

A Unified Approach to Architecture Conformance Checking

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Andrea Caracciolo
von Zürich

Leiter der Arbeit:
Prof. Dr. O. Nierstrasz
Institut für Informatik

This dissertation can be downloaded from scg.unibe.ch.

Copyright ©2016 by Andrea Caracciolo
www.andrea-caracciolo.com

This work is licensed under the terms of the *Creative Commons Attribution – Non commercial – Share Alike 3.0 Switzerland* license. The license is available at <https://creativecommons.org/licenses/by-nc-sa/3.0/ch/>



Attribution-NonCommercial-ShareAlike

ISBN: 978-1-326-55685-3
First edition, February 2016

Acknowledgments

I warmly thank every person who directly, or indirectly, contributed to this work.

First of all I would like to thank Oscar Nierstrasz for giving me the opportunity to work at the Software Composition Group. He patiently guided me throughout this journey and encouraged me to overcome my limitations by providing insightful advises.

I am grateful to Kim Mens for reviewing this thesis and for accepting to be on the PhD committee, as well as for coming to Switzerland to join the jury of the PhD defense.

I thank Haidar, for never saying no to a cup of coffee; Andrei, for keeping me motivated; and Boris, for his playfulness. Thanks to Mircea, for the interesting discussions and his contagious optimism. Thanks to Jan, for morally supporting me during the early times in Bern; and Nevena, for bringing fresh air to the group. Thanks to Oli, for being a good climbing partner; and Leo, for his friendly laughter.

Thanks to Yuriy, Natalia, Claudio and Mohammad for bringing fresh energy to the group during the last stage of my thesis.

I am indebted to Bledar, Kirill and Oskar for their contributions to the technical implementations.

I am grateful to Fabrizio, for helping me move the first steps when I first joined the group; and Erwann, for the many thought-provoking discussions over coffee.

Thanks to Iris, for always being so helpful and kind.

I also want to thanks my parents, for walking me through the early stages of life; and my dear sister Manuela, for always being on my side.

Thanks to Manuel, Alain, Ada, Giulia, Matz, Simone and all other friends that made me feel at home in Bern. Thanks to Aiko for the nice moments shared in the past year.

Abstract

Architectural decisions can be interpreted as structural and behavioral constraints that must be enforced in order to guarantee overarching qualities in a system. Enforcing those constraints in a fully automated way is often challenging and not well supported by current tools. Current approaches for checking architecture conformance either lack in usability or offer poor options for adaptation.

To overcome this problem we analyze the current state of practice and propose an approach based on an extensible, declarative and empirically-grounded specification language. This solution aims at reducing the overall cost of setting up and maintaining an architectural conformance monitoring environment by decoupling the conceptual representation of a user-defined rule from its technical specification prescribed by the underlying analysis tools. By using a declarative language, we are able to write tool-agnostic rules that are simple enough to be understood by untrained stakeholders and, at the same time, can be automatically processed by a conformance checking validator.

Besides addressing the issue of cost, we also investigate opportunities for increasing the value of conformance checking results by assisting the user towards the full alignment of the implementation with respect to its architecture. In particular, we show the benefits of providing actionable results by introducing a technique which automatically selects the optimal repairing solutions by means of simulation and profit-based quantification.

We perform various case studies to show how our approach can be successfully adopted to support truly diverse industrial projects. We also investigate the dynamics involved in choosing and adopting a new automated conformance checking solution within an industrial context.

Our approach reduces the cost of conformance checking by avoiding the need for an explicit management of the involved validation tools. The user can define rules using a convenient high-level DSL which automatically adapts to emerging analysis requirements. Increased usability and modular customization ensure lower costs and a shorter feedback loop.

Contents

1	Introduction	1
1.1	Architecture Conformance Checking	1
1.2	Our Proposal: a Unified Approach	4
1.3	Contributions	5
1.4	Outline	6
2	State of the Art	9
2.1	Theoretical Foundation	9
2.2	Architecture Specification	11
2.3	Architecture Conformance Checking	12
2.3.1	Comparison Framework	12
2.3.2	Tools and Techniques	14
2.4	Conclusion	21
3	State of the Practice	23
3.1	Research Method	23
3.2	Learning from Practitioners: a Qualitative Study	25
3.2.1	Identified Quality Attributes	25
3.2.2	Specifying Architectural Constraints	27
3.2.3	Validating Architectural Constraints	29
3.3	Corroborating the Evidence: a Quantitative Study	31
3.4	Discussion	33
3.5	Threats to Validity	35
3.6	Related Work	36
3.7	Conclusion	37
4	A Unified Approach to Conformance Checking	39
4.1	Motivation	39
4.1.1	Scattered Functionality	40
4.1.2	Specification Language Heterogeneity	41
4.1.3	Specification Language Understandability	41
4.2	Our approach in a Nutshell	42
4.3	Formal Description	44
4.3.1	Dictō Syntax	44
4.3.2	Meta-Model	45
4.3.3	Semantic domain	46
4.3.4	Semantic Interpretation	50
4.4	Prototype Implementation	52
4.5	Discussion	54
4.5.1	Scattered Functionality	54
4.5.2	Specification Language Heterogeneity	55
4.5.3	Specification Language Understandability	56

4.6	Related Work	57
4.6.1	Architecture conformance tools	57
4.6.2	Architecture description languages	58
4.7	Conclusion	59
5	Assisted Quality Improvement	61
5.1	Dependency cycles	62
5.2	Basic Concepts	63
5.2.1	Terminology	63
5.2.2	Refactoring Strategies	64
5.2.3	Strategy Applicability	66
5.3	Our Solution	67
5.3.1	Analyze Cycles	67
5.3.2	Compute Refactoring Paths	69
5.3.3	Accept Refactoring Path	72
5.4	Evaluation	72
5.4.1	JHotDraw	72
5.4.2	Industrial Project	75
5.5	Discussion	78
5.5.1	The Package Blending Problem	78
5.5.2	Prototype Limitations and Tradeoffs	79
5.5.3	Refactoring Application	80
5.5.4	Profit Function	80
5.6	Related Work	81
5.6.1	Refactoring Candidate Identification Heuristics	81
5.6.2	Refactoring Simulation	82
5.7	Conclusion	83
6	Industrial Validation	85
6.1	Our Approach	86
6.2	Case Studies	87
6.3	Evaluation	89
6.3.1	Endorsement Seeking & Process Definition	89
6.3.2	Rule Elicitation & Formalization	90
6.3.3	Feedback automation	92
6.4	Results	92
6.5	Discussion	95
6.5.1	Expressiveness in Practice	95
6.5.2	Expressiveness in Theory	97
6.5.3	Usability	99
6.5.4	Performance and Scalability	101
6.5.5	Portability and Reusability	102
6.5.6	Extensibility	102
6.6	Related Work	104
6.7	Conclusion	105

7	The Path to Industrial Adoption	107
7.1	Background	107
7.2	Study design	109
7.3	Decision Factors	111
7.3.1	Product	111
7.3.2	Process	112
7.3.3	User	113
7.4	Adoption phases	113
7.4.1	Endorsement	113
7.4.2	Process Definition	116
7.4.3	Rules Elicitation	118
7.4.4	Deployment	121
7.4.5	Promotion	123
7.5	Discussion	124
7.6	Related Work	125
7.7	Conclusion	126
8	Conclusions	127
8.1	Contributions of this Dissertation	127
8.2	Future Research Directions	128
8.3	Summary	129
	Appendices	131
A	Qualitative Study - Interview Questions	133
B	Quantitative Study - Survey Questions	135
C	Qualitative Study - Taxonomy examples	137
	Bibliography	139

1

Introduction

Software architecture can be defined as the composition of a set of fundamental design decisions [59]. As a system evolves, its architecture tends to drift away from its intended design, leading to a phenomenon called architecture erosion [104, 26]. This phenomenon is related to the fact that software systems have a natural tendency to grow more complex and consequently less maintainable over time [73]. Architecture erosion also stems from a lack of effective and easily applicable ways to encode and preserve design knowledge [66]. This loss of knowledge can easily lead to an incoherent architecture that violates its intended design [13].

Architecture erosion can be contained by means of different strategies. De Silva *et al.* [26] present an extensive survey of all the techniques and technologies that have been proposed to prevent, detect and mitigate architecture erosion. Among others, the authors mention various approaches to ensure architecture conformance. These approaches are described as simple, widely applicable and highly effective means to prevent erosion. One of these approaches is defined as “architecture compliance monitoring” (often also referred to as “architecture conformance checking”). Techniques belonging to this category have the advantage of supporting full automation and providing continuous feedback to the user. If carefully implemented and correctly maintained, these techniques can largely contribute to minimizing the impact of architecture erosion.

1.1 Architecture Conformance Checking

Architecture conformance can be ensured using tools available on the market. Unfortunately, several studies [46, 50] show that these tools are not extensively used for checking architectural conformance at the code level. The effort of implementing and supporting them over a long period of time during system evolution is fairly high [26]. We argue that this phenomenon is largely related to the low customizability and usability of current tools.

Problem Statement

Current techniques for checking the conformance of a system with respect to project-specific architectural constraints are typically not cost-effective. Tools available on the market are often dismissed because of their low usability and the high effort required for customization.

Customizability

Software projects belong to different domains and must adhere to different policies and constraints. Current approaches are designed around strong subjective assumptions and tend to address a very narrow problem space [106, 103]. To compensate for these limitations, conformance checking tools should be designed to enable both simple and advanced customization when needed.

Jaspan *et al.* report on a case study in a major company and state that the ability to customize a tool is crucial to determine its value. [60]. They report that many tools are marketed as general purpose, off the shelf products. On the other hand, experiments have shown that default configurations often lead to suboptimal results and that proper customization (consistent with the project's goals) significantly improves effectiveness and consequently adoptability of a tool [60].

Unfortunately, several studies report that many tools are not trivial to customize [61, 6]. Users typically need to invest considerable effort in discovering which warnings are actually relevant in their project. After that, they need to adapt the tool to the obtained requirements by enabling or combining existing features and, if necessary, implementing new project-specific or organization-specific rule checkers [6]. All these tasks are highly time-consuming, technically demanding and unevenly supported by current tools. In some cases, practitioners do not even manage to adapt the tool to their needs [61]. This might be due to inherent design choices made by the developers of the tool or simply the lack of appropriate extension points.

Extensibility plays an important role also during the design phase. Woods *et al.* claim that most ADLs (architectural description languages) are quite restrictive and impose a particular architectural model on the architect [121]. As a consequence, practitioners are often induced to extend their language of choice by adding new views or constraints, or both [79]. Users value extensibility and mention it as an important, currently unfulfilled, need for use in practice [79].

Usability

Poor usability is a major deterrent to widespread usage of conformance checking tools. One way to evaluate its extent is by considering the effort required to specify new architectural constraints. Existing solutions are based on different semantic assumptions, syntactic notations and input formats.

Early approaches have been largely analyzed and criticized. Malavolta *et al.* [79] show that one of the main reasons that prevents practitioners from using ADLs can be attributed to their usability deficiencies. ADLs are described as over-formalized, complex and heavyweight. Study participants also state that ADLs should be intuitive enough to support communication among different stakeholders and at the same time aim at a sufficient degree of formality to enable automatic analysis tasks [79]. Unfortunately a lower degree of formalism typically implies more limited expressivity.

Some authors tried to trade the former in favor of the latter. Mens designed a logic meta-programming language to express structural regularities [91] and later recognized the actual obstacles that users had to face when they first approached his solution [74]. The author concludes that while having a Turing-complete specification language allows for great flexibility, in practice describing most regularities requires only the use of a small subset of the features of that language provides [74].

