concept in our heuristic to determine the most *suitable* package for a class (which may lead to a *Move Class* refactoring) and the most *suitable* class for a method (which may lead to a *Move Method* refactoring), as computed by the `suitable_module` function.

Although the literature is prolific and reports several similarity coefficients that vary on their computing methods, there are few works comparing similarity coefficients using structural dependencies as source of information [73]. This lack of knowledge may lead researchers to choose an inadequate coefficient to this purpose, once there is a tendency among researchers to make habitual use of certain coefficients that others in their field are using, even without sound scientific or empirical reasons [83].

In order to make our approach as accurate as possible, we conducted a quantitative study that compares 18 coefficients and four different strategies to identify which one is the most appropriate in determining where a source code entity should be located. In short, we measured the similarity between classes and packages of 111 open source systems from the Qualitas.*class* Corpus to evaluate which configuration achieves the highest precision values. Although we have not evaluated the similarity between methods and classes, we assumed that relations *class/package* and *method/class* are quite similar w.r.t. structural dependencies. From our results, we can point out three main findings:

1. Structural dependencies are indeed precise enough to determine where a source code entity should be located. In our evaluation, we achieved an overall precision of 80% in indicating the correct package of a class up to rank 3.

2. Considering the dependency type or the multiplicity of dependencies does not improve the overall precision. Our results show that simply relying on the existence of dependencies between two entities—i.e., without considering the dependency type and the number of dependency instances—achieves the best precision results.

3. *Jaccard*—one of the most used coefficients—has not presented the best results. While *Jaccard* indicated the correct package to only 22% of the classes, other coefficients—such as *Relative Matching*, *Kulczynski*, and *Russell and Rao*—were able to achieve a precision slightly over 60%.

These findings were crucial to our decision to implement the `suitable_module` function based on: (i) structural dependencies as source of information; (ii) *Relative Matching* as the coefficient to measure the similarity between source code entities (both *class/package* and *method/class*); and (iii) sets (and not for example multisets) to represent the types that a source code entity establishes dependency with.

The remainder of this study is structured as follows. Section 4.2.1 provides a description of similarity coefficients. Section 4.2.2 describes strategies for extracting structural dependencies from a class. Section 4.2.3 presents and discusses results on comparing our coefficients and strategies in 111 real-world systems of the Qualitas.*class* Corpus.

4.2.1 Similarity Coefficients

Table 4.2 shows 18 similarity coefficients that we have evaluated to determine the most appropriated one in the context of measuring similarity among classes [83, 28]. To calculate these coefficients, we assume that a given source code entity (method, class, or package) is represented by the dependencies it establishes with other types. Therefore, the measure of the structural similarity between two source code entities i and j (i.e., S_{ij}) considers the following variables:

$\boldsymbol{a} = |Deps(i) \cap Deps(j)|$ = the number of dependencies on both entities,
$\boldsymbol{b} = |Deps(i) \setminus Deps(j)|$ = the number of dependencies on entity $\boldsymbol{i}$ only,
$\boldsymbol{c} = |Deps(j) \setminus Deps(i)|$ = the number of dependencies on entity $\boldsymbol{j}$ only, and
$\boldsymbol{d} = |\,\overline{Deps(i) \cup Deps(j)}\,|$ = the number of dependencies on neither of the entities.

where $Deps(E)$ is the function that returns the set of types that the source code entity E establishes dependency with.

For instance, *Jaccard*—one of the simplest and most used coefficient in our field—is defined by:

$$S_{ij} = \frac{a}{a+b+c} \tag{4.1}$$

Basically, *Jaccard* indicates maximum similarity when two entities have identical dependencies, i.e., when $b = c = 0$ and thus $S_{ij} = 1.0$. On the other hand, it indicates minimum similarity when there are no dependencies in common, i.e., when $a = 0$ and thus $S_{ij} = 0.0$.

Table 4.2: General Purpose Similarity Coefficients

Coefficient	Definition S_{ij}	Range
1. Jaccard	$a/(a+b+c)$	0–1*
2. Simple matching	$(a+d)/(a+b+c+d)$	0–1*
3. Yule	$(ad-bc)/(ad+bc)$	-1–1*
4. Hamann	$[(a+d)-(b+c)]/[(a+d)+(b+c)]$	-1–1*
5. Sorenson	$2a/(2a+b+c)$	0–1*
6. Rogers and Tanimoto	$(a+d)/[a+2(b+c)+d]$	0–1*
7. Sokal and Sneath	$2(a+d)/[2(a+d)+b+c]$	0–1*
8. Russell and Rao	$a/(a+b+c+d)$	0–1*
9. Baroni-Urbani and Buser	$[a+(ad)^{\frac{1}{2}}]/[a+b+c+(ad)^{\frac{1}{2}}]$	0–1*
10. Sokal binary distance	$[(b+c)/(a+b+c+d)]^{\frac{1}{2}}$	0*–1
11. Ochiai	$a/[(a+b)(a+c)]^{\frac{1}{2}}$	0–1*
12. Phi	$(ad-bc)/[(a+b)(a+c)(b+d)(c+d)]^{\frac{1}{2}}$	-1–1*
13. PSC	$a^2/[(b+a)(c+a)]$	0–1*
14. Dot-product	$a/(b+c+2a)$	0–1*
15. Kulczynski	$\frac{1}{2}[a/(a+b)+a/(a+c)]$	0–1*
16. Sokal and Sneath 2	$a/[a+2(b+c)]$	0–1*
17. Sokal and Sneath 4	$\frac{1}{4}[a/(a+b)+a/(a+c)+d/(b+d)+d/(c+d)]$	0–1*
18. Relative Matching	$[a+(ad)^{\frac{1}{2}}]/[a+b+c+d+(ad)^{\frac{1}{2}}]$	0–1*

The symbol "$*$" denotes the maximum similarity.

As an illustrative example, Code 4.1 presents two hypothetical classes. In order to measure the similarity between `Bar` and `Foo`, we first determine the value of the variables a, b, c, and d. In this example: $a = 3$ since both classes rely on `A`, `B`, and `X`; $b = 1$ since only `Bar` relies on `D`; $c = 2$ since only `Foo` relies on `E` and `G`; and $d = 3$ since none establishes dependencies with the three other classes of the system (namely `C`, `F`, and `Y`). For example, the similarity between `Bar` and `Foo` using *Jaccard* results in 0.5, whereas using *Phi* decreases to 0.35 or using *Kulczynski* increases to 0.675.

```
class Bar extends X {
    A a;
    B b;

    exampleBar(D d){
        a.f();
        d.g();
    }
}
```

```
class Foo extends X {
    B b;
    G g;

    exampleFoo(E e){
        e.j();
        new A().f()
    }
}
```

Code 4.1: Hypothetical classes to explain the measurement of similarity

Each coefficient has unique properties that distinguish it from the others. For example, while *Jaccard* does not consider what both entities do not have in order to compute their similarity (variable d), *Simple matching* and ten other coefficients contemplate this variable. The *Yule* and *Hamann* coefficients are mathematically related.

Although both have the same variables in their numerators and denominators, *Yule* relates them by multiplication whereas *Hamann* relates the variable by addition.

As other examples, *Sorenson*[5] gives twice the weight to what the entities have in common (variable a), while the *Rogers and Tanimoto* coefficient gives twice the weight to what each entity has independently (variables b and c). Except for the variable d in the denominator, *Russell and Rao* resembles *Jaccard*. On the other hand, the *Sokal and Sneath* coefficient—which is quite similar to *Simple matching*—reduces the importance of what each entity has independently (variables b and c) by half.

Kulczynski and *Sokal and Sneath 4* are based on conditional probability. *Kulczynski* assumes that a characteristic is present in one item, given that it is present in the other, whereas the *Sokal and Sneath 4* coefficient assumes that a characteristic in one item matches the value in the other.

Finally, *Relative Matching* considers a set of similarity properties such as no mismatch (S_{ij} tends to 1 for b and c close to 0), minimum match (S_{ij} tends to 0 for a and $d = 0$ or close to 0), no match ($S_{ij} = 0$ for $a = 0$), complete match ($S_{ij} = 1$ for $a = |U|$, where U represents the universal set), and maximum match (S_{ij} tends to 1 for $a + d$ tending to $|U|$, $a \neq 0$, and the higher the a, the higher the S_{ij}).

4.2.2 Strategies

In order to measure the similarity between two source code entities, we assume that a given source code entity (method, class, or package) is represented by the *structural dependencies* it establishes with other types. We also distinguish the type of the dependency, i.e., whether a given dependency was established by accessing methods and fields (`access`), declaring variables (`declare`), creating objects (`create`), extending classes (`extend`), implementing interfaces (`implement`), throwing exceptions (`throw`), or using annotations (`useannotation`). Structural dependencies are extracted from the source code using a function named $Deps(E, S)$. Basically, this function returns `E`'s dependencies according to a strategy `S`. A strategy is a pair $[\texttt{C}, \texttt{D}]$ that defines the *collection*[6] and the *dependency information* to be employed in the extraction. The collection `C` can assume one of the following values:

1. *set*: a collection that contains no duplicated elements. For example, if a class establishes more than one dependency to `java.util.Date`, it considers only one.

[5] *Sorenson* is also referred in the literature as *Czekanowski* or *Dice*.

[6] We use the generic term "*collection*" when we do not want to be specific about the kind of structure (set or multiset) under consideration.

2. *multiset*: a generalization of the notion of set in which elements are allowed to appear more than once. For instance, if a class establishes three dependencies to `java.util.Date`, we actually consider all of them.

The dependency information `D` can assume one of the following values:

1. *target type* (*tt*): in this case the extraction function returns a collection of target types that the entity establishes dependencies with. Thus, an element is a single [`T`], denoting the existence of at least one dependency between the entity under analysis and `T`.

2. *target and dependency type* (*dtt*): in this case the extraction function returns a collection whose elements are pairs [`dt`, `T`], denoting the existence of a dependency of type `dt` between the class under analysis and `T`.

```
public class Bar {
    public void foo (Date d){
        if (d == null){
            d = new Date();
        } else {
            new Date()
        }
    }
}
```

Code 4.2: An example class that explains our strategies

As an illustrative example, consider the class presented in Code 4.2. The collection returned by function *Deps*(`Bar`, `S`) differs according to the employed strategy `S`. More specifically, the following calls (and respective results) are possible:

$Deps($`Bar`$, [$`set`$, $`tt`$]) = \{$ `Date` $\}$

$Deps($`Bar`$, [$`mset`$, $`tt`$]) = ($ {`Date`}, {(`Date`,3)} $)$

$Deps($`Bar`$, [$`set`$, $`dtt`$]) = \{$ [`declare`, `Date`], [`create`, `Date`] $\}$

$Deps($`Bar`$, [$`mset`$, $`dtt`$]) = ($ {[`declare`, `Date`], [`create`, `Date`]}, {([`declare`,`Date`],1),([`create`,`Date`],2)} $)$

Strategies that rely on *target and dependency type* (`dtt`) may be particularly important when the classes of the system rely on types differently according to their location. For example, a factory method that creates a `DTO` (Data Transfer Object) and a logic presentation method that handles a `DTO` may not be similar. As another example, a class that implements `java.io.Serializable` and a method that declares

a variable of type `java.io.Serializable` may also not be similar. Although this strategy might perform better in particular cases, our evaluation is concerned with the overall precision.

Last but not least, the set of dependencies of a package `Pkg` is calculated by the union of the set of the dependencies of its classes as follows:

$$Deps(\texttt{Pkg}, \texttt{S}) = \bigcup_{C_i \in Pkg} Deps(\texttt{C}_\texttt{i}, \texttt{S})$$

4.2.3 Evaluation

4.2.3.1 Research Questions

We designed a study to address the following overarching research questions:

RQ #1 – Are structural dependencies precise enough to indicate whether a class is located in its correct package?

RQ #2 – Considering the multiplicity of dependencies—i.e., a multiset rather than a set—improves the overall precision?

RQ #3 – Considering the dependency type—i.e., representing a target dependency as a pair $[\texttt{dt}, \texttt{T}]$ rather than only a single type $[\texttt{T}]$—improves the overall precision?

RQ #4 – Which coefficient is the most suitable to measure the similarity among classes of object-oriented systems?

4.2.3.2 Target Systems

As described in Section 4.1, our evaluation relies on the Qualitas.*class* Corpus, which is a compiled version of the Qualitas Corpus proposed by Tempero et al. [97]. The corpus includes many popular systems, such as JRE, Eclipse, NetBeans, and Apache Tomcat. It is worth noting that the corpus is large and heterogeneous, ranging from small frameworks and text processors to complete IDEs and virtual machines.

4.2.3.3 Major Assumption

We made the following assumption due to the infeasibility of obtaining a 100% accurate oracle for thousands of classes:

> "*We assume that every class under analysis is in its right package.*"

Therefore, similarity coefficients should indicate the *current* package of the class as its *most suitable one*.

Risks of the Assumption: Although our assumption is likely to be false, we argue that such assumption only favors our results when the following two unlikely conditions hold for a class C implemented in a package Pkg: (a) C is mistakenly implemented in the package Pkg; and (b) Pkg is exactly the package the similarity coefficient indicates as most suitable one. For example, when condition (a) holds, but condition (b) does not hold, our results are not favored. On the other hand, despite having a probability of misplaced classes, these classes in our corpus affect equally the results, making at least our comparison of the similarity coefficients and strategies fair.

4.2.3.4 Methodology

To provide answers to our research questions, we performed the following tasks:

1. *Data Extraction*: For the 111 projects provided by the Qualitas.*class* Corpus, we first extracted the structural dependencies of classes using the four strategies described in Section 4.2.2, which are [`set`, `tt`], [`set`, `dtt`], [`mset`, `tt`], and [`mset`, `dtt`].

2. *Comparative Analysis*: Second, we measured the similarity using all coefficients described in Section 4.2.1. The coefficients were applied to measure the similarity between pairs [`class`, `package`] from our corpus.

3. *Qualitative Analysis*: Last, we conducted a qualitative discussion in order to answer our research questions.

4.2.3.5 Experimental Setup

In order to conduct this study, the following policies have been proposed:

1. We have not considered the class under analysis while searching for its right location. For example, when measuring the similarity between a class `A` and its package `Pkg`, we actually considered its own package `Pkg` as being $\texttt{Pkg} - \{\texttt{A}\}$.

   Thereupon, we have not sought the suitable location of classes whose package contains a single class. For example, assume that package `Pkg` contains only class `A`. It would be unlikely to indicate package `Pkg` as the correct package for class `A`, since $Deps(\texttt{Pkg} - \{\texttt{A}\}, \texttt{S}) = \phi$.

2. We have not considered a class $\texttt{C}_\texttt{i}$ when $|Deps(\texttt{C}_\texttt{i}, \texttt{S})| < 5$, i.e., we have not evaluated classes that establish less than five dependencies. These classes contain too little information to make any inference based on their structural dependencies.

3. We have not evaluated test classes, since most systems organize their test classes in a single package. Consequently, the test package contains classes related to different components of the system—i.e., they are not structurally related—which certainly reduces the precision of any approach based on structural dependencies.

4. We have filtered trivial dependencies, such as those established with primitive and wrappers types (e.g., `int` and `java.lang.Integer`), `java.lang.String`, and `java.lang.Object`. Since virtually all classes establish dependencies with these types, they do not actually contribute for the measure of similarity. This decision is quite similar to text retrieval systems that exclude stop words because they are rarely helpful in describing the content of a document [6].

4.2.3.6 Results

Figure 4.2 illustrates the overall precision for each coefficient regarding the four analyzed strategies. The overall precision is defined by the ratio between the total number of analyzed classes that have their location (package) correctly predicted by the similarity coefficient and the total number of analyzed classes. We also provided the Top 1, 2, and 3 ranking, which stands for the position of the correct package of a class. As an example, considering strategy [`set`, `tt`], the *Relative Matching* precision has reached 60% on Top 1, 72% on Top 2, and 78% on Top 3. In other words, it means that *Relative Matching* located the correct package of a class 60% on the first position of its ranking, 12% on the second position, and 6% on the third position.

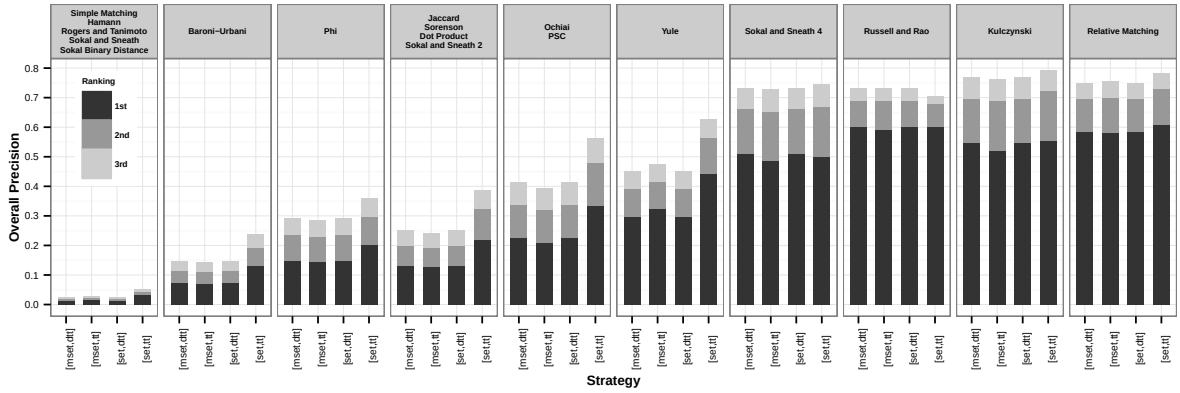


Figure 4.2: Top 3 ranking of similarity coefficients using all strategies

Before providing answers to our research questions, it is worth noting that many similarity coefficients presented very similar (mostly identical) results. In fact, the Spearman correlation among these coefficients was very close to 1, which allowed us to group them. The multiple correlation among *Simple Matching*, *Hamann*, *Rogers and*

Tanimoto, *Sokal and Sneath*, and *Sokal Binary Distance* presented lowest correlation value of 0.999994. Similarly, *Jaccard*, *Sorenson*, *Dot-product*, and *Sokal and Sneath 2* presented lowest correlation value of 0.999999. Finally, *Ochiai* and *PSC* presented correlation equal to 0.998251. These results explain why there is no variance in the ranks within the same group.

Next, we answer our research questions based mainly on Figure 4.2. Our data interpretation always considers the Top 3 ranking—when not stated differently.

RQ #1: *Are structural dependencies precise enough to indicate whether a class is located in its correct package?*

Yes. As can be observed in Figure 4.2, there are coefficients that achieved a high precision when determining the package where a class should be located. In particular, *Relative Matching*, *Kulczynski*, *Russell and Rao*, and *Sokal and Sneath 4* achieved, in the worst scenario, over than 70% of precision.

RQ #2: *Considering the multiplicity of dependencies—i.e., a multiset rather than a set—improves the overall precision?*

No. Figure 4.2 shows that strategies that use traditional sets ([`set`, `tt`] and [`set`, `dtt`]) perform better than equivalent multiset-based strategies for all coefficients. The only exception is the *Russell And Rao* coefficient, which presented results slightly better for the [`mset`, `tt`] strategy. More important, if we consider only Top 1, a traditional set-based strategy performs better than multiset-based one for all coefficients.

RQ #3 – *Considering the dependency type—i.e., representing a target dependency as a pair* [`dt`, `T`] *rather than only a single type* [`T`]*—improves the overall precision?*

No. On the one hand, analyzing set data, we can observe that [`set`, `tt`] provides better results for all coefficients, except for *Russell and Rao* and *Sokal and Sneath 4* that presented results slightly better using the dependency type ([`set`, `dtt`]).

On the other hand, we notice that multiset-based data (i.e., [`mset`, `tt`] and [`mset`, `dtt`]) presents very similar results for all coefficients. This fact is expected since the extracted collections are usually very similar. For instance, assume a collection `A` extracted using strategy [`mset`, `dtt`] and a collection `B` extracted using strategy [`mset`, `tt`], for the same class. If `A(i)` = [`access`, `Foo`] for an index `i`, then `B(i)` = [`Foo`].

From now on, our discussion considers only strategy [`set`, `tt`], since we showed that the use of multiset (`mset`) and dependency type (`dtt`) does not actually improve the overall precision.

RQ #4: *Which coefficient is the most suitable to measure the similarity among classes of object-oriented systems?*

Relative Matching*, *Kulczynski*, and *Russell and Rao. These coefficients reached the highest precision values in our study. As can be observed in Figure 4.2, *Relative Matching* (60.83%) and *Russell and Rao* (60.27%) achieved the highest similarity values of the Top 1, and *Relative Matching* (72.78% and 78.18%) and *Kulczynski* (72.15% and 79.23%) of the Top 2 and 3, respectively.

As the central finding of our study, these three coefficients significantly outperform *Jaccard*—one of the most used similarity coefficients. While *Jaccard* indicated the correct package to 22% (Top 1) and 39% (Top 3) of the classes, *Relative Matching*, *Kulczynski* and *Russell and Rao* were able to indicate the correct package to 60% (Top 1) and 79% (Top 3).

To better explain this behavior, we anecdotally analyzed some systems to understand the influence of the variables a, b, c, and d (see Section 4.2.1) on the precision measurements. We performed this analysis by plotting each variable against the ranking. Our major finding regards to the fact that large packages negatively influence *Jaccard* and other coefficients that presented very low precision (e.g., *Simple Matching*). Usually, large packages imply a relevant difference between c and d, which negatively impacts the precision of coefficients like *Jaccard*. On the other hand, by their nature, this scenario does not influence *Relative Matching*, *Kulczynski*, and *Russell and Rao*. It explains why these coefficients presented the best results on systems that contain either small or large packages.

4.2.3.7 Supplementary Results

Figure 4.3 illustrates the Top N ranking of every coefficient using strategy [`set`, `tt`]. In contrast to Figure 4.2 that displays only the Top 3, it displays the full distribution of the ranks until full coverage (i.e., precision of 1.0). As a second relevant finding from our study, Figure 4.3 shows that there is no coefficient that drastically improves its precision right after the top 3 ranking (e.g., Top 4 or Top 5). This behavior reinforces our decision in using Top 3.

As can also be observed in Figure 4.3, *Russell and Rao* achieved precision of 1.0 in the rank 79. It means that the suitable package of a class was detected, in the worst case, in the 79th position. This result is quite relevant, since the other coefficients only achieved precision of 1.0 from the rank 347.

Since our analysis so far has considered the overall precision, we also analyzed the results of each coefficient per system. Figure 4.4 summarizes the number of systems

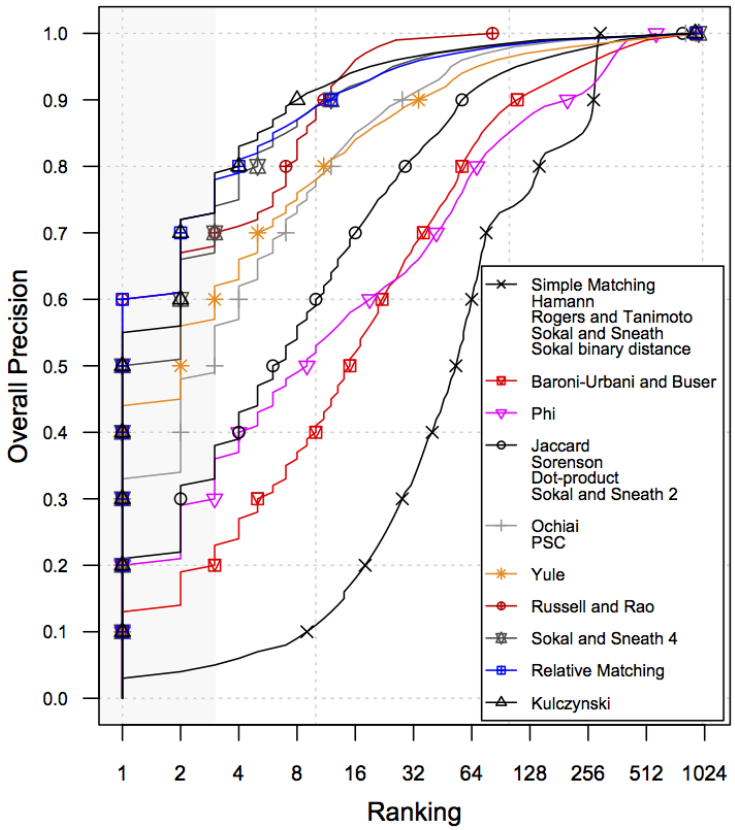


Figure 4.3: General ranking using strategy [`set`, `tt`]

in which a particular coefficient has presented the best result (i.e., better identified the correct package of a class). For instance, *Relative matching* has better determined correct modules to *Eclipse* classes, whereas the *Russell and Rao* coefficient has behaved better to *ArgoUML* classes. Furthermore, we have not presented some coefficients (e.g., *Simple Matching* and *Jaccard*) because they have not presented the best result for any system.

As can be observed in Figure 4.4, *Relative Matching*, *Kulczynski*, and *Russell and Rao* presented the best results for most systems, which reinforces our claim that these coefficients are the most suitable ones to assess the similarity among classes in object-oriented systems.

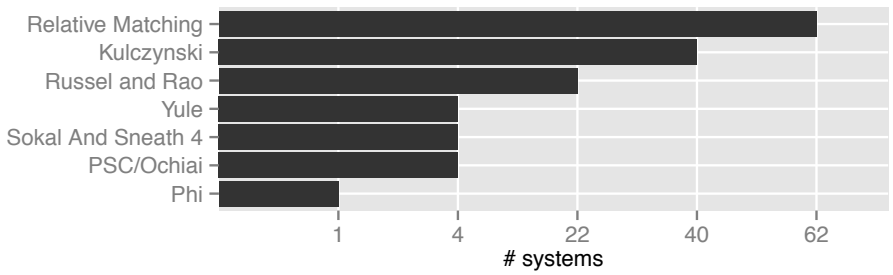


Figure 4.4: # systems in which a particular coefficient presented the best result

4.2.3.8 Threats to Validity

In this section, we identify and classify threats to validity in our evaluation [20, 113]:

Conclusion Validity: First, our study assumes that every class is in its right location (i.e., package). Although there might be misplaced classes, we argue that, besides we considered a stable and trustworthy collection of systems, these misplaced classes can lead to a bias on the results in rare circumstances, as already discussed in Section 4.2.3.3. However, we acknowledge that we need to assess the impact of misplaced classes in our corpus or even to replicate our experiment in a more reliable dataset (future work). Second, our analysis is based only on the overall precision, as defined in Section 4.2.3.6. Although the distribution of the precision per system could provide more insights, we argue that the overall precision supports the main objective of our study, which is to indicate the best coefficient and strategy to be employed by our suitable module heuristic.

External Validity: We must state a particular threat that might impact the approach proposed in this thesis. Although the relations *class/package* and *method/class* seem similar w.r.t. structural dependencies, we cannot claim that an empirical study assessing the suitable class for a method—instead of the suitable package for a class, as conducted—will provide equivalent results.

4.3 Final Remarks

The approach proposed in this thesis relies on the structural similarity between source code entities to recommend *Move Method* and *Move Class* refactorings, as computed by the `suitable_module` function. Particularly, we take the position that the choice of a similarity coefficient should not be made without well-founded reasons, as usually occur [83]. Therefore, we conducted a quantitative study that compares 18 similarity coefficients and four strategies to identify which one is the most appropriate in determining where a class should be located.

We observed that *Jaccard*—one of the most used coefficients by software engineering tools—has not presented the best results. Particularly, large packages imply a large difference between variables c and d, which negatively impacts the precision of coefficients like *Jaccard*, but does not affect other coefficients. While *Jaccard* indicated the correct package to only 22% of the classes, other coefficients—such as *Relative Matching*, *Kulczynski*, and *Russell and Rao*—achieved a precision slightly over 60%.

Moreover, we also observed that the simplest strategy to extract structural dependencies from a class—set with only types ([`set`, `tt`])—has presented the best results.

These findings led us: (i) to choose *Relative Matching* as the coefficient to measure the similarity between source code entities (both *class/package* and *method/class*), since it presented the best overall precision (60.93% on Top 1) and the best result for most systems (62 out of 111); and (ii) to adopt the strategy [`set`, `tt`], since we evidenced that neither multisets (`mset`) or fine-grained dependency types (`dtt`) improves the overall precision.

The study described in this chapter was based on the Qualitas.*class* Corpus, which is a compiled variant of the Qualitas Corpus proposed by Tempero et al. [97]. The Qualitas.*class* Corpus is publicly available at:

http://aserg.labsoft.dcc.ufmg.br/qualitas.class

In the next chapter, we describe the evaluation of our recommendation system. Basically, we present and discuss results on applying our repairing recommendations in two real-world systems.

Chapter 5

Evaluation of the Recommendation System

In this chapter, we present and discuss results on applying the approach proposed in this thesis in two real-world systems. The main goal is to demonstrate the applicability of our approach in real development contexts through an analysis of the correctness and complexity of our repairing recommendations.

We organized this chapter as follows. Section 5.1 states our research questions. Section 5.2 describes the two real-world systems our evaluation is based on. Section 5.3 presents the methodology we adopted. Sections 5.4 and 5.5 present the results on applying our repairing recommendations in our target systems. Section 5.6 provides a qualitative discussion on the results and Section 5.7 enumerates the lessons learned. Finally, Section 5.8 points the threats to validity and Section 5.9 concludes this chapter with a general discussion.

5.1 Research Questions

We designed a study to address the following research questions:

RQ #1 – For *what portion* of architectural violations detected in a real-world system can the proposed approach provide repairing recommendations?

RQ #2 – Does the proposed approach provide *correct* recommendations for repairing architectural violations?

RQ #3 – How *complex* is to discover a *correct* repair action without the support of our recommendation system?

RQ #4 – How *complex* is to reject an *incorrect* repair action provided by our recommendation system?

The strategies we follow to assert *correctness* (*RQ #2*) and to measure *complexity* (*RQ #3 and RQ #4*) are presented in the Methodology section (respectively in Sections 5.3.2 and 5.3.3).

5.2 Target Systems

Our evaluation relies on two Java web-based systems. The first one is a 21 KLOC open-source strategic management system, called Geplanes.[1] The system handles strategic management activities, including management plans, goals, performance indicators, actions, etc. The second one is a large and complex customer care platform of a major Brazilian telecommunication company. Due to a non-disclosure agreement, we will omit the company's name in this thesis and will refer to this second system just as BrTCom. The system has 728 KLOC and it handles a full range of customer related services, including account activation, claims and inquiries, offers, etc. Table 5.1 shows information on the size of both systems.

Table 5.1: Target systems used in the evaluation

	Geplanes	**BrTCom**
LOC	21,799	728,814
Subsystems	1	146
Packages	25	2,289
Classes	278	4,724
Interfaces	1	1,893
External libraries	47	58

5.3 Methodology

To provide answers to our research questions, we performed the following major steps:

5.3.1 Triggering Recommendations

As illustrated in Figure 5.1, the chief architect of each system first defined the architectural constraints based on an existing high-level model of their systems. Since the

[1] http://www.softwarepublico.gov.br

constraints were provided in natural language, we translated the informal definitions to DCL and validated them with the architects in a 30-minute individual meeting.

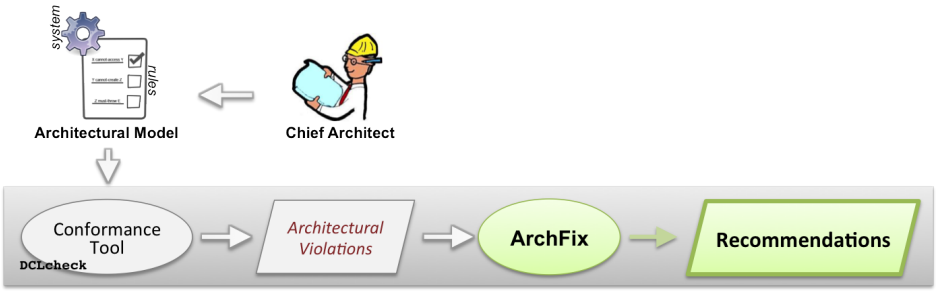


Figure 5.1: Methodology followed in the evaluation

Next, we executed the `DCLcheck` tool to detect violations in both systems. A second meeting was scheduled with the architects to validate the architectural violations raised by `DCLcheck`. Although Geplanes' meeting lasted 25 minutes, BrTCom's meeting lasted almost one hour due to the considerable number of violations unrecognized by the architect. This additional time was necessary to refine the initial definitions of modules and to disregard violations that in fact represent exceptions to general rules or that are not relevant (e.g., violations detected in test classes). Finally, we executed `ArchFix` to provide repairing recommendations for the true violations asserted by the architects.

It is important to mention that when this evaluation was conducted, `ArchFix` was limited to trigger a single repairing recommendation, specifically the first one that fits. In fact, our approach has been recently improved to trigger multiple recommendations, as detailed in Section 3.3.4 (page 33). Since the architects involved in this study were no longer available, this study does not contemplate an evaluation of the prioritization policy, which is listed as future work in Section 6.3.

5.3.2 Correctness Evaluation

We showed the recommendations to the chief architect of each system who classified them as *correct*, *partially correct*, or *incorrect*. We instructed the architects to classify a recommendation as *correct* when it is the appropriate solution to the detected violation (e.g., a violation whose fixing involves replacing an instantiation with the respective factory method and our approach has precisely suggested this repair action), as *incorrect* when it is definitely not part of the architectural fix (e.g., a violation whose fixing involves making the class implement a particular interface, but our approach has suggested moving the class to another module), and as *partially correct* when the

recommendation is only part of the required repair action (e.g., a violation whose fixing involves replacing an annotation with a new one, but our approach has only suggested inserting the new annotation, without a suggestion to remove the existing one).

A third meeting was scheduled with the architects to evaluate the correctness of the triggered recommendations. Since Geplanes triggered few violations (only 41 in total), this classification was possible during a 30-minute meeting. For each recommendation, we reminded the architect of the constraint, presented the code responsible by the violation, and the recommendation triggered by our approach. For BrTCom, due to the large number of detected violations in this system (787 in total), we grouped similar recommendations before the meeting and asked the architect to classify only a subset of the recommendations in a given group. For instance, a single constraint (named `TC11`) raised 270 violations due to forbidden declarations. Since the violations were very similar, we randomly selected six to be scored by the architect. In this way, in two 30-minute meetings with the architect, it was possible to classify 89 recommendations that represent the whole set of recommendations triggered for the BrTCom system.

5.3.3 Complexity Evaluation

We conducted a 30-minute meeting with the Geplanes' architect to assess the complexity of all triggered recommendations. In the case of BrTCom, similarly to the correctness evaluation, we asked the architect to assess the complexity of the subset of 89 recommendations in a single 50-minute meeting.

During these meetings, for each correct recommendation, we asked the architects to assess how complex would be for a typical developer to discover the suggested repair actions without the support of our recommendation system. More specifically, we instructed the architects to consider the scope of the classes that might be inspected by the developers as the most relevant property to assess this complexity. We defined three levels of complexity: *minor* (when discovering the correct repair actions does not require inspecting other classes), *moderate* (when discovering the correct repair actions might require inspecting classes located in well-known modules), or *major* (when discovering the correct repair actions might require inspecting classes whose location is not known a priori).