Other authors seem to share this opinion and propose solutions based on ad-hoc DSLs (domain specific languages). DSLs allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain [29]. If well designed, DSLs can reduce the cost of the specification process by reducing the technical and intellectual burden put on the user. Fowler argues that “business-readable” DSLs can be used to build deep and rich communication channels between technical and less-technical stakeholders¹. DSLs can be more or less usable depending on how close they are to the target domain that they intend to model. Pruijt *et al.* compare a large number of DSL-based conformance checking solutions and observe that various architectural constraints (documented in literature) can not be expressed in any of the studied solutions [106]. Other constraints can only be specified through tedious workarounds, introducing a semantic gap between the user’s mental model and the actual representation expected by the tool. This has a negative effect on the maintainability of the specification.

Another usability concern that influences the adoption of conformance checking tools is related to their operational process [98]. Current tools are highly specialized and typically handle only a small subset of the constraints that a user needs to check [106]. For this reason, practitioners often need to use multiple tools [98]. The results produced by the selected tools should preferably be integrated into a single coherent report [6]. Automating this process and the execution of the tools in general requires a considerable amount of effort [108]. The cost of workflow integration can be seen as another aspect of the usability of a tool.

Finally, practitioners estimate the usefulness of a tool by criticizing the way its results are presented. The fact that reported issues might not be easy to assess, fix or find has a direct impact on the usability of the tool [61, 98].

¹<http://www.martinfowler.com/bliki/BusinessReadableDSL.html>

1.2 Our Proposal: a Unified Approach

Thesis

To increase the cost-effectiveness of architecture conformance checking, we need a unified approach based on an extensible, declarative and empirically-grounded specification language.

As we have seen before, a good conformance checking tool should provide at the same time adequate support for customization and good end-user usability. We propose an approach that addresses both of these requirements.

Our approach (illustrated in Figure 1.1) is composed of two parts:

- **Dictō**: An empirically-grounded DSL that can be used to specify a wide range of architectural rules;
- **Probō**: A tool integration framework that enables users to verify custom defined rules using third-party tools.

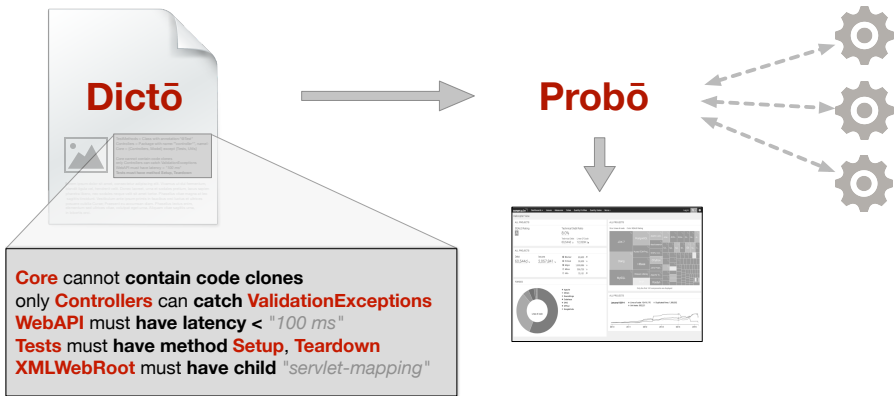


Figure 1.1: Approach overview

Dictō is a language that aims at supporting software architects in formalizing and testing prescriptive assertions on functional and non-functional aspects of a software system. Instead of dealing with multiple tool-specific formalisms, one can define several types of architectural constraints using one uniform, highly-readable, formal language (see example in Figure 1.1).

Probō is an integration framework in which Dictō constructs can be defined along with the logic required to validate the concepts they are expressing. Developers

can create a new rule template (e.g., *Method* must be executed in $< Integer$ ms) by implementing a set of pre-defined data transformers for a given target tool. These transformers must be capable of (1) generating an input specification that is consistent with the user specified invariants; (2) interpreting the results produced by the tool.

Our approach offers the following additional benefits:

- **Separation of concerns:** conceptual design (specification of constraints) and operational effort (rule evaluation) are managed separately. Technically trained developers can extend Probō to support emerging requirements while end-users benefit from a specification language (Dictō) that adapts to these extensions by automatically integrating new specification constructs. Extensions are reusable and moderately sized.
- **Support for communication:** a specification encoding valuable architectural knowledge should be accessible and readable by multiple parties, including stakeholders that do not have the skills necessary to operate the tool used to verify the expressed constraints. Our approach allows practitioners to encode project-specific constraints using a highly readable and executable specification language inspired by common documentation practices.
- **Single integration point:** Probō is designed to coordinate the interaction with multiple third-party tools. These tools generally need to be set up, automated and integrated in the current workflow according to their specific operational requirements. By providing a single unified infrastructure, we only require the user to perform these actions once.

We designed a solution that is capable of addressing important operational needs while focusing on maintaining good usability. Another aspect that affects usability is the quality of the results produced during the analysis process. Violations reported at the end of this process should be easy to understand and assess. To support these requirements, we implemented a tool (*Marea*) that offers detailed suggestions on how to remove detected package cycles. By introducing similar tools and integrating our results into various common process-automation platforms (e.g., SonarQube, Teamcity), we gained a better understanding of what practitioners need and how conformance checking solutions can be improved.

1.3 Contributions

The main contributions of this dissertation are:

1. **An empirical analysis on how practitioners manage quality requirements.** Understanding the practical implications of describing, maintaining and validating quality requirements is essential to build cost-effective conformance

checking tools. In our work, we analyze the state-of-the-art in industrial practice and identify various challenges that prevent practitioners from automatically validating architectural constraints.

This study has been published in a conference [16].

2. **A validated approach for specifying and evaluating architectural rules.** Studies show that architecture is often checked for conformance using non-automatic techniques. This is mostly due to the high cost involved in integrating, customizing and maintaining solutions currently available on the market. We propose a novel approach that addresses these issues. Our approach consists of *Dictō*, a declarative practice-inspired DSL for specifying architectural rules, and *Probō* an integration framework that evaluates those rules using third-party off-the-shelf analysis tools.

We realized a prototype that supports rules for system implemented in Java, PHP and Smalltalk². This approach has been published in a conference [19], a magazine [18] and a workshop [17].

3. **A case study that demonstrates the benefit of actionable results for conformance checking.** Architectural violations should ideally be easy to assess, fix and locate within the target code base. All major tools satisfy the latter requirement but fail in providing contextual hints on how to fix complex violations. In our work we present *Marea*, a tool that simulates potential refactoring strategies for removing user-specified cycles and provides practical feedback to the user. We realized a prototype for systems implemented in Java [1] and evaluated its utility in multiple case studies. This approach has been published in a conference [20].
4. **An empirical analysis on the adoption of conformance checking solution in industrial organizations.** Practitioners are reluctant to invest in new solutions because of the difficulty of estimating the cost-effectiveness of a new tool, scarce resources allocated to quality related activities and a general lack of expertise in the domain. In this work we investigate the dynamics involved in choosing and adopting a new automated conformance checking solution within an industrial context.

This study has been published in a workshop [21].

1.4 Outline

This dissertation is structured as follows:

²<http://scg.unibe.ch/dicto/>

Chapter 2 – We discuss the related work of this thesis. We present various solutions to architecture conformance checking and analyze the main shortcomings in the context of each approach.

Chapter 3 – We investigate the current state of practice by analyzing material collected during a series of interviews and a questionnaire.

Chapter 4 – We propose a novel approach to architecture conformance checking. The approach is designed to be both adaptable and convenient to use.

Chapter 5 – We discuss the value of actionable results by analyzing the benefit of providing operational suggestions to the user.

Chapter 6 – We validate our approach to architecture conformance checking on three distinct industrial projects.

Chapter 7 – We analyze the dynamics involved in choosing and adopting an automated conformance monitoring solution.

Chapter 8 – Concludes the dissertation and outlines future work.

2

State of the Art

In this chapter, we survey the state of the art in supporting architecture conformance checking. First, we discuss the theoretical foundation of this dissertation and provide a clear definition of the concepts that will be discussed in the remainder of this work. Second, we review existing approaches for architecture conformance checking and categorize them based on their ability to adapt to user's needs (customizability) and their convenience of use (usability).

2.1 Theoretical Foundation

The study of software architecture is the study of how software systems are designed and built [114]. A system's architecture is the sum of all principal¹ design decisions that were taken during the development process to achieve some desired quality attributes. In this dissertation, we define software architecture as follows.

Def. 1 (Software Architecture). *The set of design elements that have a particular form, explicated by a set of rationale. – Perry and Wolf [104]*

The above definition distinguishes between three kinds of architectural *elements*: processing elements; data elements; connecting elements. These elements are organized according to a suitably designed *form*, consisting of properties and relationships. Properties and relationships are used to define constraints on architectural elements. Constraints are determined by considerations ranging from basic functional aspects to various non-functional aspects such as economics, performance and reliability. The *rationale* explicates the satisfaction of the system constraints.

Fielding provides a similar interpretation, describing architecture as a set of elements constrained in their relationships in order to achieve a desired set of fundamental design properties [37].

¹"Principal" implies a degree of importance that grants a design decision "architectural status". How one defines "principal", according to Taylor *et al.*, depends on what the stakeholders define as the system goals.

Bass *et al.* define architecture as a set of software elements characterized by externally visible properties and the relationships existing among them [8]. By “externally visible properties”, the authors refer to assumptions other components can make of a component, such as provided services, performance characteristics, fault handling. In our interpretation, assumptions can be intended as contracts when coupled with a complementary number of constraints that ensure their validity.

Taylor *et al.* define architecture as the set of principal design decisions governing a system [114]. This definition is closely related to the previously named concept of *rationale* (defined by Perry and Wolf as a set of choices made in defining an architecture). In our interpretation, we assume that design decisions eventually need to be reified into structure by means of constraints.

Based on the previous definitions, we can conclude that architecture is partially defined through design constraints. Architectural design constraints have been often associated to patterns and styles [45, 8]. In this dissertation we opt for a broader definition of the term.

Def. 2 (Architectural Design Constraint). *A design constraint defines what the system, or parts of it, may not do. – Bosch [13]*

Architecture documentation can be at the same time prescriptive and descriptive [22]. For some stakeholders it prescribes what should be true by placing constraints on decisions to be made. For other audiences it describes what is true, by describing decisions already made, about a system’s design. If constraints are intentionally described in a prescriptive manner, they are typically referred to as rules.

Def. 3 (Architectural Design Rule). *A design rule specifies a particular way of performing a certain task. – Bosch [13]*

Architectural rules are typically discussed and specified in industrial practice, but are only vaguely outlined in academic literature. Several studies analyzed which kind of quality attributes are more or less relevant to practitioners. How these requirements are concretely pursued, remains an open question. In chapter 3, we report on an empirical study which provides a taxonomy of the rules commonly considered in the context of an industrial project. We also investigate which of the techniques that are currently available (see section 2.3) are actually used to check those rules.

When the implementation drifts away from its intended design and fails to comply to the architectural constraints defined in its specification, we observe a phenomenon called architectural erosion [104]. This phenomenon can be controlled by periodically checking if the implemented architecture complies with the intended architecture.

Def. 4 (Architecture Conformance Checking). *Establishes the means to validate whether the implementation is faithful to the intended architecture during both the development and subsequent maintenance phases of a system. – De Silva et al. [26]*

In this dissertation we argue that current solutions for conformance checking are not cost-efficient because they fail to provide at the same time good usability and extensive customizability. In chapter 4, we propose a novel solution that addresses both these aspects by offering a DSL which automatically adapts itself to the extensions applied to the underlying analysis infrastructure.

2.2 Architecture Specification

Architecture might be specified in different forms and notations depending on the purpose they serve. Considerable effort has been invested in defining formal notations for supporting the unambiguous description and analysis of an architecture.