As an example of a recommendation with *minor* complexity, assume a violation in which a `View` class misses a required annotation and our approach correctly suggests adding the annotation. In this case, a typical developer can conclude after a local inspection that the class is indeed a `View` class and requires the annotation. On the

other hand, as an example of a recommendation with *moderate* complexity, assume a violation in which an object of a `Product` class is created in a module that is not allowed to perform this operation and our approach correctly suggests replacing the direct instantiation with the respective factory method. In this case, a typical developer needs to inspect other classes in the current module or even in the `Factory` module to find the appropriate factory method. Finally, as an example of a recommendation with *major* complexity, assume a violation in which a class is located in the wrong module and our approach correctly suggests moving the class to another module. In this case, a typical developer may need to perform a system-wide inspection to determine the most suitable module.

Finally, we also asked the architects to assess how complex would be for a typical developer to discover that a given recommendation is incorrect. In this case, we also relied on the scope of the classes that might be inspected by the developers to make the decision.

5.4 Geplanes Results

The results achieved by applying our methodology to Geplanes are discussed next.

Recommendations (RQ #1): Table 5.2 lists the architectural constraints prescribed by the Geplanes' chief architect. Figure 5.2 illustrates such constraints and the detected architectural violations in the form of a reflexion model [68].[2] They are mainly related to the MVC-based framework used by Geplanes' current implementation. In general, constraints `GP1`−`GP7` require that classes from some modules receive particular annotations, constraints `GP8`−`GP9` restrict the modules that are allowed to receive particular annotations, and constraints `GP10`−`GP11` forbid some modules to create classes of specific modules. As reported in Table 5.2, we found 41 architectural violations and `ArchFix` was able to provide recommendations for all of them (100%).

Correctness (RQ #2): Table 5.2 also presents the results of the correctness classification, according to Geplanes' architect. As reported, 31 recommendations were classified as *correct* (75%), eight recommendations were classified as *partially correct* (20%), and two recommendations were classified as *incorrect* (5%).

As reported in Table 5.2, most violations are related to constraints `GP1`−`GP7`. In such cases, the usual recommendation was adding the required annotation to the class

[2]We are using this reflexion model just for illustrative purposes. In fact, as mentioned in Section 5.3.1, we relied on DCL to detect architectural violations in our target systems.

Table 5.2: Recommendations and correctness evaluation (Geplanes)

Violations		Number	Recommendations (correct – pr. cor. – incor.)	Total
GP1	[Entities, must, useannotation, javax.persistence.Entity]	3	A5(1–0–1); A6(1–0–0)	2–0–1
GP2	[Entities, must, useannotation, javax.persistence.Id]	1	A8(1–0–0)	1–0–0
GP3	[Entities, must, useannotation, javax.persistence.GeneratedValue]	1	A8(1–0–0)	1–0–0
GP4	[Entities, must, useannotation, linkcom.neo.bean.DescriptionProperty]	18	A6(18–0–0)	18–0–0
GP5	[Controllers, must, useannotation, linkcom.neo.controller.DefaultAction]	1	A6(1–0–0)	1–0–0
GP6	[Controllers, must, useannotation, linkcom.neo.controller.Controller]	2	A6(1–0–1)	1–0–1
GP7	[Services, must, useannotation, linkcom.neo.bean.ServiceBean]	1	A6(1–0–0)	1–0–0
GP8	[Entities, cannot, useannotation, javax.persistence.Entity]	1	D21(1–0–0)	1–0–0
GP9	[Controllers, cannot, useannotation, linkcom.neo.controller.Input]	1	D24(1–0–0)	1–0–0
GP10	[System, cannot, create, [Services, DAOs, Controllers]]	5	D12(2–3–0)	2–3–0
GP11	[DAOs, cannot, create, linkcom.neo.persistence.QueryBuilder]	7	D11(2–0–0); D13(0–5–0)	2–5–0
		41		**31–8–2**

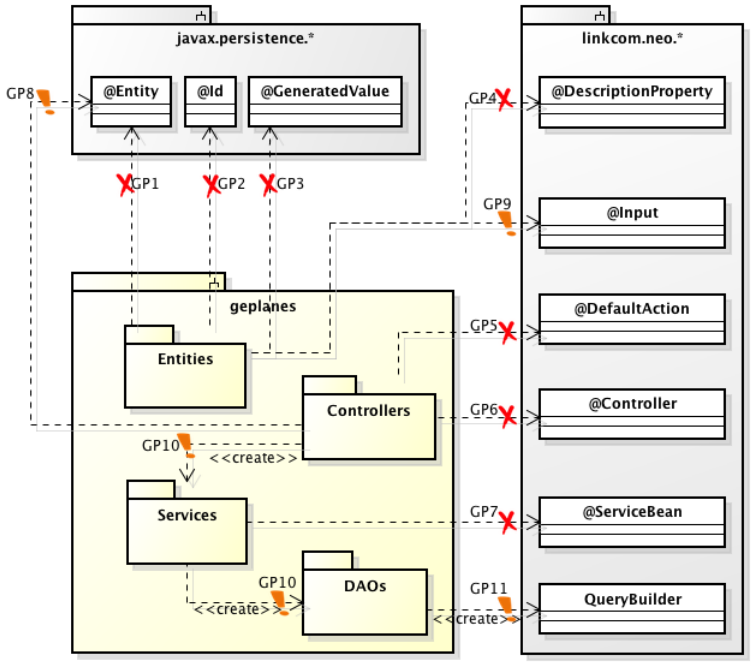


Figure 5.2: Geplanes' reflexion model (an 'x' denotes absences and a '!' denotes divergences)

or method where the violation was detected (recs. $A6$ and $A8$). For example, as can be observed in Code 5.1, class `AnomaliaCrud` is missing annotation `@Controller` required by the underlying MVC-based framework. Therefore, this absence represents a violation of constraint `GP6`, as illustrated by an arrow with an 'x' mark from `Controllers` to `@Controller` in Figure 5.2. In this particular case, our approach correctly suggested adding annotation `@Controller` (rec. $A6$) to the `AnomaliaCrud` class.

```
1  package br.com.linkcom.sgm.controller.crud;
2
3  public class AnomaliaCrud extends
4                          SGMCrudController<AnomaliaFiltro, Anomalia, Anomalia> {
5      private FerramentaAnomaliaService fas = new FerramentaAnomaliaService();
6      ...
7  }
```

Code 5.1: Controller class `AnomaliaCrud`

`ArchFix` also suggested removing instantiations of objects that are provided by dependency injection techniques (rec. $D12$). For example, the `AnomaliaCrud` class (Code 5.1) creates an object of `FerramentaAnomaliaService` (line 5). This instantiation represents a divergence regarding constraint `GP10`, as illustrated by an arrow with an '!' mark from `Controllers` to `Services` in Figure 5.2. The recommendation for this particular divergence was regarded as *partially correct* because, besides removing the instantiation, the architect also indicated the need to create a local setter method for the respective attribute `fas`.

As a final example, class `Verbo`—located in module `Controllers`—has an annotation `@Entity`, as can be observed at line 3 in Code 5.2. The use of this annotation represents a divergence regarding constraint `GP8`, as illustrated by an arrow with an '!' mark from `Controllers` to `@Entity` in Figure 5.2. In this case, our approach correctly suggested moving `Verbo` to module `Entities` (rec. $D21$). More specifically, the `suitable_module` function returned `Entities` as the module with the highest similarity—$S(Deps(\texttt{Verbo}), Deps(\texttt{Entities}))$=0.114—, whereas the current module of the class (`Controllers`) has a significant lower similarity—$S(Deps(\texttt{Verbo}), Deps(\texttt{Controllers}))$=0.038.

```
1  package br.com.linkcom.sgm.controller;
2
3  @Entity
4  @SequenceGenerator(name = "sq_verbo", sequenceName = "sq_verbo")
5  public class Verbo {
6      private Integer id;
7      private String nome;
8      ...
9      @Id
10     @GeneratedValue(strategy=GenerationType.AUTO, generator="sq_verbo")
11     public Integer getId() {
12         return id;
13     }
14 }
```

Code 5.2: *"Controller"* class `Verbo`

Complexity (RQ #3, RQ #4): In the case of correct recommendations, two recommendations have been scored as having a *minor* complexity ($A5$ and $A6$), four recommendations ($D11$, $D21$, $D24$, and $A8$) as having a *moderate* complexity, and another recommendation ($D12$) as having a *major* complexity. For example, rec. $A5$ for violations `GP1` was classified as having a *minor* complexity, because it just requires adding an annotation to classes that represent database entities (which can be locally inferred by the presence of other database annotations). On the other hand, rec. $D11$ for violations `GP11` was classified as having a *moderate* complexity, because it requires replacing the direct instantiation of a query builder object with its factory method. In this case, the developer should search for the factory inspecting classes located in the `DAO` module (i.e., a previously known module). Finally, rec. $D12$ for violations `GP10` has been scored as *major* complexity. Although it solely suggests to remove the instantiation of `DataSource` and `Controller` objects, the developer must be aware that such objects are provided by a dependency injection framework.

In the case of incorrect recommendations, rec. $A5$ has been considered as having *moderate* complexity and rec. $A6$ as presenting a *minor* complexity. Rec. $A5$ suggests moving a class without a particular annotation to other module. In this case, the developer must inspect the classes of the target module to realize that the moving is incorrect. On the other side, rec. $A6$ suggests adding a specific annotation and the developer can infer that such annotation is incorrect by considering only the class itself.

5.5 BrTCom Results

The results for BrTCom are discussed next.

Recommendations (RQ #1): Table 5.3 lists the 14 architectural constraints prescribed by the BrTCom's chief architect. These constraints are mainly used to enforce several architectural rules, such as decomposition in layers (e.g., `TC8`), factories (e.g., `TC5`), interfaces (e.g., `TC13`), persistence patterns (e.g., `TC1`), etc. `ArchFix` raised 727 recommendations for the 787 violations we detected using the constraints in Table 5.3 (92%). Regarding the 60 violations without recommendation, 54 violations are associated to `TC14`. Particularly in such violations, `ServerLayer` classes were calling methods that should not be implemented in `ClientUtil` classes but rather in a layer common both to server and client modules.

Table 5.3: Recommendations and correctness evaluation (BrTCom)

Violations		Number	Recommendations (correct – pr. cor. – incor.)	Total
TC1	[DTOs, must, implement, java.io.Serializable]	63	A3(53−0−0); A4(0−0−10)	53−0−10
TC2	[SAOs, must, extend, brtcom.server.sao.AbstractSAO]	1	A4(1−0−0)	1−0−0
TC3	[Controllers, must, useannotation, brtcom.client.controller.Controller]	1	A6(1−0−0)	1−0−0
TC4	[DataSources, must, useannotation, brtcom.client.datasource.DataSource]	1	A6(1−0−0)	1−0−0
TC5	[$\overline{\text{brtcom.server.dao.BaseJPADAO}}$, cannot, create, DAOs]	13	D11(13−0−0)	13−0−0
TC6	[$\overline{\text{DAOs}}$, cannot, throw, brtcom.server.dao.DAOException]	15	D15(11−0−0);D16(2−0−0)	13−0−0
TC7	[$\overline{[\text{CtrlLayer}, \text{DSLayer}]}$, cannot, useannotation, CtrlDSAnnotations]	20	D21(2−0−0); D22(18−0−0)	20−0−0
TC8	[$\overline{[\text{Services}, \text{SAOLayer}]}$, cannot, depend, SAOs]	5	D20(1−0−0)	1−0−0
TC9	[System, cannot, create, [Controllers, DataSources]]	3	D12(3−0−0)	3−0−0
TC10	[CtrlLayer, cannot, create, java.util.Date]	84	D13(0−84−0)	0−84−0
TC11	[ScreenWrappers, cannot, useannotation, JavaLangAnnotations]	18	D21(0−0−1); D22(17−0−0)	17−0−1
TC12	[System, cannot, depend, java.lang.System]	14	D9(14−0−0)	14−0−0
TC13	[ServicesAsync, cannot, declare, UnallowedAbstractTypes]	270	D2(270−0−0)	270−0−0
TC14	[ServerLayer, cannot, depend, ClientUtil]	279	D7(225−0−0)	225−0−0
		787		**632−84−11**

Correctness (RQ #2): As can be observed in Table 5.3, BrTCom's architect has scored 632 recommendations as *correct* (80%), 84 recommendations as *partially correct* (11%), and 11 recommendations as *incorrect* (2%).

For example, constraint `TC1`, which specifies that `DTO` classes must implement `Serializable`, raised violations in 63 classes. For 53 classes, `ArchFix` suggested the correct repair action, i.e., making the class implement `Serializable` (rec. $A3$). Nevertheless, Data Transfer Objects (DTOs) by their very nature rely extensively on types from the Java API. For this reason, `ArchFix`—based on the structural similarity calculated by the `suitable_module` function described in Section 3.3.5 (page 34)—has improperly recommended moving the other 10 classes to the `Constants` module (rec. $A4$), whose classes are also heavily based on Java's built-in types.

The highest number of correct recommendations for a single constraint has been raised for the 270 violations associated to constraint `TC13`, which forbids `ServicesAsync` classes to declare abstract types due to a pattern recommended by the GWT framework. For each of such violations, `ArchFix` suggested a more specialized type (rec. $D2$). For instance, most of the violations were due to references to `List` and `Map` in GWT interfaces. In such cases, `ArchFix` has properly suggested replacing these abstract types with the concrete types `ArrayList` and `HashMap`, respectively.

Complexity (RQ #3, RQ #4): BrTCom's architect assessed the complexity of correct recommendations in the following way: four recommendations as having a *minor* complexity ($D9$, $D22$, $A3$, and $A6$), two recommendations ($D2$ and $D11$) as having a *moderate* complexity, and six recommendations ($D7$, $D12$, $D15$, $D16$, $D20$, and $D21$) as present-

ing a *major* complexity. For example, rec. $D2$ for violations `TC13` was considered as having a *moderate* complexity, since the concrete implementations for an abstract type usually have a well-known location (e.g., the package `java.util`). On the other hand, rec. $D7$ for `Server` classes making an unauthorized use of `Client` services associated to constraint `TC14` was scored as having a *major* complexity, because it is not trivial to delimit the scope of the task of searching for a class that provides equivalent services to the ones provided by another class.

Regarding the incorrect recommendations, one recommendation ($D21$) was considered as having a *major* complexity and another recommendation ($A4$) as presenting a *moderate* complexity. Rec. $D21$ suggests moving an incorrectly annotated class to another module (where the annotation does not represent a violation). The task of discarding it was considered as having a *major* complexity because a global understanding of the system is required to decide whether the indicated module is correct or not.[3]

5.6 Analysis of Results

Based on the experience gained with the Geplanes and BrTCom case studies, we conducted a deeper analysis on the major themes of the detected violations and the root causes of them. Table 5.4 classifies the detected violations according to a set of architectural defect types initially proposed by Knodel and Popescu [50] and later extended by us [102]. As a general fact, the architects ascribed most of the detected violations due to the lack of awareness about the architectural model [110] and *copy-and-paste* procedures [29].

Table 5.4: Classification of the detected violations (focusing on correctness)

Architectural Defect Type	**Constraints**	**# viols.**	**# recs.**	**# correct recs.**	**precision**
Misusage of persistence patterns	`GP1−3; TC1`	68	68	57	0.84
Misusage of domain-specific patterns	`GP4−7; TC2−4`	25	25	24	0.96
Bypassing layers	`GP8−9; TC6−8; TC14`	321	261	261	1.00
Unintended dependencies	`GP10; TC9; TC11−12`	40	40	36	0.90
Bypassing creational patterns	`GP11; TC5`	20	20	15	0.75
Context exploration	`TC10`	84	84	0	0.00
Misusage of interfaces	`TC13`	270	270	270	1.00
		828	768	663	

[3]A similar recommendation ($A5$) was triggered in Geplanes' case study and it was considered as having a *moderate* complexity. Basically, the recommendations involves moving a DTO, a well-known design pattern. On the other hand, rec. $D21$ triggered in BrTCom involves moving a domain-specific class, which explains its higher complexity.

Next, we rely on this classification to answer our research questions.

RQ #1 – *For what portion of architectural violations detected in a real-world system can the proposed approach provide repairing recommendations?*

As reported in Table 5.4, our approach was able to trigger recommendations for 768 out of 828 detected violations (93%). Particularly, we were able to provide recommendations for all architectural defect types, with the exception of 60 constraints whose violations were classified in the *bypassing layers* category. However, we argue that our approach can be improved to handle some of the violations in this category. As an example, we noticed that an additional recommendation suggesting the replacement of an exception with another one could correctly repair the two violations without recommendations in the case of constraint `TC6`. As another example, BrTCom's architect mentioned that an additional recommendation suggesting to move the accessed class or method might address the 54 violations without recommendations in the case of constraint `TC14`. Currently, `ArchFix` only suggests moving the class or method where the violations was detected. For example, in Code 3.1 (page 27), we can suggest to move the method `init` (where the violation was detected) to another class, but currently we do not suggest moving the method `foo`. In short, our proposed approach handles the vast majority of the detected violations and further improvements are possible and some of them seem to be general to OO systems.

RQ #2 – *Does the proposed approach provide correct recommendations for repairing architectural violations?*

As can be observed in Table 5.4, our approach triggered correct recommendations for 663 out of 828 detected violations (80%). More specifically, we achieved a precision greater than 75% for all architectural defect types, with the exception of the constraints whose violation were classified in the *context exploration* category. We define *precision* as the number of correct recommendations by the total number of recommendations triggered by `ArchFix`.

According to the architects' feedback, `ArchFix` was very precise handling the architectural defect types that do not involve complex repair actions. As evidence, BrTCom's architect highlighted the 270 violations on constraint `TC13` (*misusage of interfaces*) fixed by recommendation *D*2—which suggests replacing the declaration of an unauthorized abstract type (mostly, `List`) with one of its concrete subtypes (mostly, `ArrayList`)—as one of the "most important recommendations". According to the architect, by not following this recommendation (associated to the correct use of the

GWT framework), the current implementation experiences an overhead in the size of the generated JavaScript code, with important negative consequences both in terms of CPU performance and network bandwidth consumption. On the other hand, `ArchFix` was unable to achieve good precision on violations related to *context exploration*. For example, constraint `TC10` defines that `Client` classes cannot create `Date` objects (to avoid time synchronization bugs). As a result of the conformance process, we found 84 violations of this constraint in BrTCom classes. However, the recommendations suggested by `ArchFix` just include the removal of the instantiations. This repair action was classified by BrTCom's architect as *partially correct*. In fact, he indicated that the correct repair action in this case would require a refactoring in the `Server` interfaces to return `Date` instances in particular cases. Therefore, instead of creating the `Date` on the client process, the client code should invoke the refactored interfaces.

As another relevant finding, we noticed that some incorrect recommendations were triggered because the `suitable_module` function did not indicate the current (and also correct) module as the most suitable one. For instance, in the case of one violation in constraint `TC11` due to *unintended dependencies*, our approach failed to indicate the most suitable module. However, the module calculated with the second best similarity was in fact the correct one. For this reason, we are considering a revision in our `suitable_module` implementation to indicate more than one module whenever the similarity value is very close to the highest calculated one, or even boost the priority of the current module.

Last but not least, as a practical contribution of our evaluation, the architects of both systems opened a maintenance request in the issue management platform of their systems requesting a correction for the detected violations and suggesting the use of the recommendations provided by `ArchFix`. Particularly, Geplanes' maintainers have repaired the detected violations in the system's main development trunk. On the other hand, BrTCom's maintainers still have not repaired the detected violations. In fact, it is not clear to the company the *return on investment* (ROI) of fixing architectural violations.

RQ #3 – *How complex is to discover a correct repair action without the support of our recommendation system?*

As reported in Table 5.5, most correct recommendations were scored as having *moderate* or *major* complexity. Particularly, 241 violations related to *bypassing layers* were the ones classified predominantly as having a *major* complexity. After asking the architects for clarification, we realized that repairing these violations requires a global

understanding of the system, including knowledge on the public interfaces of all layers. On the other hand, recommendations associated to the *misusage of persistence* and other *domain-specific patterns* were mostly classified as having a *minor* complexity. They are usually associated to missing or incorrect use of annotations, which can be fixed more easily, by just inserting or removing a given annotation.

Table 5.5: Classification of the detected violations (focusing on complexity)

	Complexity					
	correct recs.			**incorrect recs.**		
Architectural Defect Type	*minor*	*moderate*	*major*	*minor*	*moderate*	*major*
Misusage of persistence patterns	55	2	-	-	11	-
Misusage of domain-specific patterns	24	-	-	1	-	-
Bypassing layers	18	2	241	-	-	-
Unintended dependencies	31	-	5	-	-	1
Bypassing creational patterns	-	15	-	-	-	-
Context exploration	-	-	-	-	-	-
Misusage of interfaces	-	270	-	-	-	-
	128	289	246	1	11	1

RQ #4 – *How complex is to reject an incorrect repair action provided by our recommendation system?*

As also reported in Table 5.5, the results indicate that rejecting an incorrect recommendation has a higher complexity than accepting a correct one (12 out of 13 incorrect recs. require a *moderate* or *major* effort). However, we argue that the number of correct recommendations raised by our approach outperforms by a large margin the number of incorrect ones (663 vs 13 recs. in our case studies).

5.7 Lessons Learned

We identified five lessons learned from our experience on evaluating the use of `ArchFix` in the Geplanes and BrTCom systems. We learned that we could have avoided many incorrect recommendations if the `suitable_module` function indicated a set of modules with close similarity values, rather than a single module. In certain cases, we noticed that the suitable module was indeed the one with the second best similarity. For this reason, we are considering a revision in our suitable module heuristic to indicate more than one module whenever the similarity value is very close to the highest calculated one.

Second, the architects highlighted that `ArchFix` does not target senior developers but mainly developers who recently joined the project. They typically refer to the lack of awareness about architectural rules as the main cause of the violations. According to the architects, our approach might help less experienced developers to understand the system architecture by showing the correct repair procedure for the detected violations.

Third, the real value of conformance checking occurs when it is integrated to the regular development and maintenance processes. As an evidence, Knodel et at. demonstrate that teams supported by constructive compliance checking may insert about 60% less structural violations into the architecture [49]. In this thesis, we claim that an architectural repair recommendation system—such as `ArchFix`—can also be integrated into the regular development process. In such way, besides the detection of points of violations, we would also contemplate a mechanism to repair the detected violations.

Forth, only 17 out of the 32 proposed recommendations, which emerged from our initial study [102], were triggered in our case studies. However, we believe that the unused recommendations are generic enough to be triggered in other architecture erosion fixing contexts. For example, the unused recommendation $D1$ addresses violations due to the *misusage of interfaces*, which were commonly detected in other case studies [88, 50, 102].

Last but not least, most recommendations were scored as having *moderate* or *major* complexity. Therefore, we claim that our approach may save developers' time when fixing architectural violations, although we have not measured this aspect in our evaluation. Instead, we assumed that the more complex the recommendation is, the more time is required to discover and to apply the suggested repair action.

5.8 Threats to Validity

In this section, we identify and classify threats to validity in our evaluation [20, 113]:

Internal Validity: Since many steps of our evaluation require the involvement of architects, they could be affected negatively (e.g., tired or bored) during the experiment. In order to minimize this threat, we conducted one meeting for each task: constraints definition, violations validation, correctness assessment, and complexity assessment. More important, we carefully planned each meeting to last up to 50 minutes.

External Validity: First, although we used two industrial-strength web systems that have different architectures and constraints, we cannot claim that our approach will provide equivalent results in other systems (as usual in empirical studies in software

engineering). Second, since the target systems presented a moderate number of violations (slightly over 1 violation/KLOC), we cannot claim that our approach provides the same precision in systems already facing a major architectural erosion process. Fundamentally, a major erosion process may impact the precision of underlying functions, such as `suitable_module`. Third, since we detected violations using DCL, we cannot claim that our approach provides equivalent coverage and precision when using other architecture conformance tools. More important, the proposed approach targets specific classes of violations, and other types of violations may be present in a particular system. However, DCL was able to express all constraints proposed by the architects of two large and complex systems (BrTCom, as reported in this thesis, and SGP, our training system, as reported in Section 3.3.1, page 29). Fourth, since we used half of the proposed repairing recommendations, it was not possible to evaluate the remaining recommendations. However, the unused recommendations would have been useful at least once in our training system [102].

Construct Validity: In our evaluation, we relied on two chief architects (one per system) to define the constraints, to validate whether the detected violations are in fact true positives, to judge the correctness, and to assess the complexity of the recommendations. As typical in human-based classifications, our results might be affected by some degree of subjectivity. However, it is important to highlight that we interviewed the chief architects who designed the evaluated architectures and are responsible for their maintenance and evolution. Therefore, they are the right experts to evaluate the correctness of a given recommendation. Moreover, our assessment of complexity is based on a fairly precise definition, which relies on the scope of the classes that might be inspected. Furthermore, instead of assessing each violation separately, we grouped similar recommendations to make the evaluation easier to the architects. More important, the architects' answers for the recommendations in the same group were always consistent.

5.9 Final Remarks

In this chapter, we conducted an evaluation with two industrial-strength systems that provided us with encouraging feedback on the applicability and correctness of our recommendations. For the first system—a 21 KLOC open-source strategic management system—our approach indicated correct repairing recommendations for 31 out of 41 violations detected as the result of an architecture conformance process. For the second system—a 728 KLOC customer care system used by a major telecommunication

company—our approach triggered correct recommendations for 632 out of 787 violations, as asserted by the system's architect.

In conclusion, considering both systems, `ArchFix` indicated the correct repair action for 663 out of 828 violations detected as the result of an architecture conformance process (80%). Moreover, the architects scored 81% of these recommendations as having *moderate* or *major* complexity. These results support our claim that the approach proposed in this thesis can assist developers when repairing violations in the static architecture of object-oriented systems.

Chapter 6

Conclusion

Architectural erosion is a recurrent problem faced by software architects. However, a clear dichotomy is perceived in the tools already designed to tackle this problem. On the one hand, there are several approaches and commercial tools proposed to uncover architectural violations [68, 96, 23, 102, 2, 16]. On the other hand, the task of fixing the hundreds of violations raised after an architecture conformance process is normally conducted with limited tool support.

To address this shortcoming, we described a solution based on recommendation system principles that provides repairing guidelines for developers and maintainers when fixing violations in the module architecture view of object-oriented systems. The proposed system provides recommendations for violations—divergences and absences—raised by static architecture conformance checking approaches. We also elaborated a suitable module heuristic to determine the correct module for source code entities based on their structural similarity and designed a tool—called `ArchFix`—that implements our approach and hence provides recommendations for architectural violations in Java systems. Moreover, we conducted an evaluation with two industrial-strength systems that provided us with encouraging feedback on the applicability of our recommendations. Considering both systems, `ArchFix` indicated the *correct* repair actions for 632 (80%) out of 828 violations detected as the result of an architecture conformance process. The architects also scored 80% of these recommendations as having *moderate* or *major* complexity.

We organized this chapter as follows. First, Section 6.1 reviews the contributions of our research. Next, Section 6.2 points the limitations of our approach. Finally, Section 6.3 presents the further work.

6.1 Contributions

This research makes the following contributions:

- The design of a solution based on recommendation system principles that provides repairing guidelines for developers and maintainers when reversing software architecture erosion (Chapter 3);
- The specification of 32 repairing recommendations that emerged after an in-depth investigation of possible fixes for more than 2,200 violations (Section 3.3);
- A prototype tool called `ArchFix` that implements our approach and hence provides recommendations for violations—divergences and absences—raised by static architectural constraints (Section 3.4);
- The employment of speculative analysis on our prioritization policy (Section 3.3.4);
- A suitable module heuristic to infer the correct module for source code entities based on their structural similarity (Section 3.3.5);
- An empirical study that supports the implementation decisions (coefficients and strategies) related to our suitable module heuristic (Section 4.2);
- An evaluation of the correctness and complexity of our repairing recommendations on two real-world systems, including a qualitative discussion on the results (Chapter 5);
- The Qualitas.*class* Corpus, which is a compiled version of the original Qualitas Corpus (Section 4.1);
- A review of the state-of-the-art and state-of-the-practice w.r.t. architectural models, architecture conformance approaches, refactoring techniques, remodularization methods, and recommendation systems (Chapter 2).

6.2 Limitations

Our work has the following limitations:

- Our approach may *not* provide the same precision in systems already facing a major architectural erosion process. Such scenario may impact the precision of important functions, such as `suitable_module`;

- Our suitable module heuristic is based *only* on structural similarity, even though the way that developers decide how to fix a violation might consider semantic aspects;
- Our catalog of repairing recommendations is *not* complete, which is far ahead of our initial objectives;
- The preconditions of our repairing recommendations are *not* complete. They only provide minimal conditions to prevent the insertion of new violations and to reduce undesirable side effects;
- Our tool has been *initially* designed to work as a recommendation system and automatizes *only* some trivial repair actions, such as replacing a type and removing annotations;
- We have *not* evaluated whether our approach provides equivalent coverage and precision when using architecture conformance tools other than DCL;
- We have *not* evaluated whether our approach provides equivalent results in contexts other than web-based systems;
- Our approach has *not* been designed to deal with concurrency issues.

6.3 Future Work

In particular, we intend to complement this work with the following future work:

- *The Proposed Approach*: (i) the design of an architecture-repair recommendation language to allow maintainers to extend our catalog of repairing recommendations with their own domain-specific ones; (ii) a crossover study to compare the effort required by developers to repair architectural violations with and without `ArchFix`; (iii) new cases studies to refine and to extend the catalog of recommendations, to evaluate the prioritization policy based on speculative analysis, and to demonstrate the applicability of our approach in contexts other than web-based systems; (iv) an evolutionary case study in which architectural violations were detected and solved as the system evolves to quantify how many violations were repaired exactly as our approach would have suggested; (v) an analysis of the order in which the suggested repair actions are applied to maximize the number of repairing recommendations; and (vi) the design of a strategy to indicate confidence levels for triggered repairing recommendations.

- *The Suitable Module Heuristic*: (i) the replication of the conducted experiment to consider the most suitable class for a method; (ii) the assessment the impact of misplaced classes in the corpus; (iii) a statistical analysis of the distribution of the precision per system; (iv) the use of semantic properties to improve the precision of our heuristic; (v) a sensitivity analysis of the factors `a`, `b`, `c`, and `d` in the ranking to propose a specific coefficient for measuring similarity in object-oriented systems; and (vi) an investigation of the impact on the results when the similarity between a class `C` and a package `Pkg` are calculated by the average of the resulting similarity between `C` and each `Pkg` class—rather than by the resulting similarity between `C` and `Pkg`, as evaluated in this thesis.

- *The ArchFix Tool*: (i) the implementation of the repairing functions; (ii) the investigation of new heuristics to enhance the precision of auxiliary functions, such as `factory` and `delegate`; and (iii) the definition of a public interface to make our solution more extensible and open in order to facilitate its use with other architectural conformance tools.

Bibliography

[1] Ackermann, C., Lindvall, M., and Cleaveland, R. (2009). Towards behavioral reflexion models. In *20th International Symposium on Software Reliability Engineering (ISSRE)*, pages 175–184.

[2] Aldrich, J., Chambers, C., and Notkin, D. (2002). ArchJava: connecting software architecture to implementation. In *22nd International Conference on Software Engineering (ICSE)*, pages 187–197.

[3] Allen, R. and Garlan, D. (1997). A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249.

[4] Anquetil, N. and Laval, J. (2011). Legacy software restructuring: Analyzing a concrete case. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 279–286.

[5] Anquetil, N. and Lethbridge, T. (1999). Experiments with clustering as a software remodularization method. In *6th Working Conference on Reverse Engineering (WCRE)*, pages 235–255.

[6] Baeza-Yates, R. and Ribeiro-Neto, B. (2011). *Modern Information Retrieval*. Pearson, 2nd edition.

[7] Baldwin, C. Y. and Clark, K. B. (1999). *Design Rules: The Power of Modularity*. MIT Press.

[8] Bansiya, J. and Davis, C. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17.

[9] Bischofberger, W. R., Kühl, J., and Löffler, S. (2004). Sotograph - a pragmatic approach to source code architecture conformance checking. In *European Workshop on Software Architecture (EWSA)*, pages 1–9.

[10] Bittencourt, R. (2010). Conformance checking during software evolution. In *17th Working Conference on Reverse Engineering (WCRE)*, pages 289–292.

[11] Bittencourt, R., Jansen de Souza Santos, G., Guerrero, D., and Murphy, G. (2010). Improving automated mapping in reflexion models using information retrieval techniques. In *17th Working Conference on Reverse Engineering (WCRE)*, pages 163–172.

[12] Borba, P., Sampaio, A., Cavalcanti, A., and Cornélio, M. (2004). Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, 52(1-3):53–100.

[13] Borchers, J. (2011). Invited talk: Reengineering from a practitioner's view – a personal lesson's learned assessment. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 1–2.

[14] Bourquin, F. and Keller, R. K. (2007). High-impact refactoring based on architecture violations. In *11th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 149–158.

[15] Brunet, J., Guerreiro, D., and Figueiredo, J. (2009). Design tests: An approach to programmatically check your code against design rules. In *31st International Conference on Software Engineering (ICSE), New Ideas and Emerging Results Track*, pages 255 –258.

[16] Brunet, J., Guerreiro, D., and Figueiredo, J. (2011). Structural conformance checking with design tests: An evaluation of usability and scalability. In *27th International Conference on Software Maintenance (ICSM)*, pages 143–152.

[17] Burke, R. (2007). Hybrid web recommender systems. In *The Adaptive Web*, volume 4321 of *Lecture Notes in Computer Science*, pages 377–408. Springer Berlin Heidelberg.

[18] Chern, R. and Volder, K. D. (2008). The impact of static-dynamic coupling on remodularization. In *23rd Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 261–276.

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

[20] Cook, T. D. and Campbell, D. T. (1979). *Quasi-experimentation: design and analysis issues for field settings*. Houghton Mifflin.

[21] Cubranic, D. and Murphy, G. (2003). Hipikat: recommending pertinent software development artifacts. In *25th International Conference on Software Engineering*, pages 408–418.

[22] Dagenais, B. and Robillard, M. P. (2008). Recommending adaptive changes for framework evolution. In *30th International Conference on Software Engineering (ICSE)*, pages 481–490.

[23] de Moor, O. (2007). Keynote address: .QL for source code analysis. In *7th IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 3–14.

[24] De Schutter, K. (2012). Automated architectural reviews with semmle. In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 557–565.

[25] de Silva, L. and Balasubramaniam, D. (2012). Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1):132–151.

[26] Ducasse, S. and Pollet, D. (2009). Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591.

[27] Eichberg, M., Kloppenburg, S., Klose, K., and Mezini, M. (2008). Defining and continuous checking of structural program dependencies. In *30th International Conference on Software Engineering (ICSE)*, pages 391–400.

[28] Everitt, B. S., Landau, S., Leese, M., and Stahl, D. (2011). *Cluster Analysis*. Wiley, 5th edition.

[29] Feilkas, M., Ratiu, D., and Jurgens, E. (2009). The loss of architectural knowledge during system evolution: An industrial case study. In *17th IEEE International Conference on Program Comprehension (ICPC)*, pages 188–197.

[30] Fokaefs, M., Tsantalis, N., Stroulia, E., and Chatzigeorgiou, A. (2011). Jdeodorant: identification and application of extract class refactorings. In *33rd International Conference on Software Engineering (ICSE)*, pages 1037–1039.

[31] Fokaefs, M., Tsantalis, N., Stroulia, E., and Chatzigeorgiou, A. (2012). Identification and application of extract class refactorings in object-oriented systems. *Journal of Systems and Software*, 85(10):2241–2260.

[32] Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley, Boston.

[33] Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley, Boston.