Medvidovic *et al.* describe several languages that can be used to this end (*e.g.*, Rapide [75], ACME [42], Darwin [76], AADL [36], UML [12]). These languages are commonly referred to as ADLs (architectural description languages) or ALs (architectural languages) and can be used to describe an architecture in terms of properties and relations. Most of the proposed approaches (with the exception of UML) originated from academic research projects and are often criticized for their formal notation and weak tool support [79]. Medvidovic *et al.* [85] state that ADLs need to be extensible in order to accommodate the definition of new abstractions and reflect the business context and application domain of the target project. Only few languages (*e.g.*, UML, AADL) have been used in some form of experimental code transformation process [86, 69], but in general it's safe to assume that they are more frequently used for design specification only.

Some of those modeling languages also provide dedicated constructs for the definition of constraints. Constraints are defined to complement the specification of the architectural model. Aesop [41], for example, allows users to declare topological invariants (*i.e.*, allowed dependencies) as part of the definition of an architectural style. Such an invariant can be used to perform a structural check of the model and verify that entities of a certain type are only connected to other entities of a certain type through pre-defined ports. SADL [95] and Wright [4] support the declaration of first-order logic predicates for the definition of similar invariants. As in Aesop, invariants are designed to control the type of entity defined as end-points of communication links. Rapide [75] offers support for behavioral constraints that can be used to define run-time invariants (*e.g.*, message values, invocation sequences, abstract state). ACME [42] relies on a separated constraint language called Armani. This language allows the writing of first-order logic predicates and can be used to define type constraints (like in previously mentioned languages) and heuristics (*i.e.*, numerical thresholds for limiting the size of specific parts of the model). REAL is another constraint language designed for AADL [36]. This language is used to verify type checking invariants and constraints on the graph structure. Finally, OCL [100] is a language for defining constraints in UML models. UML is used in order to describe a system in terms of entities and relationships. Entities can be annotated

(e.g., stereotypes, fields, methods) and relationships can be either static or dynamic (e.g., class diagrams, sequence diagrams). OCL provides a set of functions that can be used to navigate the graph structure of a model and create assertions on identified elements. Assertions can be defined to constraint property values, entity types and relationship cardinalities. OCL is a very expressive language which has been taken as inspirational source in several other approaches for defining architectural constraints [116, 84].

We can conclude by saying that all the solutions mentioned in this section are of secondary importance to the vast majority of practitioners. Current approaches can only be used to reason about abstract representations that are completely disconnected from the implementation. In such circumstances, a complete description of the architecture is only valuable if considered in the context of a model-driven software development process. Since this type of process is not yet largely practiced in industry, we will refrain from further analysis.

In chapter 3 we report on how architectural constraints are actually specified in industrial practice. Our study shows that practitioners write architectural descriptions with a clear goal and audience in mind and rarely rely on formal notations.

2.3 Architecture Conformance Checking

Software architectures are designed to guarantee a certain set of fundamental functional and non-functional requirements. To ensure the correct realization of these requirements, one may check that the system's implementation complies to the constraints defined in its architecture. This process is called "architecture conformance checking" and can be performed with the aid of various techniques and tools.

2.3.1 Comparison Framework

Conformance checking tools are used to test whether a given set of invariants are correctly enforced in a target system. These tools can be evaluated and classified according to numerous dimensions. In this dissertation, we choose to categorize conformance checking techniques based on their degree of *usability* and *customizability*.

Usability – We define usability as the ease of a specific tool or technique to be understood, learned and used by the user [58]. Conformance checking tools are used to evaluate the correctness of a given set of user-defined rules. These rules need to be opportunely formalized using a textual or graphical description language. The efficiency of such an activity strongly depends on the understandability and learnability of the prescribed input notation. We distinguish between techniques based on the following categories of specification languages:

DSL (domain specific language): A domain specific description language is typically designed to formulate concepts in a form and notation that is familiar to the user. DSLs typically do not assume any kind of specific technical knowledge and are mostly declarative.

GPL (general purpose language): Some specification languages are built as extensions to other programming languages. While this ensures higher expressive power it also discriminates against stakeholders that might not have the necessary skills to approach the underlying technologies.

We argue that DSLs offer higher usability than GPLs, based on the following considerations:

- Declarative languages are more *maintainable* than imperative languages. They enable a form of specification that is naturally closer to the way human beings would express a constraint without necessarily requiring the formulation of a validation strategy [51].
- DSLs are more *understandable* than GPLs. They are semantically close to the problem domain and are typically more concise than their general purpose counterpart [29].

Customizability – We define customizability as the capability of a tool or technique to adapt to changing requirements by enabling a specified modification to be implemented [58]. Conformance checking tools should accommodate the specification and analysis of constraints that are specific to the project at hand (as discussed in chapter 1). The ability to support this task depends on the expressive power of the notation used for specification and on the extensibility of the technique in use. In the remainder of this chapter we distinguish between techniques that are:

Configurable (adaptable by configuration): These techniques are designed to adapt to different contexts only within the limits of their configurability. Any further attempt of extending the provided functionality is not explicitly supported.

Extensible (adaptable by extension): These techniques are based on some kind of extensible architecture that supports the integration of new functionality within the general capabilities of the tool. Extension mechanisms might be explicitly documented or indirectly deduced by inspecting the implementation's code. Extension should not require the alteration of existing program logic.

For the purpose of our classification we will rank existing techniques according to their capability of fulfilling potential user requirements. This means that techniques supporting more radical extensions are ranked higher than those that only support foreseen changes.

2.3.2 Tools and Techniques

After defining a comparison framework, we categorize currently known conformance checking techniques according to the previously defined dimensions (Figure 2.1).

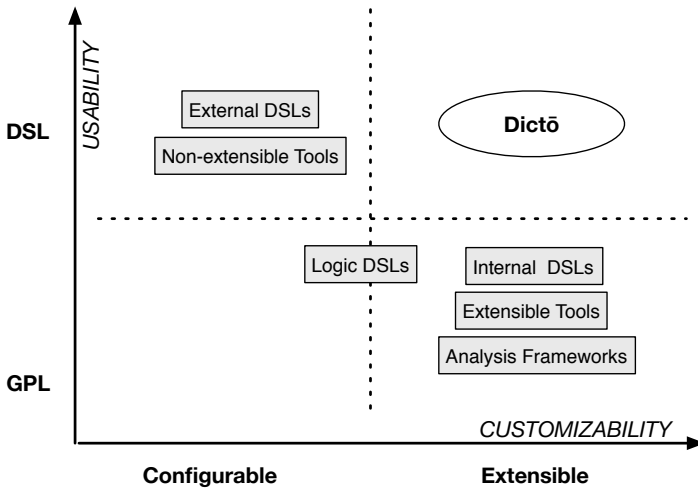


Figure 2.1: Comparison of conformance checking techniques

The tools and techniques analyzed in this chapter are only considered from the point of view of the features they offer and the observable properties that characterize them. To the best of our knowledge, there is no empirical study showing whether and how these instruments are regularly used in practice. To clarify this aspect, we performed a survey to assess how practitioners check for architecture conformance (see chapter 3).

External DSLs (textual notation / standalone rule specification)

Solutions belonging to this category are capable of checking rules specified in a dedicated textual DSL. These DSLs vary in expressive power and domain. *DCL* [115], *TamDera* [48], *Classcycle*², *InCode.Rules* [80], *Macker*³ are designed to define constraints on code dependencies (e.g., accesses, declarations, extensions). *InCode.Rules* can also be used to identify classes affected by specific design flaws (e.g., god class, data class). *DCL* and *Classcycle* allow the user to define rules as single statements with a clearly defined syntactical structure. *TamDera* allows the user to define hierarchical concepts, which slightly improves the modularity of the specification. *InCode.Rules* supports rule composition: each rule can be used to define an exception

²<http://classycle.sourceforge.net>

³<https://innig.net/macker/>

to another rule. Macker relies on an XML-based specification language that is comparable to DCL in terms of expressivity.

*Lattix Architect*⁴ is a tool which has a graphical interface showing the dependencies in a system through a DSM (dependency structure matrix). Architects can define new constraints in which they allow or forbid dependencies between different types of entities (*e.g.*, interfaces, classes, packages). The notation used to accomplish this task is similar to the one provided by other tools mentioned in this category (*e.g.*, DCL, Classcycle). Developers can identify rule violations and cycles by visually navigating the reverse engineered DSM.

All the tools belonging to this category are characterized by high usability and a well defined strict specification language. The authors of DCL claim that their language is more usable than other logic inspired alternatives (see category: Logic DSL) [115]. Those are supposedly based on a more complex and heavyweight notation and offer poor performance. A similar claim is made by Lozano *et al.*, who recognize the difficulty that typical users encounter when approaching solutions that require a basic understanding of logic programming [74]. Terra *et al.* also compare DCL to alternative solutions based on reflection models and dependency structure matrices (see category: Configurable Tools), stating that their language is more expressive and handles a wider set of constraint types [115]. A similar claim is also made by Marinescu *et al.* [80]. All the mentioned languages are declarative and do not require any specific programming skill. None of the mentioned solutions offers any suitable extension mechanism. Rules can be defined by using the constructs offered by the supported notation. Each attempt to further customize the tools requires the modification of the specification language parser as well as non obvious changes to the core logic of the analyzer.

External DSLs (textual notation / embedded rule specification)

Some practitioners prefer to keep the specification of architectural rules as close as possible to the elements that they constrain. DSLs like *ArchFace* [118], *ArchJava* [3], *CCEL* [30] enable users to specify rules as an integral part of their code in the form of comments or program statements using a pre-processed external DSL. In most of the cases, rules are checked at build-time using a dedicated analyzer that reports all encountered inconsistencies. *ArchFace* uses a slightly different approach and compiles rules into contracts (*i.e.*, aspects) that prevent undesired runtime behavior (*e.g.*, disallowed interactions between objects).

Having rules directly embedded in the source code should ideally reduce the gap between a prescriptive specification and the corresponding target implementation. Developers are directly exposed to architectural rules during their regular development activity. They can adapt rules to the current implementation and vice-versa. This approach guarantees a closer feedback loop which is reduced even more if rules

⁴<http://lattix.com>

are checked at compilation-time. On the downside, accepting that those that have to follow the rules are also the ones that maintain them might introduce a bias in the process. Additionally, introducing new language constructs is likely to break tool compatibility (e.g., IDE, analysis tools).

All the mentioned approaches do not support any explicit extension mechanism. They can be used to formulate complex conditions by combining pre-configured language elements, but they cannot be easily extended in their functionality.

External DSLs (textual notation / standalone query specification)

Architectural rules can also be encoded as queries. DSLs, such as *CQLinq*⁵ and *Semmlle .QL* [25], are languages that enable code exploration and that can be used to check architectural constraints. Users may define invariants in the form of queries which are not expected to yield results. In case a result is found, this is reported to the user as an architectural violation.

Both *CQLinq* and *Semmlle .QL* are based on a SQL-inspired syntax. A query consists of a conditional select statement where code entities of various nature (e.g., Types, Methods, Namespaces) are compared with specific values. *Semmlle .QL* supports the definition of new accessory macros that can be introduced to increase the readability of the specification. Both solutions are implemented as IDE extensions but could theoretically also be run standalone.

Passos *et al.* [103] compare *Semmlle .QL* to similar solutions such as *SAVE* and *Latix Architect*. They conclude that *Semmlle .QL* offers an expressive and intuitive syntax but, compared to the other solutions, lacks adequate support for the representation of high-level concepts (e.g., logical components). This limitation cannot be circumvented, given the lack of support for extension. Architectural rules can only be defined by composing supported operators and expressions.