[34] Frenzel, P., Koschke, R., Breu, A., and Angstmann, K. (2007). Extending the reflexion method for consolidating software variants into product lines. In *14th Working Conference on Reverse Engineering (WCRE)*, pages 160 –169.

[35] Garcia, J., Popescu, D., Edwards, G., and Medvidovic, N. (2009a). Identifying architectural bad smells. In *13th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 255–258.

[36] Garcia, J., Popescu, D., Edwards, G., and Medvidovic, N. (2009b). Toward a catalogue of architectural bad smells. In *5th International Conference on the Quality of Software Architectures (QoSA)*, pages 146–162.

[37] Garlan, D., Monroe, R., and Wile, D. (1997). Acme: an architecture description interchange language. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 1–15.

[38] Garlan, D. and Shaw, M. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall.

[39] Glorie, M., Zaidman, A., van Deursen, A., and Hofland, L. (2009). Splitting a large software repository for easing future software evolution - an industrial experience report. *Journal of Software Maintenance*, 21(2):113–141.

[40] Henderson-Sellers, B. (1996). *Object-oriented metrics: measures of complexity*. Prentice-Hall.

[41] High, T. and Sutton, I. (2010). Re-architecting a large code base: The "remodularization" of xenon. http://www.slideshare.net/Rinky25/rearchitecting-a-large-codebase.

[42] Hochstein, L. and Lindvall, M. (2005). Combating architectural degeneration: a survey. *Information and Software Technology*, 47(10):643–656.

[43] Holmes, R., Walker, R., and Murphy, G. (2006). Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32(12):952–970.

[44] Hou, D. and Hoover, H. J. (2006). Using SCL to specify and check design intent in source code. *IEEE Transactions on Software Engineering*, 32(6):404–423.

[45] Hou, D., Hoover, H. J., and Rudnicki, P. (2004). Specifying framework constraints with FCL. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 96–110.

[46] Kerievsky, J. (2004). *Refactoring to Patterns*. Pearson.

[47] Knodel, J., Muthig, D., Haury, U., and Meier, G. (2008a). Architecture compliance checking - experiences from successful technology transfer to industry. In *12th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 43–52.

[48] Knodel, J., Muthig, D., Naab, M., and Lindvall, M. (2006). Static evaluation of software architectures. In *10th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 279–294.

[49] Knodel, J., Muthig, D., and Rost, D. (2008b). Constructive architecture compliance checking - an experiment on support by live feedback. In *24th International Conference on Software Maintenance (ICSM)*, pages 287–296.

[50] Knodel, J. and Popescu, D. (2007). A comparison of static architecture compliance checking approaches. In *6th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, page 12.

[51] Koschke, R., Frenzel, P., Breu, A., and Angstmann, K. (2009). Extending the reflexion method for consolidating software variants into product lines. *Software Quality Journal*, 17:331–366.

[52] Koschke, R. and Simon, D. (2003). Hierarchical reflexion models. In *10th Working Conference on Reverse Engineering (WCRE)*, pages 36–47.

[53] Kruchten, P. (1995). The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50.

[54] Lanza, M., Marinescu, R., and Ducasse, S. (2005). *Object-Oriented Metrics in Practice*. Springer-Verlag.

[55] Lindvall, M. and Muthig, D. (2008). Bridging the software architecture gap. *Computer*, 41(6):98–101.

[56] Luckham, D., Kenney, J., Augustin, L., Vera, J., Bryan, D., and Mann, W. (1995). Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–354.

[57] Luckham, D. and Vera, J. (1995). An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734.

[58] Maffort, C., Valente, M. T., Anquetil, N., Hora, A., and Bigonha, M. (2013a). Heuristics for discovering architectural violations. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 1–10.

[59] Maffort, C., Valente, M. T., Bigonha, M., Hora, A., Anquetil, N., and Menezes, J. (2013b). Mining architectural patterns using association rules. In *25th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 375–380.

[60] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. (1995). Specifying distributed software architectures. In *5th European Software Engineering Conference (ESEC)*, pages 137–153.

[61] Martin, R. (1994). OO design quality metrics – an analysis of dependencies. In *Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA'94*, pages 1--8.

[62] McMillan, C., Poshyvanyk, D., and Grechanik, M. (2010). Recommending source code examples via API call usages and documentation. In *2nd International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pages 21–25.

[63] Medvidovic, N. and Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93.

[64] Mitchell, B. S. and Mancoridis, S. (2006). On the automatic modularization of software systems using the Bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208.

[65] Moghadam, I. H. and Cinnéide, M. Ó. (2012). Automated refactoring using design differencing. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 43–52.

[66] Montandon, J. E., Borges, H., Felix, D., and Valente, M. T. (2013). Documenting apis with examples: Lessons learned with the apiminer platform. In *20th Working Conference on Reverse Engineering (WCRE), Practice Track*, pages 1–8.

[67] Muşlu, K., Brun, Y., Holmes, R., Ernst, M. D., and Notkin, D. (2012). Speculative analysis of integrated development environment recommendations. In *27th*

Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 1–15.

[68] Murphy, G., Notkin, D., and Sullivan, K. (1995). Software reflexion models: Bridging the gap between source and high-level models. In *3rd Symposium on Foundations of Software Engineering (FSE)*, pages 18–28.

[69] Murphy, G., Notkin, D., and Sullivan, K. (2001). Software reflexion models. *IEEE Transactions on Software Engineering*, 27(4):364–380.

[70] Murphy-Hill, E. and Black, A. P. (2008). Seven habits of a highly effective smell detector. In *1st International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pages 11–15.

[71] Murphy-Hill, E., Parnin, C., and Black, A. P. (2009). How we refactor, and how we know it. In *31st International Conference on Software Engineering (ICSE)*, pages 287–297.

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

[73] Naseem, R., Maqbool, O., and Muhammad, S. (2011). Improved similarity measures for software clustering. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 45–54.

[74] O'Keeffe, M. K. and Cinnéide, M. Ó. (2006). Search-based software maintenance. In *10th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 249–260.

[75] Opdyke, W. (1992). *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign.

[76] Parnas, D. L. (1994). Software aging. In *16th International Conference on Software Engineering (ICSE)*, pages 279–287.

[77] Passos, L., Terra, R., Diniz, R., Valente, M. T., and Mendonça, N. (2010). Static architecture-conformance checking: An illustrative overview. *IEEE Software*, 27(5):82–89.

[78] Perry, D. E. and Wolf, A. L. (1992). Foundations for the study of software architecture. *Software Engineering Notes*, 17(4):40–52.

[79] Pruijt, L., Koppe, C., and Brinkkemper, S. (2013). On the accuracy of architecture compliance checking support. In *21st IEEE International Conference on Program Comprehension (ICPC)*, pages 172–181.

[80] Rama, G. M. and Patel, N. (2010). Software modularization operators. In *26th International Conference on Software Maintenance (ICSM)*, pages 1–10.

[81] Resnick, P. and Varian, H. R. (1997). Recommender systems. *Communications of the ACM*, 40(3):56–58.

[82] Robillard, M., Walker, R., and Zimmermann, T. (2010). Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86.

[83] Romesburg, H. C. (2005). *Cluster Analysis for Researchers*. Lulu Press, North Carolina.

[84] Roock, S. and Lippert, M. (2006). *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley.

[85] Sales, V., Terra, R., Miranda, L. F., and Valente, M. T. (2013a). JMove: Seus métodos em classes apropriadas. In *IV Brazilian Conference on Software: Theory and Practice (CBSoft), Tools Session*, pages 1–6.

[86] Sales, V., Terra, R., Miranda, L. F., and Valente, M. T. (2013b). Recommending move method refactorings using dependency sets. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 232–241.

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

[88] Sarkar, S., Ramachandran, S., Kumar, G. S., Iyengar, M. K., Rangarajan, K., and Sivagnanam, S. (2009). Modularization of a large-scale business application: A case study. *IEEE Software*, 26:28–35.

[89] Schäefer, M. and de Moor, O. (2010). Specifying and implementing refactorings. In *25th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 286–301.

[90] Schäfer, M., Ekman, T., and de Moor, O. (2009). Challenge proposal: verification of refactorings. In *3rd Workshop on Programming Languages meets Program Verification (PLPV)*, pages 67–72.

[91] Silva, L. H., Terra, R., and Valente, M. T. (2011). A case study on improving maintainability and evolvability using architectural constraints. In *X Simpósio Brasileiro de Qualidade de Software (SBQS)*, pages 1–15.

[92] Simon, F., Steinbruckner, F., and Lewerentz, C. (2001). Metrics based refactoring. In *5th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 30–38.

[93] Soares, G., Gheyi, R., and Massoni, T. (2013). Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 39(2):147–162.

[94] Soares, G., Gheyi, R., Serey, D., and Massoni, T. (2010). Making program refactoring safer. *IEEE Software*, 27(4):52–57.

[95] Steimann, F. and Thies, A. (2009). From public to private to absent: Refactoring Java programs under constrained accessibility. In *23rd European Conference on Object-Oriented Programming (ECOOP)*, pages 419–443.

[96] Sullivan, K. J., Griswold, W. G., Cai, Y., and Hallen, B. (2001). The structure and value of modularity in software design. In *9th International Symposium on Foundations of Software Engineering (FSE)*, pages 99–108.

[97] Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. (2010). The Qualitas Corpus: A curated collection of Java code for empirical studies. In *17th Asia Pacific Software Engineering Conference (APSEC)*, pages 336–345.

[98] Terra, R., Brunet, J., Miranda, L. F., Valente, M. T., Serey, D., Castilho, D., and Bigonha, R. S. (2013a). Measuring the structural similarity between source code entities. In *25th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 753–758.

[99] Terra, R., Miranda, L. F., Valente, M. T., and Bigonha, R. S. (2013b). Qualitas.class Corpus: A compiled version of the Qualitas Corpus. *Software Engineering Notes*, 38(5):1–4.

[100] Terra, R. and Valente, M. T. (2008a). Towards a dependency constraint language to manage software architectures. In *2nd European Conference on Software Architecture (ECSA)*, pages 256–263.

[101] Terra, R. and Valente, M. T. (2008b). Verificação estática de arquiteturas de software utilizando restrições de dependência. In *II Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS)*, pages 1–14.

[102] Terra, R. and Valente, M. T. (2009). A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 32(12):1073–1094.

[103] Terra, R. and Valente, M. T. (2010). Definição de padrões arquiteturais e seu impacto em atividades de manutenção de software. In *VII Workshop de Manutenção de Software Moderna (WMSWM)*, pages 1–8.

[104] Terra, R., Valente, M. T., Bigonha, R. S., and Czarnecki, K. (2012a). DCLfix: A recommendation system for repairing architectural violations. In *III Brazilian Conference on Software: Theory and Practice (CBSoft), Tools Session*, pages 1–6.

[105] Terra, R., Valente, M. T., Czarnecki, K., and Bigonha, R. S. (2012b). Recommending refactorings to reverse software architecture erosion. In *16th European Conference on Software Maintenance and Reengineering (CSMR), Early Research Achievements Track*, pages 335–340.

[106] Terra, R., Valente, M. T., Czarnecki, K., and Bigonha, R. S. (2013c). A recommendation system for repairing violations detected by static architecture conformance checking. *Software: Practice and Experience*, pages 1–28.

[107] Thummalapenta, S. and Xie, T. (2007). PARSEWeb: a programmer assistant for reusing open source code on the web. In *22nd International Conference on Automated Software Engineering (ASE)*, pages 204–213.

[108] Tsantalis, N. and Chatzigeorgiou, A. (2009). Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 99:347–367.

[109] Tsantalis, N. and Chatzigeorgiou, A. (2011). Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782.

[110] Unphon, H. and Dittrich, Y. (2010). Software architecture awareness in long-term software product evolution. *Journal of Systems and Software*, 83(11):2211–2226.

[111] van Gurp, J. and Bosch, J. (2002). Design erosion: problems and causes. *Journal of Systems and Software*, 61:105–119.

[112] Verbaere, M., Ettinger, R., and de Moor, O. (2006). JunGL: a scripting language for refactoring. In *28th International Conference on Software Engineering (ICSE)*, pages 172–181.

[113] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers.

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

[115] Xing, Z. and Stroulia, E. (2008). The JDEvAn tool suite in support of object-oriented evolutionary development. In *30th International Conference on Software Engineering (ICSE), Research Demonstration Track*, pages 951–952.

[116] Ye, Y. and Fischer, G. (2005). Reuse-conducive development environments. *Automated Software Engineering*, 12:199–235.

[117] Zimmermann, T., Zeller, A., Weissgerber, P., and Diehl, S. (2005). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445.

Appendix A

Repairing Functions

In this appendix, we provide detailed descriptions and illustrative examples on the non-trivial *repairing* functions used by `ArchFix` to recommend architectural repair actions, as introduced in Table 3.2 (Chapter 3).

Function `inline(exp, v, S)`: Inlines `exp` in the uses of variable `v` in the block of code `S`. To illustrate, assume a constraint in the form `Bar cannot−declare Foo`, but it is allowed to *access* `Foo` as presented in Figure A.1*(before)*. Therefore, a violation [`Bar, cannot, declare, Foo`] occurs at line 3 in this figure.

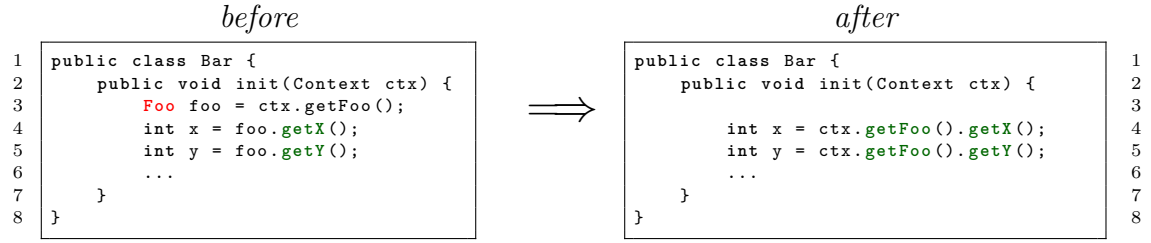


Figure A.1: `inline( [ctx.getFoo()], foo, S )`

Since class `Bar` is allowed to *access* `Foo`, a potential repair action to fix such violation is as follows: `inline( [ctx.getFoo()], foo, S )`, where `S` is the body of method `init(Context)`. As can be observed in Figure A.1*(after)*, this repair action removes the declaration of variable `foo` (line 3) and inlines the expression `ctx.getFoo()` to its previous uses (lines 4–5).

Function `promote_param(f, v, exp)`: Promotes variable `v` to a formal parameter of method `f`; `exp` is the corresponding argument in the calls to `f`. To illustrate, assume a constraint in the form `Facade cannot−access Foo`, but `Facade` is allowed to *declare* `Foo` as presented in Figure A.2*(before)*. Therefore, a violation [`Facade`, `cannot`, `access`, `Foo`] occurs at line 9 in this figure.

before

```
1  public class MainClass {
2      public void main() {
3          ...
4          facade.setup(foo);
5      }
6  }
7  public class Facade {
8      public void setup(Foo foo) {
9          Date d = foo.getDate();
10         ...
11     }
12 }
```

⟹

after

```
1  public class MainClass {
2      public void main() {
3          ...
4          facade.setup(foo, foo.getDate());
5      }
6  }
7  public class Facade {
8      public void setup(Foo foo, Date d) {
9          ...
10         ...
11     }
12 }
```

Figure A.2: `promote_param( setup(Foo), d, [foo.getDate()] )`

Assuming that `MainClass` is allowed to access `Foo`, a potential repair action to fix such violation may be defined as follows: `promote_param( setup(Foo), d, [foo.getDate()] )`. As can be observed in Figure A.2*(after)*, after this repair action, (i) variable `d` is promoted to a formal parameter of `setup(Foo)` (line 8); and (ii) calls to `setup(Foo, Date)` are adjusted to include the actual parameter `foo.getDate()` (line 4).

Function `unwrap_return(f, T, exp)`: Considering a method `f` that returns `new T(exp)`, this function modifies this statement to just return `exp` and moves the instantiations of the wrapper type `T` to the respective call sites. To illustrate, assume a constraint in the form `Model cannot−create Foo` as presented in Figure A.3*(before)*. Therefore, a violation [`Model`, `cannot`, `create`, `Foo`] occurs at line 10 in this figure.

before

```
1  public class MainClass {
2      public void main() {
3          Foo foo = m.retrieve();
4          ...
5      }
6  }
7  public class Model {
8      public Foo retrieve() {
9          ...
10         return new Foo(obj);
11     }
12 }
```

⟹

after

```
1  public class MainClass {
2      public void main() {
3          Foo foo = new Foo(m.retrieve());
4          ...
5      }
6  }
7  public class Model {
8      public ObjType retrieve() {
9          ...
10         return obj;
11     }
12 }
```

Figure A.3: `unwrap_return( retrieve(), Foo, [obj] )`

Assuming that class `MainClass` is allowed to *create* `Foo`, a potential repair action to fix this violation is as follows: `unwrap_return(retrieve(), Foo, [obj])`. As can be observed in Figure A.3*(after)*, (i) the return statement is modified to return just `obj` (line 10); (ii) the return type of `retrieve` is modified to `ObjType` (i.e., the type of `obj`) (line 8); and (iii) an object of `Foo` is created to wrap the result returned by the call to `retrieve` (line 3).

Appendix B

Auxiliary Functions

In this appendix, we provide detailed descriptions and illustrative examples on the non-trivial *auxiliary* functions used by `ArchFix` to recommend architectural repair actions, as introduced in Table 3.3 (Chapter 3).

Function `delegate(f)`: Searches a delegate method for `f`, i.e., a method that just encapsulates a call to `f` [33]. Basically, our heuristic to find delegate methods consists in finding a method that (i) only invokes method `f` and (ii) returns the same type of `f`. For instance, as presented in Figure B.1, `Persistence::persist(Bar)` is a delegate method for `Bar::save(Connection)` because it only forwards the call (line 5).

```
1 public class Persistence {
2     Connection conn = ...;
3     ...
4     public int persist(Bar bar) {
5         return bar.save(conn);
6     }
7 }
```

Figure B.1: `delegate( Bar::save(Connection) ) = Persistence::persist(Bar)`

Function `factory(C, exp)`: Searches for a factory method for class `C`, accepting `exp` as input. Basically, our heuristic to find a factory method for a class `C` consists in finding a method that just returns an object of type `C` created using `exp`. For instance, as presented in Figure B.2, `DAOFactory::getBar(int)` is a factory method for `Bar` (lines 3–5).

```
1 public class DAOFactory {
2     ...
3     public int getBar(int max) {
4         return new Bar(max);
5     }
6 }
```

Figure B.2: `factory(Bar, {[5]})` = `DAOFactory::getBar(int)`

Function `gen_decl(f)`: Returns a declaration `D` for a variable `c` of type `C`, where `C` is the class that defines the method `f`. The simplest scenario happens when `f` is a static method and this function returns the static reference to the class `C`. However, when `f` is not static, this function proceeds as follows: (i) if `C` is a singleton, it uses the `getInstance` method, e.g., `C c = C.getInstance()`; (ii) when C has a factory, it obtains an instance from the factory, e.g., `C c = Factory.getC()`; (iii) otherwise the function creates a null-initialized stub object to make the call, e.g., `C c = new C(...)`.

Function `gen_factory(C, exp)`: Generates a factory for class `C`, accepting `exp` as input. Basically, it creates a class using the template illustrated on the left of Figure B.3, where <T> is the target type and <exp_t> is the list of expression types. For instance, the subfigure on the right illustrates the generated factory class for `gen_factory(Bar, {['a'], [5]})`.

Factory Template

```
1  public class <T>Factory {
2      private <T> <t>;
3
4      private <T>Factory() { }
5
6      public <T> get<t>( <exp_t> ) {
7          if (<t> == null) {
8              <t> = new <T>( <exp_t> );
9          }
10         return <t>;
11     }
12 }
```

⟹

Factory of class `Bar`

```
1  public class BarFactory {
2      private Bar bar;
3
4      private BarFactory() { }
5
6      public Bar getBar(char c, int i) {
7          if (bar == null) {
8              bar = new Bar(c,i);
9          }
10         return bar;
11     }
12 }
```

Figure B.3: `gen_factory(Bar, {['a'], [5]})`

Appendix C

Description of Repairing Recommendations

This appendix describes the architectural repair recommendations in details. We explain each proposed recommendation—the preconditions and the indicated repair action—and then illustrate application scenarios.

[A, cannot, declare, B]			
B b; S	$\Longrightarrow$ replace([B], [B′]),	if B′ ∈ super(B) ∧ typecheck([B′ b ; S]) ∧ B′ ∉ M_B	$D1$
B b; S	$\Longrightarrow$ replace([B], [B′]),	if B′ ∈ sub(B) ∧ typecheck([B′ b ; S]) ∧ B′ ∉ M_B	$D2$
B b = exp; S	$\Longrightarrow$ propagate([exp], b, [S]),	if can(A, access, B)	$D3$
g (B b) { S }	$\Longrightarrow$ remove([B b]),	if typecheck([g(){ S }])	$D4$
try { S } catch (B b) { S′}	$\Longrightarrow$ replace([B], [B′]),	if B′ ∈ super(B) ∧ typecheck([try { S } catch (B b) { S′}]) ∧ B′ ∉ M_B	$D5$

***D*1**: Replace the unauthorized type B with one of its supertypes B′, since such B′ is outside M_B and it can be type checked. This recommendation is particularly useful to handle violations due to the use of a concrete implementation of a service, instead of its general interface.

***D*2**: Replace the unauthorized type B with one of its subtypes B′. As an example, developers when implementing web-based systems using GWT (Google Web Toolkit) should avoid the use of generic types (e.g., `java.util.Collection`) on GWT interfaces to reduce the size of the generated JavaScript code. Therefore, whenever possible, they should rely on more specialized types (e.g., `java.util.ArrayList` instead of `java.util.Collection`).

***D*3**: Remove the unauthorized declaration followed by the propagation of the initialization expression `exp` to all uses of the declared identifier. This recommendation can be triggered when `A` cannot declare `B` but is allowed to access it.

***D*4**: Remove a parameter declaration whenever it is not being used, i.e., the removal does not cause any compiler error.

***D*5**: Replace the unauthorized exception of type `B` with one of its supertypes `B′` whenever such `B′` is outside $\mathtt{M_B}$.

[A, cannot, access, B]		
b.f	$\Longrightarrow$ replace([b.f], [D; c.g]), if g = delegate(f) $\wedge$ ⟨D,c⟩ = gen_decl(g) $\wedge$ type(c) $\notin \mathtt{M_B}$	*D*6
b.f	$\Longrightarrow$ replace([b.f], [D; c.g]), if equals_sig(f, g) $\wedge$ ⟨D,c⟩ = gen_decl(g) $\wedge$ type(c) $\notin \mathtt{M_B}$	*D*7
b.f	$\Longrightarrow$ g = extract([b.f]), move(g, M), if M = suitable_module(g) $\wedge$ can(A, access, M)	*D*8
b.f	$\Longrightarrow$ remove([b.f]), if $\overline{\mathtt{M_A}} = \emptyset$	*D*9
g (p){ T v = exp_b }	$\Longrightarrow$ promote_param(g, v, [exp_b]), if $\forall$C $\in$ call_sites(g), can(C, access, B)	*D*10

***D*6**: Replace an unauthorized call to a method `f` with a call to a delegate method `g`—i.e., a method that just encapsulates a call to `f`. This recommendation can only be triggered if the type that contains such delegate method is outside $\mathtt{M_B}$.

***D*7**: Similarly to *D*6, replace an unauthorized call to a method `f` with a method `g` that has an equal signature.

***D*8**: Extract a new method `g` that encapsulates the call to `f` and then move `g` to a class inside the most suitable module `M`. Moreover, the current module ($\mathtt{M_A}$) must not be the most suitable one (`M`).

***D*9**: Remove a call to a given method `f` when no class in the system can access the class where `f` is implemented. This recommendation is particularly useful when developers access methods whose usage is restricted. For instance, developers tend to establish dependencies with the Java API `System` class (e.g., by calling `System.out.println`) as a practice of rudimentary debugging. Nevertheless, these calls must be removed, especially in web-based systems.

***D*10**: Promote variable `v`—whose initialization expression `exp_b` contains the unauthorized access—to a formal parameter of the enclosing method `g`. In this case, the initialization expression `exp_b` must be used as the argument in `g` calls. This recommendation is particularly useful when accessing `B` is allowed from all `g` call sites.

[A, cannot, create, B]		
new B(exp)	$\Longrightarrow$ replace([new B(exp)], [FB.getB(exp)]), if FB = factory(B, [exp]) $\wedge$ can(A, access, FB)	$D11$
new B(exp)	$\Longrightarrow$ replace([new B(exp)], [null]), if $\overline{\mathtt{M_A}} = \emptyset$	$D12$
new B(exp)	$\Longrightarrow$ replace([new B(exp)], [FB.getB(exp)]), if FB = gen_factory(B, [exp]) $\wedge$ can(A, access, FB)	$D13$
g (p){return new B(exp)}	$\Longrightarrow$ unwrap_return(g, B, [exp]), if $\forall$C $\in$ call_sites(g), can(C, create, B)	$D14$

***D*11**: Replace a `new` operator with a call to the `get` method of a Factory `FB`. It addresses the situation where developers—due to unawareness or forgetfulness—create directly objects of classes that have a well-defined factory.

***D*12**: Remove the instantiation whether no class in the system is allowed to create `B`. For example, this recommendation is useful when objects of some classes are supposed to be provided by dependency injection techniques.

***D*13**: Replace the `new` operator with a call to the `get` method of a new Factory class, which will be created by the auxiliary function `gen_factory`.

***D*14**: Replace an instantiation of `B` using `exp` in a return statement with only `exp`, delegating the responsibility for the creation of `B` to the call sites. Moreover, this recommendation involves modifying the signature of method `g` (as performed by the function `replace_return`) and it can only be triggered if all call sites are able to create `B`.

[A, cannot, throw, B]		
g (p) throws B { S }	$\Longrightarrow$ remove([throws B]), if typecheck([g (p) { S }])	$D15$
g (p) throws B { S }	$\Longrightarrow$ remove([throws B]), replace([S], [try {S} catch(B b) {S′}]), if can(A, declare, B) $\wedge$ S′ = user_code()	$D16$
g (p) throws B { S }	$\Longrightarrow$ replace([B], [B′]), if B′ $\in$ super(B) $\wedge$ B′ $\notin$ $\mathtt{M_B}$	$D17$
g (p) throws B { S }	$\Longrightarrow$ move(g, M), if M = suitable_module(g) $\wedge$ M $\neq$ $\mathtt{M_A}$	$D18$

***D*15**: Remove the `throws` clause, in the cases it is not need. This recommendation addresses the tendency of developers to over-declare the exceptions that can be raised in a method.

***D*16**: Remove the `throws` clause and insert a `try−catch` block around the body of the method to handle a given exception internally. In this particular case, the developers must provide the code that handles the exception, as required by the auxiliary function `user_code`.

***D*17**: Replace the unauthorized exception type B with one of its supertypes B′, assuming that B′ is outside $\mathtt{M_B}$.

***D*18**: Move method g to a class in the most suitable module M, assuming that M is different than the current module. For instance, suppose that a method g throwing `DAOException`—which possibility should be implemented by a `DAO` class—was mistakenly implemented by a class outside the `DAO` module. In such case, this recommendation suggests to move g to a `DAO` class.

[A, cannot, derive, B]		
class A derive B	$\Longrightarrow$ replace([B], [B′]), if B′ ∈ super(B) ∧ typecheck([A derive B′]) ∧ ¬override(B, B′) ∧ B′ ∉ $\mathtt{M_B}$	*D*19
class A derive B	$\Longrightarrow$ move(A, M), if M = suitable_module(A) ∧ can(A, derive, B)	*D*20

***D*19**: Make class A derive from B′, which is one of the supertypes of B. This recommendation can only be triggered if such replacement type checks, if B does not override methods of B′ (in order to avoid any possible changes in semantics), and if B′ is outside $\mathtt{M_B}$.

***D*20**: Move a class A to a most suitable module M. It is particularly useful when developers mistakenly implement a class in the wrong module, e.g., a `ProductReport` class in the `View` layer.

[A, cannot, useannotation, B]		
@B class A	$\Longrightarrow$ move(A, M), if M = suitable_module(A) ∧ M ≠ $\mathtt{M_A}$	*D*21
@B class A	$\Longrightarrow$ remove([@B]) if $\mathtt{M_A}$ = suitable_module(A)	*D*22
@B g (p){ S }	$\Longrightarrow$ move(g, M), if M = suitable_module(g) ∧ M ≠ $\mathtt{M_A}$	*D*23
@B g (p){ S }	$\Longrightarrow$ remove([@B]) if $\mathtt{M_A}$ ≠ suitable_module(A)	*D*24

***D*21**: Move class A to the most suitable module M, assuming that M is different than $\mathtt{M_A}$, i.e., the current module of A is not the most suitable one. This recommendation is particularly useful when developers implement a class in the wrong module, e.g., a class using annotations related to persistent concerns in the `Controller` layer.

***D*22**: Remove the class-type annotation B from class A whenever $\mathtt{M_A}$ is already the suitable module for A and it is not able to receive B.

***D*23**: Similarly to *D*21, move method `g` to a class located in the most suitable module `M` whenever the current module of `g` is not the most suitable one.

***D*24**: Similarly to *D*22, remove the method-type annotation `B` from method `g` whenever $\texttt{M}_\texttt{A}$ is already the suitable module for `g` and it is not able to receive `B`.

[A, must, throw, B]		
g (p){ S }	$\Longrightarrow$ replace([g (p){ S }], [g (p) throws B { S }]), remove_catch(B, S) if has_catch(B, S)	*A*1
g (p){ S }	$\Longrightarrow$ move(g, M) if M = suitable_module(g) $\wedge$ M $\neq \texttt{M}_\texttt{A}$	*A*2

***A*1**: Add exception `B` in the `throws` clause of method `g` when such exception is being handled by a certain `catch` in the method's body. This recommendation is particularly useful to guide developers in throwing exceptions that should not be handled internally by a method.

***A*2**: Move method `g` to a class in the most suitable module `M` whenever the current module of `g` is not the most suitable one.

[A, must, derive, B]		
class A	$\Longrightarrow$ replace([A], [A derive B]), if $\texttt{M}_\texttt{A}$ = suitable_module(A) $\wedge$ typecheck([class A derive B])	*A*3
class A	$\Longrightarrow$ move(A, M), if M = suitable_module(A) $\wedge$ M $\neq \texttt{M}_\texttt{A}$	*A*4

***A*3**: Make class `A` extend or implement `B`. It addresses the situation where developers have failed to derive from the base types of the module. For instance, an `Entity` class must implement `Serializable` to provide persistence. However, assume that a given entity class `Product` does not implement `Serializable`. Because `Entity` classes rely extensively on the same types, the `suitable_module` function will likely infer that `Product` is indeed in its correct module and therefore must implement `Serializable`.

***A*4**: Move class `A` to a most suitable module `M`, because the current module of `A` is not the most suitable one. For instance, suppose a class that must derive `View` but establishes dependencies only with `Controller` types. In such case, this recommendation will suggest the movement of this class to a module in the `Controller` layer.

[A, must, useannotation, B]		
class A	$\Longrightarrow$ move(A, M), if M = suitable_module(A) $\wedge$ M $\neq$ M_A $\wedge$ target(B) = type	$A5$
class A	$\Longrightarrow$ replace([class A], [@B class A]), if M_A = suitable_module(A) $\wedge$ target(B) = type	$A6$
g (p){ S }	$\Longrightarrow$ move(g, M), if M = suitable_module(A) $\wedge$ M $\neq$ M_A $\wedge$ target(B) = method	$A7$
g (p){ S }	$\Longrightarrow$ replace([g(p)], [@B g(p)]), if M_A = suitable_module(g) $\wedge$ target(B) = method	$A8$

A5: Move class A to the most suitable module M, assuming that M is different than M_A, i.e., the current module of A is not the most suitable one.

A6: Add class-type annotation B to class A whenever M_A is already the suitable module for A and hence A has to receive annotation B.

A7: Move method g to a class in the most suitable module M whenever the current module of g is not the most suitable one.

A8: Add method-type annotation B to method g, assuming that M_A is already the suitable module for g and hence g has to receive annotation B.

Appendix D

Metrics Data from the Qualitas.*class* Corpus

This appendix presents a subset of the metrics gathered for the 111 systems in the Qualitas.*class* Corpus. More specifically, the corpus includes the values of the following 23 source code metrics measured at the level of classes[1] [19, 54, 40, 61]:

- *Basic Metrics*: Number of lines of code (LOC)[2], Number of packages (NOP), Number of classes (NOCL), Number of interfaces (NOI), Number of methods (NOM), Number of attributes (NOA), Number of overridden methods (NORM), Number of parameters (PAR), Number of static methods (NSM), and Number of static attributes (NSA).

- *Complexity Metrics*: Method lines of code (MLOC), Specialization index (SIX), McCabe cyclomatic complexity (VG), Nested block depth (NBD), and Normalized distance (RMD).

- *CK Metrics*: Weighted methods per class (WMC), Depth of inheritance tree (DIT), Number of children (NOC), and Lack of cohesion in methods (LCOM).[3]

- *Coupling Metrics*: Afferent coupling (CA), Efferent coupling (CE), Instability (I), and Abstractness (A).

[1] Except for the metrics LOC, NOP, NOCL, and NOI that were measured at the system level.
[2] Our metric counts non-blank and non-comment lines of codes.
[3] Our metric relies on the LCOM HS (Henderson-Sellers) method [40].

We relied on Google CodePro Analytix[4] and Metrics[5] to compute the metrics. For each project P, the Qualitas.*class* Corpus provides a `XML` file with a `Metric` element for each metric M (identified by the attribute `id`). For example, the element <`Metric id` = ″`NOM`″ `avg` = ″`4.04`″ `stddev` = ″`7.189`″ ... > contains the average value and the standard deviation of the metric *Number of Methods*. Particularly, such element was extracted from the XML file of JASML 0.1.

Table D.1 presents a subset of the metrics gathered for the systems in the corpus. As can be noticed, the corpus is very heterogeneous. For example, systems' size ranges from 3.5 KLOC (fitjava) to 2,500 KLOC (eclipse). There are lowly- (e.g., jasml, LCOM 0.08) and highly-cohesive systems (e.g., freecs, LCOM 0.57). Analogously, there are lowly- (e.g., xmojo, CE 0.6) and highly-coupled systems (e.g., megamek, CE 38).

[4]CodePro Analytix 7.1.0, http://developers.google.com/java-dev-tools.
[5]Metrics 1.3.8, http://metrics2.sourceforge.net.