External DSLs (graphical notation / standalone rule specification)

Solutions belonging to this category are characterized by high specialization and limited configurability. Most tool supporting a graphical specification notation are designed to check dependency constraints (using the reflexion models technique [97]). *Sonargraph*⁶, *SAVE* [31], *Structure 101*⁷, *ConQAT*⁸ are some examples. These tools require developers to draw a high-level model of their architecture in which they specify components (opportunistically mapped to packages through regular expressions) and allowed/forbidden dependencies. Once defined, this model can be compared with an actual target implementation to detect potential inconsistencies.

⁵<http://www.ndepend.com>

⁶<https://www.hello2morrow.com/products/sonargraph>

⁷<http://structure101.com>

⁸<https://www.cqse.eu/en/>

These tools are mostly commercial and only require minimal initial setup effort. Adaptability can be achieved by tuning the foreseen configuration options. Pruijt *et al.* [106] compare all of the mentioned tools by encoding a reference set of rules in each of the provided specification languages. Their experiment shows that more complex rules are either not supported or very cumbersome to describe. This shows that the tools belonging to this category are designed around a clear set of assumptions that highly constraints the type of rules that are supported. Limited expressivity has the advantage of increasing the overall usability of the tool. On the other side, these tools do not explicitly support extension, and adapting a solution to unforeseen needs is either very inconvenient or impossible.

Non-extensible Tools

Tools belonging to this category offer various kinds of analysis capabilities that can be used to assess architecture conformance. The category comprises products like: *JMeter*⁹, *SoapUI*¹⁰, *LISA*¹¹. The execution process of these tools can be configured through a convenient graphical interface. None of these solutions was designed to explicitly support extensibility. The addition of new analysis features would require major adaptations to the implementation. As a consequence, these tools are only useful as long as the system under analysis is consistent with the assumptions upon which the tools were built. Being bound to a graphical interface offers benefits in terms of usability but also increases the cost of customization.

Logic DSLs

Architectural rules can naturally be translated into a language based on first order logic. Languages like *SOUL* [93], *LogEn* [32] and *SCL* [56] are good examples of practical solutions that can be used for conformance checking.

SOUL is a Prolog-inspired internal DSL implemented in Smalltalk. A set of pre-defined high-level predicates can be used to create architectural rules or define new predicates. Pre-defined predicates are evaluated using dedicated analyzers. The representation of the target architecture can be enriched by adding new facts to the fact base.

LogEn is an internal DSL implemented in DataLog, a subset of Prolog. Rules and generic predicates are conceptually specified in the same way as in *SOUL*. Facts are automatically extracted from the source code using a static analyzer. Source code entities can be grouped in logical sets (called ensembles) programmatically using a dedicated predicate or declaratively using specific annotations in the analyzed code.

⁹<http://jmeter.apache.org/>

¹⁰<http://www.soapui.org/>

¹¹<http://www.itko.com>

SCL is an external DSL inspired by *OCL*. The language is used to define first-order logic formulas that can be automatically evaluated against the source code of a target system. Users can express structural constraints in a declarative and language-independent notation using pre-defined functions and predicates.

Languages belonging to this category have the advantage of being inherently extensible within the boundaries set by the underlying language model. In fact, users can define new concepts by declaring and combining facts and predicates. This form of extensibility allows developers to adapt the notation to the specific vocabulary required to describe their architecture. Unfortunately this flexibility comes at the price of usability. In fact, these languages entail programming capabilities which typically go beyond the skills possessed by average software engineers. Lozano *et al.* [74] observe that *SOUL*, despite being a declarative language, has repeatedly proven to be rather impractical in a real scenario. Developers approaching the language had to get out of their comfort zone and learn a new programming paradigm. The authors recognize that a more lightweight and limited DSL (*i.e.*, as opposed to a Turing-complete language) is more appropriate for the specification of architectural rules.

Internal DSLs

Some tools have been developed based on the intuition that an architectural specification should be put close to the objects that it describes. This strategy is considered to reduce the risk of incurring in outdated constraints. The assumption is that code is a primary vector of information and will always be maintained more actively than any other non-executable document. As a consequence, architectural constraints should be encoded as code invariants using the same language employed for development.

CoffeeStainer [11], defines constraints as code snippets which are checked at compile-time using the reflective capabilities of the host language (*i.e.*, Java). Code snippets cannot be executed at run-time because they are defined within a comment block.

PDE [15] is another solution that exploits the features offered by the host language (*i.e.*, Java) to express architectural constraints. With *PDE*, developers can specify rules using regular annotations. Rules are checked at compile-time and can be used to inhibit hierarchical relationships (*e.g.*, forbid extension, implementation or overriding of classes and methods) and dependencies (*e.g.*, forbid instantiation or referencing to a specific class). Both *CoffeeStainer* and *PDE* are completely non-invasive techniques that can be used without altering the semantics of the system and without breaking tool compatibility.

uContracts [74] is an internal DSL designed to express a broad range of structural constraints for Smalltalk systems. It can be used to define boolean contracts for specific code entities or groups (*e.g.*, class hierarchies). The user can express constraints on naming conventions, code idioms and usage protocols. Rules can be formulated by

combining different types of pre-defined boolean constructs. The solution is tightly coupled to its host environment and uses the reflective capabilities of the language in which it's written to evaluate the specified rules. Rules are specified outside the scope of the checked project. No explicit extension mechanism is defined.

Solutions belonging to this category are generally practical to define and easy to maintain. On the other hand, opting for a technical specification immediately excludes any non-technical stakeholder from the definition and specification process. Architectural rules can only be defined through a person who is familiar with the programming language in which the DSL is implemented. This might not be an issue in open source communities with flat hierarchies, where everybody is generally equally involved in the making of the system and sufficiently acquainted with the technologies and tools used in the process. But, according to our experience, more structured teams working in an industrial setting might have more defined roles. In this context, stakeholders in higher positions (such as software architects, who typically design architectural constraints) are not always directly involved in the development process or might not have the time to handle implementation related issues that are not directly affecting the stability of the system.

Moreover, keeping rules too close to the context where they should be applied has also other disadvantages. Developers could be tempted to adapt the rules to the implementation or to avoid specifying new rules when new functionality is added. Transversal rules, involving entities defined in different locations spread across the code base, could also be hard to specify if the chosen technique expects them to be defined in the context where they apply (*e.g.*, dependencies can be forbidden by annotating the classes involved in the constraint; naming conventions should better not be defined by annotating every entity that needs to be constrained). This last problem can be solved by placing rules in an orthogonal context, as done in `uContracts`.

Extensible Tools

Several commercial tools available on the market are based on a plug-in architecture. *Sonarqube*¹², *Findbugs* [7], *Checkstyle*¹³, *Pmd*¹⁴ are popular examples of tools belonging to this category. These tools are mostly used as off-the-shelf solutions designed to look for violations of recommended programming practices.

They can be run with minimal setup overhead. Advanced users can extend the provided functionality by implementing new detectors that will be executed at analysis-time. Custom detectors are implemented in Java as AST visitors. *Sonarqube* and *Pmd* also support XPath rules. These rules are used to enforce conditions on specific nodes of the AST and report violations when the query yields any result.

¹²<http://www.sonarqube.org>

¹³<http://checkstyle.sourceforge.net>

¹⁴<https://pmd.github.io>

Sonarqube offers an intuitive graphical user interface where developers can enable pre-built rules or write custom declarative constraints (using XPath). If the user needs to perform more complex analysis or to integrate the functionality of a third party tool, she will have to develop a new extension module that will be added as-is to the analysis solution. This means that new rules need to be implemented, configured and maintained by experts that have sufficient technical expertise to handle the development of a Java plugin besides having a good understanding of the provided extension API. This would might become a major obstacle if end-users expect to configure the implemented extensions from the main user interface.

Checkstyle, *Findbugs* and *Pmd* can adapt their behavior based on the content of an xml configuration file. As for *Sonarqube*, plugins are AST visitors implemented in Java. *Checkstyle* plugins can be parametrized though fields declared within the xml configuration file. This allows end-users to define extensions that can be adapted to the analysis context. One downside of this approach is that rules are not self-documenting. In fact, they are only described by the name of the plugin and (in the case of *Checkstyle*) a set of named values. The semantic of the rule is completely embedded in the implementation of the plugin and needs to be separately documented.

To summarize, all the mentioned tools offer rich customization options but are based on a configuration languages with very low expressivity. To define a rule, one can only select the most appropriate analysis module and adapt a set of predetermined parameters. Usability is improved in those tools where the user can define rules using the more expressive XPath query language.

Analysis Frameworks

Full customizability can be obtained with GPL (general purpose language) analysis frameworks. Solutions like *Moose* [99] and *Rascal* [62] offer the possibility to build complex analysis routines using a Turing-complete programming language.

Moose is implemented in Smalltalk and can be used to query, manipulate an visualize models extracted from object-oriented systems. These models are in-memory object graphs that represent basic entities (*e.g.*, classes, packages) as well as relationships (*e.g.*, method invocations, type references). Architectural rules can be defined as algorithms that perform arbitrary complex lookups on the reverse engineered model.

Rascal is a meta-programming language with IDE support. Users can define custom visitors that are capable of navigating a given AST and filter out a set of elements that match programmatically defined criteria. The resulting analysis routine can be invoked manually or from command line.

Both solutions mentioned in this category are highly customizable but require extensive technical knowledge. The tools do not offer any form of support for defining architectural rules that can be easily maintained and checked on a regular basis.

2.4 Conclusion

In this chapter we introduced the main theoretical concepts that define the context of this dissertation. Later, we categorized and evaluated the main approaches currently available for checking the architecture conformance. These approaches are categorized as follows:

- *External DSLs*: Constrained, tool-specific languages with support for simple and moderately complex predicates.
- *Non-extensible Tools*: Specialized and not extensible off-the-shelf tools.
- *Logic DSLs*: Formal languages that support the specification of rules as first-order predicates. Extensibility is limited to the boundaries inherited from the underlying logic formalism.
- *Internal DSLs*: Are used to define contracts directly within the source code.
- *Extensible Tools*: Off-the-shelf tools based on a plugin architecture. New functionality is only partially adaptable by operators.
- *Analysis Frameworks*: Can be used to define complex architectural rules. Their use is limited to developers with specific technical skills.

As we see, none of the approaches belonging to the above mentioned categories is at the same time fully customizable and reasonably usable. End users need customizability as much as usability. The capability to adapt to a specific problem scenario is fundamental, but becomes irrelevant if the specification that eventually needs to be written to express the architectural concern is hard to maintain.

In the remainder of the thesis, we investigate current practices for checking architecture conformance. Based on our observations, we design Dictō, an intuitive and executable DSL which adapts to new requirements by automatically adjusting to new backend extensions. Our solution aims at satisfying both needs discussed in this chapter offering full extensibility by preserving good end-user usability.

3

State of the Practice

In this chapter we set out to survey whether the specification of architectural constraints is a common practice in IT companies. We want to understand whether this activity is systematic and supported by tools and processes or rather based on personal assumptions and using makeshift tools. Finally, we investigate whether architectural constraints, given their importance, are also automatically validated as the software system evolves.

Previous studies [36, 43, 77] propose solutions for specifying architectural invariants. Other studies [5, 105, 112, 49] rank non-functional qualities (*e.g.*, performance, usability, availability, *etc.*) by carrying out surveys. In neither case is effort made to explore quality attributes from the point of view of practitioners.

In our study we focus on the following research questions:

- **RQ1:** What kind of architectural constraints do architects define in practice?
- **RQ2:** How are architectural constraints specified?
- **RQ3:** How are architectural constraints validated?

To answer these questions, we use empirical methods to identify quality attributes that practitioners consider when designing their architecture. Furthermore we analyze how practitioners specify architectural constraints in their documentation and explore the various techniques that are used for validation.

3.1 Research Method

This study uses a mixed research methods strategy: *sequential exploratory design* [23]. This approach consists of two different research methodologies: a qualitative investigation followed by a quantitative validation survey which triangulates the results of the first.

In the first study, we focused on collecting qualitative data. The goal of this study was to gain a possibly comprehensive overview of the state of practice in the definition and validation of architectural constraints. The questions have been iteratively

#	Role	Org.	Project (domain; type)	team size
A	CEO, architect	C1	government / enterprise	<5
B	business manager	C2	government / enterprise	10-50
C	project manager	C3	insurance / enterprise	>50
D	architect	C4	logistic / enterprise(integration)	<5
E	developer	C4	logistic / enterprise(integration)	<5
F	CTO	C5	banking / enterprise	>50
G	architect	C2	government / enterprise	5-10
H	architect	C2	government / enterprise	10-50
I	architect	C6	logistic / enterprise(migration)	>50
J*	developer	C2	government / development support tool	<5
K	architect	C5	banking / enterprise	5-10
L	architect	C6	transportation / control systems	5-10
M*	developer	C5	banking / source code analysis	>5
N*	architect	C5	banking / development support tool	5-10

Table 3.1: Interview study participants. Candidates with an asterisk worked in projects aimed at supporting architectural design. The remaining candidates worked as software architects or project managers in medium to large projects and have more direct experience in architectural design.

refined by conducting three internal pilot interviews with PhD and master students with professional experience in the field. The final list of questions, used as loose guideline for the actual interviews, can be found in Appendix A. Fourteen people working for six different organizations agreed to participate in our study (Table 3.1). More than 70% of the participants have been contacted indirectly through an intermediary and had no relevant links to the academic community. The remaining subjects were contacted directly and belonged to our industrial collaboration network. All interviews were carried out independently, leading to a set of complementary and partially overlapping observations. A total of approximately 18 hours of conversation have been recorded.

The main outcome of this qualitative study was the list of quality attributes presented in Table 3.2. These quality attributes were inferred by analyzing the interviews and synthesizing the main concerns using coding techniques [94]. To support this activity, we identified and labeled architectural constraints in interview transcriptions as well as the documentation files (*i.e.*, Software Architecture Documents, Developer guidelines) that we collected at the end of several interview sessions. To gather more evidence that the observations coming from the first study actually reflected the state-of-practice of a broader community, we created an e-survey. Over a time span of two months we collected 34 valid and complete responses. Invitations were sent to professionals selected among industrial partners and collaborators (*i.e.*, convenience sampling method), including people involved in the first phase of the study. The survey was also advertised in several groups of interest related to software architecture hosted by LinkedIn and on Twitter (*i.e.*, voluntary sampling method). Survey participants were asked to specify whether the quality attributes

identified in the first study were ever encountered in a past project, their perceived level of importance (on a scale from 1 to 5, with 5 being the highest), the formalism adopted to describe them and the testing tool used for their validation. More details about the survey can be found in Appendix B.

3.2 Learning from Practitioners: a Qualitative Study

During interviews, we tried to elicit a possibly wide range of distinct architecturally significant quality attributes. We asked our respondents to enumerate those concerns that could be considered fundamental for their architecture. For each of those, we asked them to describe their main properties and the form in which they were typically specified. Table 3.2 shows all identified quality attributes. For each quality attribute, we also present additional details collected during our quantitative study (columns 3-6 in Table 3.2).

Quality attributes are categorized based on the closest matching ISO-25010[57] quality characteristic. For simplicity's sake, we decided to pair each attribute with one single category. A list of explanatory constraints for all presented quality attributes can be found in Appendix C.

3.2.1 Identified Quality Attributes

We now comment on the identified quality attributes.

Performance: performance was often mentioned as being a key concern. Requirements on *response time* and *throughput* are commonly part of the acceptance criteria defined with the customer at the beginning of a project. Several respondents (*e.g.*, **A**, **B**) define latency requirements on the execution of specific tasks (*e.g.*, The system has to answer each request within 10 ms). Others (*e.g.*, **A**, **D**) set limits for the accepted throughput (*e.g.*, The system must be able to execute a certain task 10'000 times per hour). These requirements are often validated by collecting timestamps during execution or simulating high traffic load with a script. *Hardware infrastructure* requirements, specifying the hardware resources required to support a specific software implementation, also play a role in determining performance.

Compatibility: multiple interviewees (**B**, **F**, **J**) referred to *communication* and data transmission as one of the most important aspects in their architecture. **F** built a client simulator to test conformance with the prescribed communication protocol and check syntactical/semantical data consistency. **N** defined a guideline stating that data has to be passed from one layer to the other using Data Transfer Objects. **G** wrote a detailed specification of all service interfaces composing his application (*signature* attribute). This included details regarding accepted parameter values and activity diagrams describing the message exchange protocol. Interoperability between

Category	Quality Attribute (Internal / External / Process)	Importance			Fam.	Form. Not.
		Q ₁	Q ₂	Q ₃		
Performance	Response time (E)	3	4	5	15%	14%
	Throughput (E)	3	4	4	26%	13%
	Hardware infrastructure (I)	2	3	4	29%	0%
Compatibility	Signature (I)	3	4	4	18%	52%
	File location (I)	1	3	4	29%	18%
	Data structure (I)	2	3	4	29%	47%
	Communication (I)	2	4	4	15%	22%
Usability	Visual design (E)	2	3	3.5	9%	21%
	Accessibility (E)	1	2	3.5	50%	0%
Reliability	Availability (E)	4	4	5	15%	14%
	Recoverability (E)	2	3	5	32%	5%
	Data integrity (I)	3	3	4	18%	23%
	Event handling (I)	2	3	4	35%	25%
	Software update (P)	1	2	3	59%	0%
Security	Authorization (E)	4	4	5	3%	23%
	Authentication (E)	3	4	5	21%	12%
	Data retention policy (I)	2	3	4	12%	13%
Maintainability	Meta-annotations (I)	1	3	4	32%	39%
	Code quality (I)	2	3	3.5	15%	19%
	Dependencies (I)	2.5	3	4	18%	53%
	Naming conventions (I)	2	3	3	12%	38%
Portability	Software infrastructure (I)	3	3	4	24%	8%

Table 3.2: Taxonomy of architectural constraints (grouped by supported quality characteristic). Columns (from left to right): Matching quality characteristic; Architectural constraint; Evaluated importance (first, second and third quartile); Participants who encountered the constraint in a previous project (familiarity); Participants who specified the constraint using a formal notation. Columns 3-6 contain data collected during our quantitative study.

different components and tools often requires files to be placed into pre-determined folders or structure files according to a given shared schema (*file location* attribute).

Usability: *visual design* and compliance to *accessibility* guidelines were mentioned as typical requirements for application front-ends. **H** developed a web interface that had to conform to a set of rules defined in the corporate visual style guide. This requirement was satisfied by defining global stylesheets and forcing their inclusion into all related applications.

Reliability: robustness and fault-tolerance are important features for almost any kind of application. **H**'s application was required to guarantee 96% *availability* and a clear recovery procedure was defined for each type of fault that was likely to oc-

cur. *Data integrity* is also a major concern. **K** managed to maintain internal data consistency by defining data type classes for all supported business value types. **H** and **G** constrained field values specifying Hibernate or Spring formatting annotations. Specific rules were also defined to regulate strategies for *handling events* (e.g., exceptions, notifications) and *update software packages* (e.g., libraries).

Security: security is also considered critical and is often tested thoroughly. Verification becomes a necessity when the system is directly exposed to a large untrusted audience. Testing seems to have lower priority if the application is just deployed within an intranet (**E**). Most of the time, widely known frameworks (e.g., JAAS) are used to implement *authentication* and *authorization* rules.

Maintainability: class *dependencies* and syntactic code invariants are commonly considered tightly related to software architecture. **H** even claims that “dependencies between modules are the main characteristic of a software architecture”. Requirements on these two aspects are defined to support architectural principles (loose coupling, high cohesion) and minimize the cost of future maintenance.

Portability: requirements related to *software infrastructure* configuration are common. Prescriptions on technologies to be adopted can be found in almost every specification document. **J**, for example, specifies that the “persistence layer” of his application must use Hibernate as a persistence framework. Software infrastructure requirements are often related to rules addressing compatibility issues (i.e., file location, data structure).

3.2.2 Specifying Architectural Constraints

All the participants of our study describe their architectural constraints in one or more text documents. The vast majority adopt a well-known standard template (e.g., 4+1[67], togap¹, arc42²). Textual documentation is always complemented with diagrams based on a common shared visual language (e.g., UML, BPML, BPEL, flowchart, informal notation).

Documentation Audience

Documentation is written to satisfy the needs of three main stakeholders: customers, architects and developers.

For customers: documentation is written to meet contractual requirements. In this case documentation is often seen as a burden for the architect and provides limited support to practitioners working on the project. It provides a non-technical specification that can be used to prove compliance to agreed requirements during a post-development validation phase (**G**).

¹<http://www.opengroup.org/togaf>

²<http://www.arc42.de>

For architects: documentation is written to maintain a general overview of the system and support high-level design reasoning. Some respondents believe that developers are not interested in reading about architecture. “Developers only care about functionality and tend to ignore non-functional properties” (E). This assumption supports the idea that architecture and implementation are on different levels of abstraction and are hard to link together. Low effort is usually dedicated to keep documentation aligned and up-to-date with changes originated in the implementation. I stated that he rarely got any sort of feedback from the assumed recipients of his documentation work.

For developers: documentation is a blueprint, providing a high-level description of the system to technical users involved in the development process. It is particularly useful as an initial entry-point for new developers learning about the system. D said that “new developers start by reading the documentation, look into the code and finally sort out remaining doubts by talking with colleagues”. Documentation is used to transfer knowledge, is open for change and needs to be kept up-to-date.

Documentation Intent

In our study we identified two type of documentation styles: descriptive and prescriptive.

Descriptive Documentation: is meant to provide sufficient evidence to support developers in decision making activities. It is not written to set precise guidelines and rules but to help developers in evaluating alternatives and make good design choices. Architects writing “descriptive documentation” are usually skeptical about enforcing design rules through documentation. D said that “documented rules are often perceived as pedantic and restrictive”. He added that “forcing developers to learn them beforehand is a failing strategy and often leads to poor results” because “they could be ignored and neglected”. Apparently a much better approach is to provide useful feedback to developers when they break such rules.

Prescriptive Documentation: is more oriented towards the definition of strict guidelines and rules. The goal is to limit developers in their design choices in order to guarantee high-level properties (*e.g.*, maintainability). In this case, it’s often convenient to express architectural constraints in a clear and objective way. Most of the documents collected during our studies contained coding guidelines (general practices and syntax format rules) and design constraints regarding data values and event handling.

Formalization of Architectural Constraints

architectural constraints are rarely described formally. Formal specification is only used in practice to support specific verification tools. In this case, users are forced

to extract architectural rules from the specification document and encode them in a separate file using a tool-specific notation.

In rare cases, companies develop their proprietary description language. N worked in a company where all developed applications are documented as visual diagrams based on a proprietary meta-model. Their models include a hierarchically organized set of interlinked logical components. All types of entities are characterized by various properties (*e.g.*, interface structure for components; message format, protocol, integration type for communication links). Each system, consisting of a set of components, is mapped to the specific infrastructural entity on which it is supposed to be deployed. This last information is used to feed a semi-automatic process for verifying the actual deployment configuration. N said that the documentation model adopted in his company is very helpful for keeping information consistent, accurate and closed to interpretation.