Table D.1: The Qualitas.*class* Corpus (1 of 2)

System	Version*	KLOC	NOP	NOCL	NOI	MLOC	NOM	NOA	NOC	DIT	NORM	SIX	VG	WMC	LCOM	CA	CE	I	A
ant	1.8.2	127.6	127	1627	97	6.30±12.4	7.77±10.3	2.46±4.6	0.83±6.2	2.45±1.4	0.68±1.3	0.33±0.6	2.12±3.0	17.19±28.0	0.25±0.4	14.43±69.7	7.45±14.6	0.81±0.3	0.09±0.1
antlr	3.4	47.4	20	381	20	8.80±13.3	10.14±16.2	2.73±5.5	1.45±10.8	1.90±1.2	0.82±2.3	0.35±0.8	2.05±2.9	21.66±38.4	0.20±0.3	15.60±22.4	8.60±10.8	0.58±0.4	0.07±0.1
aoi	2.8.1	110	23	604	32	12.38±29.0	10.33±11.5	6.27±7.6	0.60±3.0	2.16±1.8	1.67±3.3	0.32±0.6	3.39±6.7	37.34±62.8	0.38±0.3	62.56±93.3	18.56±19.0	0.55±0.4	0.09±0.1
argouml	0.34	105.8	77	1408	107	7.50±15.5	6.01±8.7	1.29±3.1	0.79±4.4	3.03±2.1	0.91±2.2	0.57±1.1	2.32±3.8	14.92±25.6	0.13±0.3	31.78±63.4	13.83±23.8	0.48±0.3	0.14±0.2
aspectj	1.6.9	501.8	144	3600	564	8.40±23.3	11.01±20.8	3.18±12.1	0.85±2.5	1.85±1.5	1.53±5.4	0.42±0.8	3.03±8.6	36.58±100.4	0.25±0.3	37.19±63.9	13.91±20.8	0.41±0.3	0.25±0.3
axion	1.0-M2	24.2	13	257	38	5.03±11.6	11.81±22.1	2.08±6.5	1.02±2.5	1.60±1.0	0.96±2.1	0.21±0.4	2.22±4.4	26.45±89.7	0.20±0.3	24.31±41.6	14.54±11.2	0.53±0.2	0.21±0.2
azureus	4.7.0.2	495.5	473	4038	1077	8.68±26.0	8.15±14.4	2.60±5.7	1.27±10.8	1.20±1.1	0.21±0.7	0.11±0.4	2.36±4.4	20.97±45.3	0.22±0.3	20.56±64.0	4.61±6.5	0.43±0.3	0.28±0.3
batik	1.7	194.7	117	2624	288	7.90±22.1	5.96±8.0	2.77±19.0	0.86±3.8	2.19±2.0	0.58±1.6	0.33±0.8	2.37±5.5	14.90±34.5	0.17±0.3	16.76±38.8	9.65±19.2	0.61±0.4	0.20±0.3
c_jdbc	2.0.2	95.5	145	777	16	10.60±24.1	7.53±15.5	2.50±5.0	0.65±2.9	2.88±1.9	0.43±1.2	0.25±0.7	2.47±4.1	19.45±44.7	0.22±0.3	14.88±28.9	4.26±4.6	0.51±0.4	0.11±0.2
castor	1.3.3*	219.7	381	2803	156	7.01±20.0	7.35±11.4	2.08±4.0	0.54±3.8	2.04±1.3	0.95±2.9	0.37±0.8	2.06±4.5	15.77±46.6	0.27±0.4	7.03±20.4	5.34±6.3	0.76±0.3	0.08±0.2
cayenne	3.0.1	198.2	235	3193	160	6.43±12.4	5.73±8.3	1.32±2.6	0.88±5.7	2.60±1.6	0.48±1.7	0.30±0.7	1.82±2.9	10.81±21.5	0.15±0.3	24.67±81.7	10.02±15.4	0.61±0.3	0.12±0.2
checkstyle	5.6*	36.6	42	553	18	7.40±9.5	5.00±5.0	1.20±2.2	0.76±7.4	2.52±1.5	0.64±1.1	0.44±0.8	1.92±2.4	10.01±13.0	0.14±0.3	13.90±46.6	9.57±9.3	0.88±0.3	0.08±0.2
cobertura	1.9.4.1	54.6	34	160	11	12.18±25.1	20.91±90.1	6.82±19.8	0.32±1.4	1.66±1.1	0.29±0.7	0.14±0.4	5.24±11.3	114.72±533.7	0.25±0.4	2.18±4.6	2.06±2.1	0.64±0.4	0.11±0.2
collections	3.2.1	55.4	23	676	27	6.12±10.2	8.42±9.1	1.07±1.7	0.91±3.1	3.16±2.4	1.74±2.8	0.85±1.3	1.70±1.7	16.31±26.6	0.12±0.2	22.09±48.0	17.04±12.0	0.61±0.3	0.13±0.1
colt	1.2.0	35.9	24	381	67	7.11±14.7	8.19±10.9	1.53±4.0	2.02±7.8	1.95±1.7	1.20±2.9	0.33±0.7	2.54±3.8	25.76±42.4	0.12±0.2	22.92±26.9	9.21±8.9	0.45±0.3	0.26±0.3
columba	1.0	71.9	212	1183	110	6.52±11.4	5.26±6.2	1.80±3.3	0.51±4.3	2.54±1.9	0.31±1.5	0.17±0.5	1.80±2.0	9.97±13.9	0.19±0.3	10.91±25.2	4.25±5.4	0.53±0.3	0.17±0.3
compiere	330	400.5	78	2622	57	6.50±19.5	13.24±19.1	2.83±6.7	0.78±14.0	2.53±1.4	0.57±1.4	0.23±0.6	2.26±4.1	32.00±50.4	0.27±0.5	128.51±382.9	25.35±94.6	0.51±0.4	0.09±0.2
derby	10.9.1.0*	651.1	217	3010	388	12.00±54.0	11.47±26.7	3.57±25.2	0.78±6.6	2.28±1.8	0.93±2.9	0.31±0.7	2.62±5.9	34.64±134.3	0.22±0.3	45.79±135.1	10.37±23.5	0.53±0.4	0.22±0.4
displaytag	1.2	20.5	32	320	16	6.70±13.7	5.19±8.6	1.44±4.4	0.69±6.7	2.88±1.6	0.59±2.2	0.25±0.8	1.86±2.5	9.93±19.3	0.12±0.3	13.91±27.5	8.28±20.1	0.65±0.4	0.12±0.2
drawswf	1.2.9	27.7	34	311	25	6.21±13.7	8.23±14.8	2.67±3.8	0.54±1.8	2.40±1.8	0.49±1.7	0.23±0.7	2.02±3.5	17.73±30.3	0.26±0.3	8.18±10.4	4.15±5.0	0.38±0.3	0.14±0.3
drjava	20100913-r5387	89.5	30	1210	95	7.43±22.8	6.94±15.8	1.93±8.3	1.01±6.1	2.34±1.7	0.40±1.7	0.18±0.6	1.86±2.5	13.53±37.0	0.14±0.3	34.23±50.4	14.23±16.5	0.44±0.3	0.20±0.1
eclipse_SDK	3.7.1	2484.3	1425	24871	3410	8.06±17.4	7.75±12.9	2.42±6.9	1.17±12.8	2.02±1.7	0.75±3.3	0.33±0.7	2.73±5.3	23.30±62.9	0.22±0.3	61.61±287.5	11.42±15.2	0.46±0.3	0.25±0.3
emma	2.0.5312	23.1	33	321	57	7.47±20.0	4.93±5.3	2.90±12.3	0.57±1.5	1.42±1.2	0.38±0.9	0.14±0.4	2.53±4.3	14.28±22.8	0.16±0.3	9.24±12.5	3.97±4.5	0.40±0.3	0.34±0.3
exoportal	1.0.2	96	413	2162	257	4.34±8.4	5.53±6.9	1.44±2.5	0.64±6.0	2.20±1.8	0.44±1.1	0.37±0.9	1.58±1.7	9.06±12.2	0.18±0.3	8.50±24.6	3.04±3.8	0.59±0.4	0.15±0.3
findbugs	1.3.9	110.8	67	1432	127	7.15±21.7	6.14±11.3	2.49±5.4	0.64±3.2	1.98±1.8	0.61±2.8	0.34±0.8	2.85±7.8	19.33±40.8	0.23±0.3	26.66±66.6	12.25±26.6	0.59±0.4	0.18±0.3
fitjava	1.1	3.5	5	95	0	4.38±5.4	4.37±4.3	1.82±2.7	0.70±2.8	2.36±0.9	0.60±0.7	0.66±1.0	1.88±2.1	9.14±10.8	0.19±0.3	7.00±14.0	8.00±5.7	0.82±0.3	0.00±0.0
fitlibraryforfitnesse	20110301*	46.6	157	1337	102	3.34±6.0	4.64±7.1	1.19±1.9	0.71±4.2	1.96±1.5	0.23±0.7	0.13±0.4	1.53±1.6	7.65±15.5	0.12±0.3	13.15±29.7	5.01±8.3	0.58±0.4	0.13±0.2
freecol	0.10.3	106.4	51	952	32	9.63±20.8	7.71±15.2	2.54±4.3	0.83±5.4	2.88±1.9	0.60±1.2	0.32±0.8	2.76±5.1	22.95±59.3	0.20±0.3	29.39±53.7	10.69±19.7	0.45±0.3	0.07±0.1
freecs	1.3.20100406	22.6	12	146	16	12.30±34.2	8.58±14.8	4.56±12.3	0.59±4.4	1.42±0.7	0.90±0.9	0.23±0.2	4.22±10.2	40.46±70.1	0.57±0.4	30.75±34.2	10.08±14.2	0.42±0.3	0.14±0.3
freemind	0.9.0	52.8	45	751	79	5.80±13.7	7.21±12.8	2.10±5.6	0.65±2.1	2.15±1.6	0.41±1.9	0.17±0.5	1.90±2.6	14.19±28.1	0.19±0.3	22.62±46.9	7.87±10.1	0.51±0.3	0.13±0.2
galleon	2.3.0	61.1	35	533	30	11.23±26.8	6.60±10.6	3.90±5.6	0.58±3.9	3.30±2.5	0.56±0.9	0.44±0.7	2.85±6.3	21.38±32.7	0.33±0.4	12.49±28.4	6.57±7.0	0.77±0.3	0.04±0.1
ganttproject	2.1.1*	48.6	55	785	115	5.64±13.7	6.42±9.6	2.67±6.0	0.72±2.3	1.83±1.6	0.39±0.9	0.19±0.5	1.72±2.9	11.28±22.4	0.26±0.4	20.91±38.1	7.02±7.8	0.50±0.3	0.15±0.2
gt2	9.2*	876.5	894	9991	1560	6.55±24.6	7.82±15.9	2.19±14.6	0.50±2.9	2.19±1.7	0.69±2.6	0.31±0.7	1.96±3.7	16.60±39.4	0.15±0.3	29.93±95.2	8.78±12.5	0.63±0.4	0.13±0.3
hadoop	1.1.2*	319.9	238	3968	189	8.81±17.0	5.42±9.3	2.26±4.9	0.64±3.9	1.79±1.0	0.44±1.4	0.24±0.6	2.19±3.0	13.52±29.4	0.18±0.3	27.93±93.6	7.84±16.4	0.56±0.4	0.10±0.2
heritrix	1.14.4	64.9	48	656	51	7.75±13.3	7.72±12.0	1.85±3.9	0.76±2.9	3.08±2.4	0.58±1.2	0.50±1.1	2.22±3.0	18.94±33.6	0.18±0.3	17.98±36.3	8.31±8.3	0.53±0.3	0.16±0.2
hibernate	4.2.0*	431.7	856	7119	661	5.18±12.2	6.47±11.8	1.88±3.6	0.70±6.1	1.69±1.3	0.60±2.0	0.25±0.6	1.52±1.9	10.14±23.4	0.26±0.3	13.74±72.5	4.46±7.0	0.75±0.3	0.11±0.2
hsqldb	2.0.0	149.7	34	660	58	11.36±27.7	13.33±20.1	5.93±22.0	0.48±1.8	1.74±1.2	1.27±3.7	0.23±0.6	3.30±7.0	48.57±94.9	0.32±0.4	29.26±49.7	10.65±18.4	0.53±0.3	0.19±0.2
htmlunit	2.8	100.8	43	903	32	8.61±14.3	9.00±21.6	1.16±7.9	0.91±8.3	3.00±1.9	0.42±1.3	0.28±0.8	1.67±2.5	15.59±37.5	0.10±0.3	26.05±70.6	16.44±29.0	0.62±0.3	0.04±0.1
informa	0.7.0-alpha2	218	120	1771	45	8.75±26.5	8.87±12.4	3.28±7.6	0.47±4.0	4.43±2.3	2.18±6.8	1.08±1.2	2.06±3.3	18.87±37.5	0.28±0.4	25.89±92.2	12.70±25.0	0.64±0.3	0.04±0.1
ireport	3.7.5	13.9	26	223	46	5.05±14.5	6.70±10.1	1.80±3.7	0.67±2.1	1.55±1.4	0.23±0.6	0.08±0.4	1.74±2.9	12.62±17.1	0.20±0.3	9.31±23.6	5.77±5.6	0.73±0.3	0.11±0.2
itext	5.0.3	78.3	34	583	42	8.89±21.4	9.08±16.0	3.73±7.0	0.50±2.4	1.76±1.3	0.46±1.6	0.14±0.4	2.98±5.8	30.55±62.2	0.26±0.4	14.09±33.3	8.59±21.0	0.44±0.3	0.10±0.2
ivatagroupware	0.11.3	24.7	94	228	24	8.19±19.0	8.63±9.2	2.78±4.0	0.20±0.5	2.05±1.3	0.60±1.0	0.32±0.6	2.20±3.8	19.53±32.6	0.33±0.4	5.31±11.9	2.05±2.1	0.55±0.4	0.13±0.3
jFin_DateMath	1.0.1	8.9	26	121	2	5.57±12.8	6.42±7.8	5.28±29.7	0.18±1.0	2.08±0.9	0.14±0.4	0.05±0.2	1.49±1.7	10.30±11.5	0.16±0.3	1.81±3.6	3.65±2.7	0.81±0.4	0.06±0.2
jag	6.1	15.7	18	136	7	8.11±20.9	9.10±10.9	4.21±7.0	0.33±1.2	1.95±1.6	0.29±0.5	0.14±0.6	2.36±4.1	23.28±32.2	0.38±0.4	6.89±7.3	4.28±5.4	0.36±0.3	0.05±0.1
james	2.2.0	42.8	53	531	78	8.02±19.6	6.38±10.0	2.25±4.5	0.67±2.2	1.95±1.4	0.57±3.1	0.29±0.7	2.32±4.0	15.11±30.4	0.18±0.3	8.25±18.8	6.02±9.1	0.65±0.4	0.17±0.2
jasml	0.1	5.7	5	50	1	15.26±39.8	4.04±7.2	3.42±4.6	0.48±2.0	1.50±0.7	0.24±0.4	0.22±0.4	5.30±13.8	26.82±67.7	0.08±0.2	3.00±3.0	2.00±1.4	0.57±0.4	0.02±0.0
jasperreports	3.7.4*	169.8	61	1709	285	6.62±16.0	8.98±18.9	2.60±6.3	0.97±4.9	1.55±1.1	0.16±0.5	0.08±0.4	1.85±2.8	17.77±40.6	0.22±0.4	47.13±138.1	20.39±24.6	0.58±0.3	0.20±0.3
javacc	5.0	18.3	12	135	6	12.47±34.9	5.03±6.9	2.30±3.7	0.78±3.9	1.90±1.0	0.29±1.7	0.10±0.3	3.65±8.0	30.95±71.9	0.20±0.4	2.33±4.9	2.33±3.0	0.76±0.4	0.10±0.1
jboss	5.1.0	524.9	1170	7362	1546	6.52±14.3	6.07±8.7	1.44±3.3	0.41±2.0	2.00±2.1	0.34±1.0	0.20±0.7	1.77±2.4	11.37±20.4	0.15±0.3	2.45±8.0	3.53±5.6	0.71±0.4	0.23±0.3
jchempaint	3.0.1	212.8	201	2134	122	7.76±20.5	8.06±14.8	1.32±3.4	0.89±7.1	2.26±1.7	0.48±1.5	0.31±0.9	2.10±6.7	18.06±41.0	0.12±0.3	31.73±116.3	8.66±12.5	0.58±0.3	0.09±0.2
jedit	4.3.2	109.5	38	1017	69	10.98±20.8	6.01±16.4	3.10±8.6	0.42±2.1	2.14±1.9	0.59±1.2	0.38±0.8	3.11±5.6	21.83±82.5	0.20±0.3	19.08±43.9	8.53±11.5	0.67±0.3	0.12±0.1
jena	2.6.3	65.8	48	1279	192	3.94±11.8	6.98±14.3	1.42±4.6	1.33±6.1	1.86±1.6	0.65±1.5	0.40±0.7	1.81±4.8	13.78±32.4	0.12±0.3	35.90±67.2	14.40±13.6	0.50±0.3	0.23±0.2
jext	5.0	60.2	59	761	35	10.77±36.9	4.57±8.4	2.23±4.9	0.60±6.1	2.28±1.6	0.76±1.7	0.57±1.0	3.02±8.6	15.55±36.0	0.18±0.3	10.83±34.6	6.70±6.6	0.65±0.4	0.06±0.1
jfreechart	1.0.13	143.1	69	1037	103	9.54±16.3	10.60±17.3	2.10±5.2	0.64±2.2	2.32±1.3	0.68±1.3	0.15±0.4	2.02±3.0	22.97±43.9	0.17±0.3	28.96±56.2	12.25±11.2	0.63±0.4	0.10±0.1
jgraph	5.13.0.0	31.8	34	298	23	7.77±15.4	8.70±17.2	3.07±6.3	0.70±1.7	2.52±2.0	0.80±1.7	0.57±1.1	2.56±3.8	24.38±54.1	0.24±0.4	7.85±17.5	4.09±4.1	0.60±0.3	0.09±0.2
jgraphpad	5.10.0.2	24.2	22	375	16	8.94±21.3	4.40±9.6	1.57±5.5	0.72±6.1	3.62±1.8	0.46±1.3	0.46±1.1	2.46±3.5	11.83±27.2	0.12±0.3	15.54±31.2	12.00±25.1	0.66±0.4	0.14±0.2

* a more recent version than the one in the original Qualitas Corpus [97]

The Qualitas.*class* Corpus (2 of 2)