In other cases, users face the lack of usability of current specification mechanisms. D, for example, decided to verify package dependencies using a specific testing framework (JDepend). Unfortunately the test specification required by the adopted tool was not readable enough to be included in the official documentation. To solve this problem, he decided to specify the requirements in a spreadsheet and build a parser to generate a corresponding set of tests. In this case, having a simplified and testable representation of architectural rules justified the cost for building a conversion tool.

3.2.3 Validating Architectural Constraints

We observed that architectural constraints are validated using various approaches.

Manual Validation

According to the answers collected during our study, one way of validating architectural constraints is simply by running the system and manually checking some operational properties (*e.g.*, *Response time*, *Authentication*). This validation strategy is usually preferred when automated testing tools are not available or exist but are too expensive to buy or customize. Scalability is sometimes verified by generating a large number of requests using a script and evaluating responsiveness by interacting with the application through an additional session. Properties that manifest themselves in source code (*e.g.*, *Code conventions*), are often checked through code reviews. As mentioned by L, “the number of existing [testing] tools is far from being exhaustive”. He said that “companies rarely see the value of investing time in researching new testing techniques”. In many cases manual validation seems to be the most viable and frequently chosen alternative.

No Validation

Some respondents avoid the need for direct verification by relying on a framework or code generator. If the framework is not developed internally, the fact that certain architectural constraints are actually fulfilled is based on trust. **J**, responsible for the development of an internal framework used across multiple company projects, said that “frameworks should not be invasive but support the developer by simplifying his tasks and reducing possible design decisions in a non-invasive way”. Frameworks that are built to limit implementation choices, as confirmed by **M**, are not well perceived by developers. A framework should convince developers to use its functions by offering useful services that contribute to reducing the cost of development (**J**). Code generators are typically used to simplify the maintenance and creation of modules that depend on business needs that vary through time. Our interviewees agreed on the fact that building testing tools is usually not an economically viable option. Building testing tools is also seen as a challenging task requiring advanced programming skills.

Automated Validation

When possible, architects prefer to use automated techniques. This can be done by writing programmatic tests or relying on tools developed by a third party. Existing tools do not always fit the needs of our respondents. Multiple respondents said that some of the currently available tools were lacking in flexibility and usability. **F** worked on a project where components could be identified by looking at the suffix of class names. All the tools he tried supported package name matching as the only mapping strategy. **K** was working on a system based on the OSGi framework³. He was not aware of any tool that allowed him to automatically check whether the specified dependencies existing between the OSGi bundles composing his system were actually consistent with the architectural specification. The only way to verify the alignment between implementation and specification was to manually inspect large XML configuration files.

Most of the tools force users to operate on an overly technical level. This fact prevents non-technical stakeholders from accessing valuable information and introduces new costs for setting up and maintaining architectural tests. Current testing solutions require the user to specify testing rules in separate files. Quality constraints must be specified twice: in the official documentation using natural language (for supporting communication and reasoning) and in a purpose-built formal specification file (for supporting a specific testing solution). The resulting fragmentation leads to increased costs for maintaining multiple specifications aligned and consistent.

³<http://www.osgi.org/>

3.3 Corroborating the Evidence: a Quantitative Study

To confirm the validity of our taxonomy on a larger scale, we developed a second study. This study was aimed at obtaining a more uniform overview on how quality attributes (identified in the first study and presented in Table 3.2) are considered by practitioners.

We now report some of the main observations resulting from the analysis of the obtained results.

O1. Most requirements are not formally specified: Our survey confirms that very few requirements are formally or semi-formally specified (Table 3.2). In fact, only 2 architectural constraints (*Signature, Dependencies*) out of 22 are formally specified more than 50% of the time. *Signature* architectural constraints are specified using UML with custom profiles, XSD and IDLs (OMG IDL, MIDL, WSDL). *Dependencies* are described using tool-specific notations (e.g., JDepend, ndepend, macker, DCL, SOUL), Java annotations and UML with custom profiles. Others (*Data structure, Naming conventions*) are also quite frequently formalized. *Naming conventions* can be specified using regular expressions, EBNF grammars, tool-specific notations (e.g., SOUL for IntensiVE) or Java (e.g., plugins for Checkstyle and PMD). *Data structure* architectural constraints are either specified using standard schema definition languages (DTD, XSD) or semi-formal modeling notations (ER, UML).

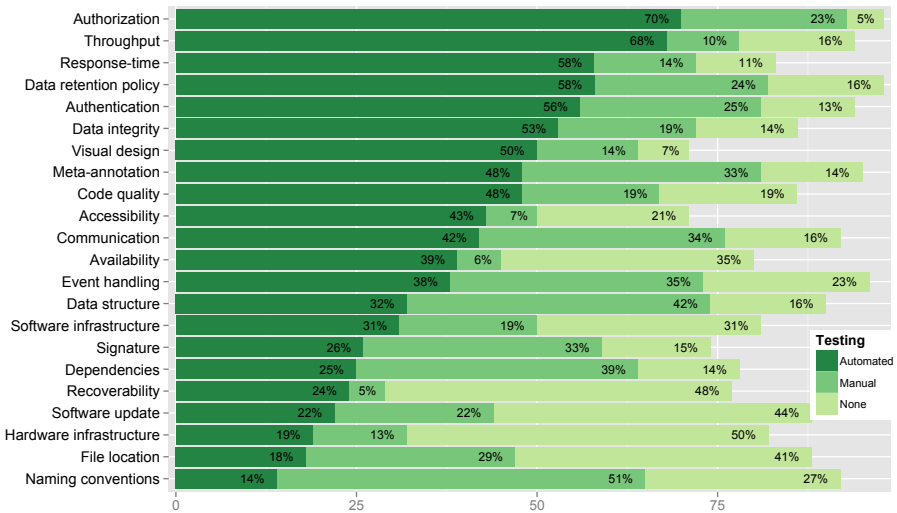


Figure 3.1: Survey results: various approaches for validating architectural constraints.

Constrained QA	Tool	Reported Testing Tools
authorization	15%	SoapUI / <i>other</i> : Framework (JAAS)
throughput	26%	Meter, LISA, Selenium, Lucust, Gatling, HP LoadRunner
response-time	17%	JMeter, LISA, Selenium
data retention policy	8%	<i>no tool specified</i>
authentication	3%	<i>other</i> : Framework (JAAS, Spring)
data integrity	8%	Moose / <i>other</i> : db-constraints, Framework
visual design	4%	<i>other</i> : Framework
code quality	39%	Sonar, Findbugs, Code critics, Checkstyle, Emma, Clover
meta-annotation	19%	dclcheck
accessibility	0%	<i>no tool specified</i>
communication	8%	Moose, dclcheck
availability	10%	DynaTrace, Gomez, Shell script + Selenium, Pingdom
event handling	12%	dclcheck, Moose
data structure	16%	Moose / <i>other</i> : Custom tools
software infrastr.	8%	<i>other</i> : Automated declarative provisioning
signature	7%	Moose, JMeter, soapUI
dependencies	22%	SAVE, dclcheck, Patternity, Jdepend, Ndepend, Macker, Intensive, SmallLint, DSM tool
recoverability	0%	<i>no tool specified</i>
software update	0%	<i>no tool specified</i>
hardware infrastr.	6%	<i>no tool specified</i>
file location	0%	<i>other</i> : Guaranteed by framework
naming conventions	11%	Code critics, Checkstyle, PMD, FxCop, Intensive, Petit-Parser

Table 3.3: Survey results related to tool-aided architectural constraints testing. Columns (from left to right): Constrained quality attribute; respondents using third-party tools for testing the constraint; adopted tools.

O2. *Automated conformance checking is not commonplace*: Results show that the use of automated techniques (*i.e.*, using white-/black-box testing or tools) for validating architectural constraints is not commonplace (Figure 3.1). On average, 59% of the surveyed population adopts non-automated techniques (*e.g.*, code review or manual validation) or avoids validation completely. Based on the results of our survey (Figure 3.1), the following architectural constraints are mostly validated manually: *Dependencies* (10 users), *Visual design* (8), *Naming conventions* (7), *Communication* (5). architectural constraints that remain most often unvalidated are: *hardware infrastructure* (50% of respondents), *recoverability* (48%) and *software update* (44%). Automated validation is not commonplace and is mostly adopted to validate architectural constraints regarding end-user properties (*e.g.*, *Response time*, *Throughput*) and security (*e.g.*, *Authorization*, *Authentication*, *Data retention policy*). Table 3.3 shows which tools are used by the participants of our survey to validate constraints related to the identified quality attributes.

O3. Tool support for automated validation is insufficient: One of the reasons why automated validation is not widespread seems to be related to the scarce availability of industrial-strength tools matching some practitioner’s needs. A number of quality attributes (e.g., *Code dependencies*, *Naming conventions*) can be checked with a large number of tools, while others (e.g., *Data integrity*, *Meta-annotations*), considered as equally important, can only rely on a much smaller range of solutions.

O4. Users’ needs are still not completely recognized: Figure 3.1 shows that several requirements are also more frequently validated manually than automatically. The most striking examples are *Data structure*, *signature*, *dependencies*. This suggests the possibility that some requirements are still left unaddressed and need to be investigated further by conducting on-the-field studies. We believe that further analysis of emerging requirements could lead to new opportunities for future research in the field of tool development and tool building support.

O5. Emphasis is given to secondary requirements: Another interesting observation is that quality attributes that were frequently encountered in the past (e.g., *Software update*, *accessibility*) generally did not have a significant impact on the outcome of related projects (See “familiarity” and “importance” columns in Table 3.2). Further studies should analyze current design and specification methodologies and propose improvements on existing documentation practices.

O6. Tools do not take advantage of existing formalizations: Figure 3.1 shows that some constraints (e.g., *dependencies*, *naming conventions*) are more often formally specified than automatically validated. However, formally specifying constraints without automatically verifying them is less than optimal. Based on our analysis, we observe that some adopted notations do not provide sufficient details to support validation (e.g. UML for describing *signature*) and other notations are not fully taken advantage of by the existing tools (e.g. regular expressions for describing *naming conventions*). We think that more empirical studies are needed in order to expose actual formalization practices. The results of these studies might expose common flaws of existing notations and provide concrete evidence of practitioner’s needs.

3.4 Discussion

In this section we discuss some general strategies that could help address the issues raised in the previous section.

Reduce the Gap between Specification and Implementation

As observed, many of the current tools force the user into a needlessly technical exercise. Several dependency testing tools (*e.g.*, JDepend, Dependometer), for example, not only require the test specification to be written using a technical notation (*i.e.*, Java or XML), but also offer poor documentation on how to do so.

Architects should be able to express their concerns in a single uniform format. Respondent **G** said that having the option to embed a formal (yet readable) test specification of his architectural rules in a Word document would be extremely appealing to him. This would allow him to write well-formed testing rules in a familiar environment with the additional benefit of automatic validation.

Terra *et al.* [115] and Marinescu *et al.* [80] proposed two different DSLs (Domain Specific Languages) for expressing architectural constraints (See section 3.6). Both languages serve the purpose of encoding valuable information in a testable yet readable format. Unfortunately the expressiveness of such DSLs is strongly defined by the capabilities of the underlying tool. Völter [119] reports on a case study where a DSL is defined progressively by interacting with the customer. The language, grammar and support tooling is developed iteratively and will eventually be used as the basis for code generation and analysis. Cucumber⁴, a behavior-driven development framework, is based on a similar concept. Tests are written by non-technical stakeholders and are checked by building an interpreter that translates the text into actual unit tests.

These approaches show that having business-readable descriptions of relevant design properties helps keep alive the conversation between all involved stakeholders. It also shows that a well-engineered DSL is useful for encoding information in a uniform and unambiguous manner, which can turn useful for supporting more sophisticated testing activities. We believe that users should not be asked to describe their architectural constraints within the boundaries defined by a testing tool. Instead, tools should be employed to verify user-defined rules on a best effort basis.

Increase Awareness through Continuous Feedback

Several respondents (**G**, **H**, **J**) use Sonarqube as a guide for driving code review activities. Sonarqube aggregates code analysis reports from multiple sources and presents them in a customizable web-based interface. Information is constantly kept up-to-date, well integrated and easy to navigate. All aspects exposed by the tool relate to general low-level characteristics of the system that are typically of little interest for architects. The strength of Sonarqube mostly seems to be bound to its integrability (analysis can be configured to run as a build step in a wide range of continuous integration servers), the concreteness of its result and the fact that all information are current and kept up-to-date.

⁴<http://cukes.info>

Having seamless access to a comprehensive set of system-wide properties and infringed rules is a good way to exercise control over non-functional aspects of an implementation. If architects had the chance to define domain-specific rules for testing design constraints that are relevant for their architecture, they would be able to reach a higher and more targeted level of control. Our intuition is that monitoring platforms, such as Sonarqube would largely benefit from being integrated with highly customizable DSL-based tools (*e.g.*, DCL [115], InCode.Rules [80]). Being able to specify similar and more articulated rules on this and other aspects of the system would eventually reduce the generality of the results minimizing the number of false warnings and optimizing review-time.

3.5 Threats to Validity

Internal Validity

During our first study, we tried to gather impressions and opinions by conducting semi-structured interviews. Our goal was to gather a clear answer to all the research questions presented in the introduction. All discussions have therefore been partially moderated by the interviewer. We did our best to minimize the influence of the interviewer on the respondent, but we cannot exclude the existence of biased answers. Some observations or questions made by the interviewer might have induced the respondent to articulate his answer in an unnatural way. The effect of a similar threat should have been mitigated by the number of different answers to the same question.

Users taking part in the survey had the right to remain anonymous. 41% of them chose not to share any identifying personal information (*i.e.*, email address). Among those, 71% (29% of the total population) did not specify their professional title. Due to this lack of information, we are unable to make general statements over the population participating to the survey. It would anyway be reasonable to assume that most of the people were either architects or professionals playing a comparable role. The fact that we contacted people belonging to our industrial collaborators network and that we posted invitations only on architecture-oriented virtual communities should support our hypothesis.

External Validity

Another limitation could be seen in the relatively modest number of participants who participated in each phase of the study. The first study involved 14 respondents, while the survey counted 34 valid results. These numbers could appear small, but in fact are comparable to those reported by similar studies. Four out of five of all the interview-based studies centered around non-functional requirements [5] involve 14 or fewer participants. If we consider the surveys related to the same topic

[5], we see that two out of four studies draw their conclusions based on fewer than 34 responses.

3.6 Related Work

In our work we discuss the nature of architectural constraints and report on the techniques used for their verification. We examine both topics from a very pragmatic point of view, taking into consideration concrete examples and specific information. To the best of our knowledge, no other empirical study covers the same topics adopting a similar standpoint.

Several surveys related to NFRs (non-functional requirements) have been carried out (See related work by Ameller *et al.* [5]). The main outcome of all these studies often consists of a ranking showing how non-functional requirements compare based on the level of importance attributed by the users. All these studies focus on generic quality characteristics ignoring actual quality attributes that practitioners address in the requirements. Our study provides new insights from a complementary point of view, showing which quality attributes are considered relevant and providing details of their validation.

Poort *et al.* [105] found a statistical correlation between the verification of NFRs and project success. According to their results, the benefits of verification are also more significant if NFRs are verified in early stages of a project. In our study we explore how NFRs get actually validated in practice.

Various research contributions show that architecture-related requirements can be formalized using ADLs (architectural description languages). ADLs allow one to model an architecture as a set of interlinked components enriched with pre-defined meta-annotations. These models are typically weakly related with the implementation. Tools are sometimes provided for checking the semantic consistency of relationships and annotations but only at the model level. Moreover, there is scarce evidence that the general concepts defined in ADLs (*i.e.*, Components, Ports, *etc.*) actually reflect the way architects think about their architecture. Case studies, showing evidence of the practical utility of the language, can only be found for a few of the most prominent ADLs (*i.e.*, AADL [35, 34] and xADL [14]). We think that the lack of support for testing concrete architectures combined with the possible mismatch between offered features and real needs can be the cause of the — by now confirmed [79] — failure of adoption of ADLs by the general public. In this chapter we draw observations that could help make ADLs more effective and useful.

Recent research efforts try to make up for these limitations by proposing more test-oriented ADLs. Terra *et al.* [115] proposed a specification language for expressing restrictions on the existence of certain types of relationships (*e.g.*, access, extension) between sets of classes. Marinescu *et al.* [80] supports the specification of undesired

dependencies and class-level anti-patterns. Both ADLs are supported by custom-built testing tools that enable rule verification at the code level. Other languages (*i.e.*, SOUL [92] and LePUS3/Class-Z [44]) are more formal and support more complex specifications. They provide the means to validate architectural constraints at code level, but also require considerable training before usage.

3.7 Conclusion

We presented the results of two empirical studies that explore how architectural constraints are defined and validated in practice. The studies show that architects care about the validation of architectural constraint but are often unable to make best use of the currently available tools.

We observe that the present offering of tools is limited in number and that several solutions are not able to satisfy common requirements (see section 3.4). Practitioners are rarely willing to develop solutions for governing architectural decay and are not motivated to formalize their architectural constraints. Current formalization notations are typically strongly tied to specific testing solutions and are often lacking in readability. To improve this situation, we propose some ideas for specifying architectural constraints and for reducing the cost of validation. Future testing solutions should take advantage of existing formalizations and provide functionalities that fulfill empirically recognized requirements.

In the future we plan to apply some of the discussed ideas by experimenting with new solutions for supporting the specification and validation of architectural constraints.

4

A Unified Approach to Conformance Checking

Software erosion can be controlled by periodically checking for consistency between the *de facto* architecture and its theoretical counterpart.

In the previous chapter we observed that this process is typically set in place but often not automated. Developers still heavily rely on manual reviews, despite the availability of dedicated tools [38] (See chapter 3).

This is partially due to the high cost involved in setting up and maintaining tool-specific validation tests. Tools often fail to fulfill the needs of practitioners and are hard to adapt to actual requirements. Most solutions are specialized for a single domain and are based on a unique set of technical and conceptual assumptions.

To reduce this cost, we propose a novel approach that unifies the functionality provided by existing tools under the umbrella of a common business-readable DSL. By using a declarative language, we are able to write tool-agnostic rules that are simple enough to be understood by untrained stakeholders and, at the same time, can be interpreted as a rigorous specification for checking architecture conformance.

4.1 Motivation

Architectural rules and constraints are essential to the formulation of architectural specifications [59]. The compliance of a system to architectural constraints can be monitored over time using various techniques (*e.g.*, Reflexion models [97]) [26].

Unfortunately, the tools that implement these techniques are at best used to provide basic support information during manual tasks. Studies show that developers still heavily rely on manual reviews as a means to check architectural conformance. In fact, static analyzers are used in only 33% of the cases [38]. The use of manual techniques does not scale and entails additional costs that could be minimized by automating parts of the process and using existing solutions and technologies. These observations are consistent with the results we obtained in our previous study in which we showed that, on average, 59% of software architects adopt non-automated

techniques (e.g., code review or manual validation) or avoid validation completely (See chapter 3).

To understand the lack of adoption of automated techniques, we investigated ourselves several tools. The first observation was that many tools provide insufficient documentation material. We ran a small experiment in which we analyzed how 4 tools compare on the evaluation of several dependency constraints: *JDepend*¹, *Macker*², *Dependometer*³ and *Classycle*⁴. The first tool, *JDepend* kept failing without reporting the cause of the error. Only later we realized that the tool required the user to explicitly list all packages contained in the analyzed system. This unintuitive requirement was not documented and required a considerable amount of time to be deduced. *Dependometer* also failed in delivering satisfying guidelines to set up our experiment. In fact, the documentation artifact that helped us the most in understanding how properly configure the tool was a loosely commented XML configuration template published on the project’s website. *Macker* could not even be set up. The documentation provided was not sufficient to cover our use case.

The reluctance of using available tools might be thus partially related to the general problem of insufficient availability of industry strength solutions and to the steep adoption curve that every individual tool presents. Practitioners have often a very personal view on how architecture should be specified and are not aware of any best practice which could support them in testing the conformance of a system. Where conformance/analysis tools exist, users typically face several other obstacles that hinder adoption. In the remainder of this section we identify three main obstacles to the adoption of architectural monitoring tools. To match these obstacles we propose corresponding requirements that, if fulfilled, can help mitigate them.

4.1.1 Scattered Functionality

Most existing tools are specialized on a narrow domain and are typically capable of evaluating only a small number of constraint types. Pruijt *et al.* [106] compare several tools for checking architecture conformance and conclude that “not one of the tested tools is able to support all the [...] rule types included in our classification”.

In our previous study (see chapter 3), we show that tools used in industry are capable of handling at most 3 out of the 22 quality requirements typically specified by practitioners. If one, for example, had to check whether a given architecture correctly fulfills a certain set of structural invariants (e.g., dependencies, meta-annotations, signatures) and meets predefined performance objectives (e.g., latency, throughput), she would need to choose at least two different tools. In a real architectural specification one would need to check many more constraints.

¹<http://clarkware.com/software/JDepend.html>

²<https://innig.net/macker/>

³<http://source.valtech.com/display/dpm/Dependometer>

⁴<http://classycle.sourceforge.net>

To reduce fragmentation and increase the operability of existing tools we suggest to:
⇒ **Req. 1:** *Consolidate the functionality offered by existing tools under a single coherent interface.*

4.1.2 Specification Language Heterogeneity

Current tools are based on different specification languages that differ in both syntax and semantics. In our experiment (introduced at the beginning of the section) we encountered three different types of specification formats: XML, Java, textual DSL.

The fact that tools operate independently, also increases the incidence of duplicated information across specifications. To evaluate three dependency rules across all tools, we had to write four specifications in four different languages. Common configuration parameters (*e.g.*, source code path, analyzed package filters, etc.) had to be replicated multiple times.

Language heterogeneity appears to be a problem also when dealing with the output of the analysis. Results are encoded in arbitrarily defined formats (*e.g.*, XML, CSV). The activity of merging the results of various tools into a single report is a time-consuming and sometimes hard to automate task.

To mitigate the costs that stem from language heterogeneity we suggest to:
⇒ **Req. 2:** *Decouple the specification from the various individual syntaxes.*

4.1.3 Specification Language Understandability

The language used to specify architectural rules is of essence. In some cases stakeholders invest effort into hiding the details of the specification language imposed by the tool and in others, they jump through hoops to adapt to an inflexible language.

In our previous study we encountered an architect who specified the dependencies allowed in his system as a dependency structure matrix inside a spreadsheet (see chapter 3). To test these dependencies, he implemented a custom generator that parsed the spreadsheet and produced an executable JDepend test suite. For this user, having a simplified and testable representation of his architectural rules justified the cost for building a (functionally unnecessary) custom conversion tool.

In their experiment, Pruijt *et al.* note that some rules need to be specified through workarounds (*e.g.*, “X is only allowed to use Y” was expressed as a combination of “X cannot use anything” and “X can use Y”)[106]. When that happens, the viewpoint of the user has to adapt to the conceptual model imposed by the tool. This forms a threat to maintainability since, once specified, rules cannot easily be traced back to the originating concern that they are expressing [106].