System	Version*	KLOC	NOP	NOCL	NOI	MLOC	NOM	NOA	NOC	DIT	NORM	SIX	VG	WMC	LCOM	CA	CE	I	A
jgrapht	0.8.1	17.2	30	324	33	6.79±10.5	4.29±4.8	1.47±2.4	0.72±2.0	1.96±1.4	0.26±1.0	0.15±0.6	1.79±1.7	8.08±10.1	0.12±0.2	19.47±40.3	6.53±10.1	0.50±0.4	0.15±0.3
jgroups	2.10.0	96.3	28	1036	80	7.77±12.3	7.07±9.9	3.51±5.3	0.55±3.7	1.50±1.0	0.83±2.0	0.27±0.5	2.45±3.2	20.40±39.1	0.29±0.3	46.07±87.9	15.25±17.5	0.52±0.3	0.13±0.2
jhotdraw	7.5.1	79.7	66	765	60	7.41±17.5	8.58±9.9	2.18±3.7	0.74±2.9	2.55±1.7	1.27±2.2	0.53±0.9	2.20±3.5	20.57±32.5	0.22±0.3	22.15±39.0	6.65±8.2	0.37±0.3	0.12±0.2
jmeter	2.5.1	94.8	175	1038	80	6.67±12.4	8.04±9.8	2.21±4.5	0.76±3.5	2.70±2.1	0.54±1.7	0.26±0.7	1.86±2.5	16.18±22.6	0.18±0.3	15.38±44.4	4.33±5.9	0.60±0.4	0.12±0.3
jmoney	0.4.4	8.2	4	83	3	9.77±29.6	6.93±8.3	5.18±8.0	0.45±1.7	3.20±2.0	0.49±1.5	0.46±1.1	1.88±2.3	13.39±19.4	0.32±0.4	13.00±11.4	8.50±7.4	0.50±0.3	0.08±0.1
joggplayer	1.1.4s	29.9	17	300	17	9.18±26.5	6.15±7.4	3.96±8.8	0.28±1.4	2.37±2.0	0.54±3.0	0.23±0.6	2.46±4.2	18.22±37.4	0.30±0.4	8.00±11.0	7.94±15.5	0.40±0.4	0.11±0.2
jparse	0.96	24.8	4	75	6	23.44±66.6	10.92±13.5	8.08±28.7	1.33±5.3	4.53±2.2	0.77±1.1	0.50±1.0	7.28±19.1	87.84±311.8	0.26±0.3	21.50±20.5	16.75±7.5	0.62±0.3	0.16±0.1
jpf	1.5.1	13.3	10	140	31	7.26±16.5	8.23±7.2	3.15±3.6	0.73±1.5	1.59±1.4	0.35±0.8	0.20±0.6	2.33±3.5	20.95±27.4	0.37±0.4	12.10±18.3	7.80±5.5	0.61±0.4	0.28±0.3
jrat	1.0-beta1*	14	77	382	37	3.63±5.5	4.29±4.9	1.58±2.2	0.30±1.1	1.56±1.3	0.49±1.8	0.18±0.5	1.42±1.1	6.85±7.8	0.19±0.3	5.82±17.0	3.26±2.6	0.66±0.3	0.12±0.2
jre	1.6.0	923.9	425	9923	1700	7.33±20.3	8.22±19.6	2.06±5.3	4.01±173.7	1.81±1.5	0.91±3.6	0.32±0.8	2.37±4.6	21.46±52.4	0.16±0.3	44.76±155.3	8.31±16.6	0.37±0.3	0.34±0.3
jrefactory	2.9.19	123.4	113	1668	53	6.82±14.2	6.09±18.8	1.67±6.1	0.57±3.2	2.92±2.1	0.51±1.2	0.38±0.8	2.50±5.1	17.40±165.5	0.13±0.3	19.04±44.1	8.88±14.0	0.56±0.3	0.07±0.1
jruby	1.7.3*	244.3	139	3689	176	5.81±20.5	5.69±20.0	1.23±5.6	2.22±46.2	1.64±1.2	0.66±2.2	0.25±0.6	2.11±4.5	14.63±59.3	0.10±0.2	36.94±119.4	8.94±18.9	0.54±0.4	0.12±0.2
jsXe	04_beta	18.5	14	251	11	8.93±19.1	5.04±8.4	2.04±3.5	0.21±1.2	2.05±1.8	0.39±0.9	0.22±0.6	2.71±5.8	15.29±31.5	0.19±0.3	15.07±17.2	6.50±4.3	0.42±0.2	0.09±0.1
jspwiki	2.8.4	60.2	70	582	36	9.69±16.4	6.51±11.9	1.82±3.5	0.47±2.3	2.03±1.1	0.53±1.2	0.25±0.6	2.24±3.2	16.02±27.7	0.19±0.3	14.24±40.2	6.16±7.5	0.54±0.3	0.11±0.2
jtopen	7.8*	342	15	1915	100	8.89±27.5	11.63±20.6	3.24±7.0	0.59±2.8	2.09±1.3	1.24±2.7	0.39±0.7	2.76±7.0	33.67±70.6	0.25±0.4	30.00±86.9	33.67±45.7	0.78±0.3	0.09±0.1
jung	2.0.1	37.9	44	588	63	6.20±14.4	5.87±8.7	2.34±3.8	0.67±2.1	1.79±1.6	0.40±1.2	0.19±0.6	1.86±2.3	11.44±15.8	0.22±0.3	20.36±34.6	7.14±6.9	0.51±0.3	0.17±0.2
junit	4.1	6.6	28	171	15	4.02±4.9	4.68±5.1	0.93±1.2	0.70±1.8	1.57±1.2	0.29±0.9	0.16±0.5	1.52±1.1	8.63±11.5	0.11±0.2	8.32±12.5	4.21±3.2	0.56±0.3	0.22±0.2
log4j	2.0-beta*	33.3	64	616	55	5.98±10.2	4.40±8.6	1.54±2.8	0.43±1.6	1.45±1.0	0.31±0.9	0.13±0.3	1.98±2.5	10.48±17.2	0.15±0.3	16.11±38.2	5.75±5.3	0.63±0.4	0.12±0.2
lucene	4.2.0*	413	387	4629	142	9.99±27.7	5.14±6.3	2.18±3.6	0.98±9.7	2.28±1.2	0.43±1.1	0.21±0.6	2.50±9.3	13.79±33.0	0.18±0.3	23.16±100.3	6.66±11.8	0.69±0.3	0.07±0.2
marauroa	3.8.1	17.7	41	247	14	6.22±9.1	6.57±7.7	1.98±3.4	0.41±2.2	1.55±0.9	0.37±0.7	0.10±0.2	1.81±2.0	12.72±19.9	0.18±0.3	10.98±19.7	4.39±6.0	0.60±0.4	0.10±0.2
maven	3.0.5*	70.9	181	916	175	6.25±13.8	6.72±12.9	1.80±3.4	0.41±1.0	1.68±1.5	0.38±2.0	0.10±0.4	1.81±2.3	12.70±30.1	0.18±0.3	6.61±16.9	3.76±5.4	0.66±0.4	0.20±0.3
megamek	0.35.18	242.8	37	1859	64	13.93±43.4	6.53±23.8	3.52±12.5	0.81±5.8	4.10±2.3	0.61±2.5	0.40±0.9	3.80±12.5	27.30±145.9	0.16±0.3	67.57±211.8	38.00±122.5	0.47±0.4	0.14±0.2
mvnforum	1.2.2-ga	105.3	75	784	144	8.20±23.0	9.88±15.6	2.85±7.0	0.51±2.7	1.15±0.9	0.20±1.0	0.07±0.3	2.39±6.2	27.44±48.2	0.22±0.4	20.12±41.8	7.11±8.8	0.53±0.4	0.15±0.3
myfaces_core	2.1.10*	343.3	277	3426	265	7.71±19.4	6.92±10.6	2.12±8.0	0.90±3.4	1.97±1.4	0.74±2.8	0.24±0.6	2.17±4.0	17.02±32.4	0.17±0.3	16.14±47.9	4.84±7.3	0.53±0.4	0.20±0.3
nakedobjects	4.0.0	133.9	496	2975	470	3.54±6.5	5.77±8.6	1.16±2.4	1.01±4.0	1.82±1.3	0.41±1.2	0.19±0.5	1.40±1.3	8.48±13.7	0.13±0.3	17.27±54.8	4.72±6.3	0.57±0.4	0.22±0.3
nekohtml	1.9.14	7.6	7	64	5	9.56±18.6	7.56±9.9	3.42±7.4	0.39±1.3	1.67±1.1	0.73±2.1	0.22±0.5	3.30±5.7	27.36±57.3	0.22±0.3	4.86±5.0	4.86±2.6	0.63±0.3	0.09±0.2
netbeans	7.3*	2153.2	1959	25054	2374	7.75±30.8	6.44±10.1	2.43±5.8	0.66±5.2	1.97±1.6	0.56±1.6	0.27±0.7	2.45±5.8	18.03±47.6	0.22±0.3	6.39±12.9	6.11±7.3	0.64±0.3	0.17±0.3
openjms	0.7.7-beta-1	39.4	66	616	79	5.85±9.6	6.12±7.5	1.62±2.8	0.66±1.5	1.86±1.5	0.34±0.8	0.14±0.4	1.86±2.0	12.44±18.3	0.18±0.3	11.21±17.8	5.44±6.2	0.52±0.4	0.17±0.2
oscache	2.3*	7.6	22	115	10	6.82±12.2	6.09±8.3	2.21±3.2	0.36±0.9	2.03±1.2	0.50±1.2	0.23±0.5	2.36±3.8	15.42±32.9	0.22±0.3	4.18±7.9	3.23±1.8	0.63±0.3	0.13±0.2
picocontainer	2.10.2	9.3	15	206	29	3.79±6.1	5.85±7.4	0.98±1.6	0.97±2.2	1.75±1.4	0.45±1.0	0.25±0.6	1.58±1.7	9.71±16.4	0.09±0.2	12.40±26.5	8.87±9.7	0.55±0.3	0.23±0.3
pmd	4.2.5	60.7	88	872	52	6.37±17.0	6.30±21.9	1.70±8.0	0.71±4.6	2.27±1.4	0.53±1.2	0.44±0.8	2.61±6.8	17.59±128.1	0.11±0.3	10.41±38.8	5.36±6.7	0.75±0.4	0.09±0.2
poi	3.6	203.1	212	2414	131	6.63±16.0	7.34±18.3	2.05±7.8	0.55±3.8	2.12±1.2	0.41±1.1	0.15±0.4	1.81±3.3	14.68±30.5	0.18±0.3	16.63±48.8	7.71±13.7	0.54±0.3	0.09±0.2
pooka	3.0-080505	44.5	28	491	36	7.67±13.2	8.31±12.2	2.31±4.0	0.48±1.6	2.48±1.9	0.85±3.1	0.27±0.8	2.53±3.2	21.50±38.8	0.26±0.4	16.93±27.4	7.50±12.6	0.46±0.3	0.13±0.2
proguard	4.9*	62.6	35	648	42	5.62±16.3	8.35±13.7	2.31±5.1	1.48±10.2	1.71±1.2	2.48±6.8	0.54±0.8	1.97±4.5	17.33±29.6	0.17±0.3	48.17±80.3	12.89±12.8	0.46±0.3	0.15±0.2
quartz	1.8.3	28.6	51	269	36	7.08±18.0	10.28±19.0	2.57±8.2	0.94±4.0	1.47±1.1	0.40±1.2	0.13±0.4	2.00±3.8	21.40±47.1	0.20±0.3	5.41±19.1	3.82±4.2	0.77±0.3	0.10±0.2
quickserver	1.4.7	18.3	28	196	23	8.33±24.0	7.50±15.2	3.65±8.0	0.40±1.4	1.50±1.3	0.47±2.3	0.15±0.4	2.44±5.7	20.03±44.6	0.26±0.4	8.64±17.3	4.71±3.5	0.61±0.4	0.13±0.3
quilt	0.6-a-5	8	20	113	9	6.63±13.3	6.99±7.9	3.40±4.6	0.49±1.7	1.55±1.0	0.46±0.9	0.18±0.4	1.91±2.5	13.81±14.7	0.31±0.4	6.05±8.1	4.05±4.7	0.57±0.4	0.14±0.2
roller	5.0.1*	65.7	95	738	76	6.08±13.2	8.06±9.9	2.34±3.6	0.40±2.1	1.78±1.3	0.67±1.2	0.30±0.6	1.96±2.8	17.46±22.1	0.28±0.4	8.21±16.8	6.22±7.0	0.65±0.3	0.13±0.2
rssowl	2.0.5	100.6	54	771	110	11.34±25.1	7.55±9.7	2.91±7.1	1.09±5.3	1.77±1.4	0.37±1.0	0.13±0.4	2.60±4.0	21.29±36.8	0.24±0.4	39.18±68.1	11.50±9.5	0.48±0.3	0.16±0.3
sablecc	3.2	28.4	5	242	5	9.45±41.9	8.56±23.2	1.81±3.6	0.96±5.9	2.15±1.0	2.09±5.4	0.58±0.7	2.08±6.8	18.34±53.1	0.15±0.3	33.40±47.1	26.00±33.3	0.56±0.3	0.10±0.1
sandmark	3.4	93.2	127	1045	42	9.47±24.6	6.15±8.5	2.16±3.2	0.62±2.4	2.10±1.4	0.52±1.3	0.20±0.4	2.77±5.9	19.42±40.3	0.19±0.3	10.83±39.0	5.07±8.4	0.72±0.3	0.08±0.2
springframework	3.0.5	329.4	598	5999	692	5.73±10.0	5.76±9.4	1.35±3.1	0.74±4.0	1.81±1.7	0.28±0.8	0.17±0.6	1.58±1.9	9.46±17.2	0.15±0.3	6.57±16.0	5.39±6.1	0.71±0.3	0.20±0.3
squirrel_sql	3.1.2	6.9	3	73	16	7.61±14.7	7.71±9.3	3.96±5.6	0.26±0.6	1.62±1.8	0.41±1.0	0.15±0.5	1.74±1.6	13.51±22.0	0.42±0.4	6.67±6.6	12.67±14.4	0.59±0.3	0.09±0.1
struts	2.2.1	143.4	261	2239	169	6.07±12.8	6.29±8.8	2.06±4.4	0.74±4.5	2.24±1.7	0.50±1.9	0.23±0.6	1.89±3.4	12.28±28.8	0.22±0.3	13.35±50.3	5.37±7.8	0.68±0.4	0.11±0.2
sunflow	0.07.2	22	22	209	19	10.28±25.7	6.13±7.5	3.95±6.2	0.64±2.5	1.17±0.9	0.12±0.4	0.07±0.3	2.80±5.5	19.68±29.6	0.20±0.3	18.54±29.4	6.18±6.3	0.52±0.3	0.03±0.1
tapestry	5.1.0.5	97.8	144	2133	406	4.88±7.8	4.31±9.5	1.41±2.7	0.74±4.2	1.39±1.5	0.15±1.6	0.05±0.3	1.38±1.2	6.30±13.8	0.14±0.3	29.66±84.3	11.12±25.4	0.60±0.4	0.17±0.3
tomcat	7.0.2	181.2	157	1876	187	7.38±17.4	8.15±15.0	2.69±5.7	0.61±2.9	1.68±1.2	0.71±2.3	0.30±0.7	2.53±4.5	22.24±49.0	0.23±0.3	14.62±31.8	4.69±7.4	0.54±0.4	0.16±0.3
trove	2.1.0	5.8	4	72	9	8.35±12.2	6.75±10.3	0.86±1.3	1.78±6.0	1.61±1.2	0.51±1.5	0.14±0.4	1.81±1.6	13.15±20.2	0.09±0.2	1.25±1.3	4.25±3.6	0.83±0.2	0.11±0.1
velocity	1.6.4	37	42	445	47	7.68±23.8	6.16±10.4	1.76±5.6	0.65±3.5	2.00±1.4	0.63±2.3	0.26±0.5	2.57±9.8	17.37±95.5	0.13±0.3	14.07±22.9	7.29±13.2	0.66±0.4	0.14±0.2
wct	1.5.2	52.3	130	677	82	4.62±11.9	8.99±11.2	3.24±5.2	0.39±1.7	1.93±1.9	0.23±0.5	0.14±0.5	1.66±2.4	15.36±23.7	0.36±0.4	10.29±22.1	3.61±4.9	0.51±0.4	0.13±0.2
webmail	0.7.10	10.1	19	115	11	6.42±15.4	8.66±10.3	1.98±2.8	0.77±2.7	1.57±1.1	0.34±0.7	0.15±0.4	2.11±3.4	19.25±34.4	0.23±0.3	12.05±15.3	3.58±4.8	0.43±0.4	0.12±0.3
weka	3.6.9*	272.6	113	1766	122	9.92±23.2	10.41±12.5	3.88±6.4	1.29±10.3	2.48±1.5	1.10±1.8	0.33±0.6	2.68±4.5	30.68±44.3	0.32±0.4	20.50±85.9	9.97±11.3	0.66±0.3	0.13±0.2
xalan	2.7.1	183.9	89	1409	238	9.88±36.9	8.21±15.3	1.85±4.1	1.01±3.5	2.05±1.6	1.11±2.6	0.65±1.2	2.41±5.7	21.47±42.9	0.17±0.3	27.86±56.2	8.70±16.3	0.56±0.4	0.22±0.3
xerces	2.10.0	126	53	947	157	8.98±23.1	9.76±13.6	2.98±7.0	0.76±2.8	2.00±1.8	1.21±3.4	0.43±1.0	3.06±6.3	31.79±67.4	0.22±0.3	21.11±38.2	10.15±12.8	0.56±0.3	0.19±0.3
xmojo	5.0.0	4	15	32	4	9.31±16.3	8.31±6.8	5.16±5.3	0.16±0.4	1.50±1.5	0.22±0.5	0.07±0.2	2.25±3.4	20.56±24.5	0.48±0.4	0.87±0.8	0.60±0.5	0.47±0.4	0.10±0.2

* a more recent version than the one in the original Qualitas Corpus [97]