Finally, the success of several test-oriented requirement formalization solutions, such as FitNesse⁵ and Cucumber⁶ demonstrates the need for managing business and architectural rules through a readable and accessible interface. These solutions clearly separate the definition of a rule from the mechanisms used to test it. This enables less-technical stakeholders to contribute to the definition of requirements that are at the same time readable and testable.

To improve specification understandability we argue that:

⇒ **Req. 3:** *Rules should be designed using a specification language that reflects current practices.*

4.2 Our approach in a Nutshell

We propose a novel approach to architecture conformance checking that addresses the limitations identified in section 4.1. Our solution aims at utilizing the functionality offered by existing tools to test architectural rules specified using a single coherent specification language. This goal is achieved by integrating tools through custom-developed adaptors and transforming user-defined rules into easily verifiable boolean predicates. In our approach, we decouple the specification, as formulated by the user, from the conceptual and operational idiosyncrasies characterizing the tools used to evaluate it.

Our approach consists of:

- **Dictō:** A DSL for the specification of architectural rules. The language aims at supporting software architects in formalizing and testing prescriptive assertions on functional and non-functional aspects of a software system.
- **Probō:** A tool coordination framework that verifies rules written with Dicto using third-party tools. Supported tools and analyzers are managed through custom crafted adapters.

With our approach, a software architect could define a new rule by writing the statement highlighted in Figure 4.1 (line 5).

This rule states that a certain number of *subject entities* (i.e., Test, View) must fulfill a given *constraint* (i.e., depend on) with respect to a certain number of *objects* (i.e., Model, Controller). Different *modifiers* (e.g., must, cannot, only .. can) can be used to change the semantics of a given rule. In our example, we require that all subject entities *can only* depend on the object entities.

The remaining statements shown in our example (lines 1-4 in Figure 4.1), are used to define the mapping between the *symbolic entities* used in rules and the corresponding concrete entities present in the system. Symbolic entities have a *type* (e.g., Package,

⁵<http://www.fitnessse.org>

⁶<http://cukes.info>

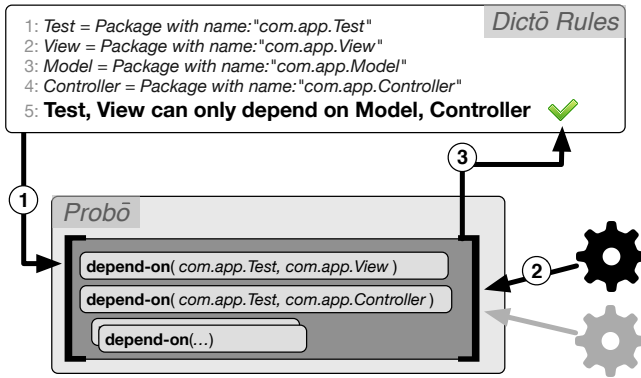


Figure 4.1: An overview of our approach: (1) rule normalization and predicate definition (2) predicate evaluation (3) result presentation.

Class, Website) and are described by *properties*. A symbolic entity may be mapped to multiple concrete entities by using regular expressions as property values (e.g., `Test = Package with name:"*.Test"`) or specifying properties which are common to multiple concrete entities (e.g., `Test = Package with parentPackage:"org.test"`).

Rules written in Dictō can automatically be validated by Probō (Figure 4.1-1). The proposed tool suite is designed to evaluate constraints related to structural and behavioral properties which can be checked automatically at any point in time (this excludes properties which cannot be directly measured by inspecting or executing intermediate development artifacts; e.g., usability, resource consumption). The evaluator normalizes each rule into a conjunction of smaller and more manageable sub-rules. Normalized rules are used to generate predicates, which are evaluated through third-party tools. In our example, the considered rule can be evaluated by assessing the truth-value of the following predicates⁷:

```

1 depend-on(com.app.Test, com.app.View)
2 depend-on(com.app.Test, com.app.Controller)
3 depend-on(com.app.Test, com.app.Test)
4 depend-on(com.app.Test, com.app.Util)
5 depend-on(com.app.Test, com.app.Model)

```

Predicates are evaluated by external tools through custom implemented adapters (Figure 4.1-2). Adapters are assigned to predicates according to a set of pre-defined syntactic matching criteria specified in the adaptor class. They are responsible of generating a test specification that, once executed, produces sufficient information to evaluate the predicates they are assigned to.

⁷In this chapter, predicates are represented using a Prolog-like notation.

In our example, we evaluate the obtained predicates using Moose⁸, a software analysis platform that can be used to explore structural characteristics of an object-oriented system using user-defined queries. Moose can be executed from the command line and queries can be specified in a script passed as argument. To test our rule (line 5 in Figure 4.1), we generate a set of queries that check whether each pair of packages indicated in the intercepted predicates are actually dependent on each other. The results are fed back to Probō and used to compose an aggregated report that lists all the rules violated in the analyzed system (Figure 4.1-3).

4.3 Formal Description

We here formalize the syntax of the Dictō language and describe how rules get evaluated in Probō.

4.3.1 Dictō Syntax

Dictō is designed to resemble the form and structure of industrial specifications. In a previous study (see chapter 3), we collected several documentation artifacts used in real projects. Some contained developer guidelines while other described higher level design decisions and constraints. We focused on identifying statements that could be categorized as *rules*. Hereafter we report selected sentences (translated and adapted from German) encountered during the process :

1. MoneyAmount **must** be annotated with @org.hibernate.annotations.Columns [...].
2. The execution time of validateCombination() **must** be below 10 ms.
3. The (XML) Text Element **must** contain a Font Element with the following attributes: size, style, [...].
4. ApplicationExceptions **cannot** automatically trigger a rollback when the exception is thrown by a method belonging to a BusinessService.
5. Models [...] **cannot** invoke operations from Business Services.
6. Throwable **cannot** be caught (**only** its subclasses).
7. [...] Domain.jar [...] can be accessed by Webservice and Admin GUI. Write operations **can only** be accessed by Admin GUI.

⁸<http://www.moosetechnology.org>

Based on this limited sample, we observe that rules are essentially predicates related to a variable number of subjects through the use of modal verbs (e.g., must, can). One of the documents that we analyzed clearly defines the modal verbs used in the artifact. The list includes: must, cannot, should, should not, can. Since Dictō was designed to support conformance checking, we chose to support *must* and *cannot* (with its variations *only can* and *can only*).

All the rules reported above can be converted in Dictō statements as follows:

```

1 MoneyAmount must be annotated with "@[.]Columns"
2 ValidateCombination must be executed in < 10 ms
3 TextElement must contain FontElementWithAttributes
4 AppExThrownByBS cannot invoke Rollback
5 Models cannot invoke BSMETHODS
6 only ThrowableSubclasses can be caught
7 Domain can only be accessed by AdminGUI

```

These statements can be evaluated by third-party tools through purpose built adapters. Each rule contains references to symbolic entities (capitalized) which are declaratively mapped to concrete entities existing in the project (e.g., package, class). The first rule, for example, contains a reference to *MoneyAmount* which could be declared as follows:

```

1 MoneyAmount = Class with name:"*.MoneyAmount"

```

A symbolic entity is characterized by a set of properties. Properties can be seen as a complement to the information encoded in a rule. If, for example, we consider rule number 3, we see that the rule not only requires that the (XML) element *Text* contains *Font*, but also specifies that *Font* must have a certain number of attributes. Instead of specifying these two conditions as two separate Dictō rules (which is also possible), we can choose to write a single rule (as specified in the box above) and describe the *Font* entity as follows:

```

1 FontElementWithAttributes = XMLElement with name:"Font-Element", attribute:"size", attribute:"style",
...

```

All the statements presented in this section are syntactically consistent with the specification in Figure 4.2.

4.3.2 Meta-Model

Dictō specifications are evaluated using Probō. Probō transforms a parsed specification into a model that conforms to the meta-model illustrated in Figure 4.3.

In our meta-model, \mathbb{R} is the set of user-defined rules, \mathbb{S} are user-defined symbolic entities, and \mathbb{C} are concrete software entities existing in the analyzed system. \mathbb{E} is

```

specification = (entity | rule)*
entity = symbol '=' type 'with' prop ':' val (' prop ':' val)*
rule = (rule-subj ('must' | 'cannot' | 'can only') rule-pred)
      | ('only' rule-subj 'can' rule-pred)
rule-subj = symbol (' symbol)*
rule-pred = predName (val | symbol) (' (val | symbol))*
prop = predName = symbol = type = String
val = StringLiteral | Integer

```

Figure 4.2: DSL syntax specification (EBNF)

the union of \mathbb{S} and \mathbb{C} . Π are the properties that describe both concrete and symbolic entities, and \mathbb{P} are the predicates into which a rule is converted. Concrete entities are automatically deduced from the source code by using a fact extractor. This tool statically analyzes the project and returns a list of concrete entities found in the source code. In our prototype implementation (section 4.4) we use VerveineJ⁹ as a Java fact extractor. Elements belonging to the previously described sets are differentiated through categorization elements defined on the right hand side of the model diagram (illustrated with grey background in Figure 4.3). These categories are pre-defined in Prob δ (based on the adapters supported or other types of configuration). The categories are defined as the following sets: \mathbb{R}_m is the set of rule modes allowed for a rule, \mathbb{E}_t and Π_n are the sets of entity types and property names supported by the framework, and \mathbb{P}_n is the set of predicate names for which dedicated adapter support exists. Additionally we define \mathbb{V} , a set containing primitive values defined by the user (*i.e.*, Strings and Integers).

The rule modes currently supported in our solution are the following: *must*, *cannot*, *can-only*, *only-can*. Entity types and property names are defined based on the information produced by the fact extractor. Our current fact extractor is able to detect, for example, packages which are described by various properties (*e.g.*, name, is empty) and relationships (*e.g.*, parent package, contained classes). Since we have this information, we can decide to support entities of type “package” associated with properties named “name” and “parentPackage”. Predicate names are defined by the adapters installed in Prob δ . If an adapter declares that it is capable of handling rules containing “depend on”, it means that we can provide support for rules with predicate name “depend on”. User-defined rules may include multiple predicates declared in different adapters.

4.3.3 Semantic domain

Entities belonging to previously described sets are related through various associations. Entities and properties support the associations described in Figure 4.4.

⁹<https://gforge.inria.fr/projects/verveinej/>

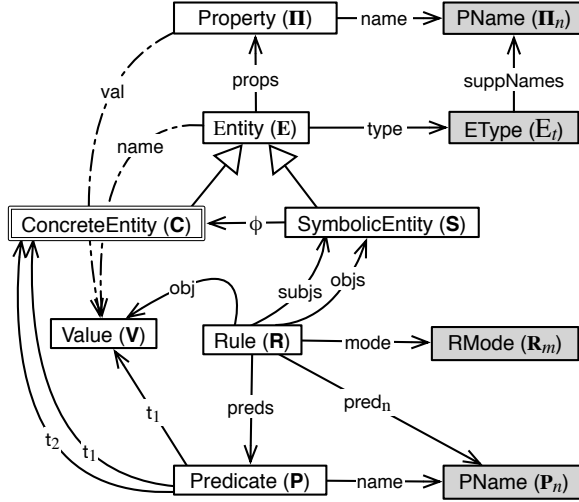


Figure 4.3: Semantic domain meta-model. White entities are defined by the user while grey ones depend on implementation choices taken in Prob \bar{o} .

$name : \mathbb{E} \rightarrow \text{String}$	entity name	(1)
$type : \mathbb{E} \rightarrow \mathbb{E}_t$	entity type	(2)
$value : \mathbb{I} \rightarrow \mathbb{V}$	property value	(3)
$name : \mathbb{I} \rightarrow \mathbb{I}_n$	property name	(4)
$suppNames : \mathbb{E}_t \rightarrow 2^{\mathbb{I}_n}$	supported property names	(5)
$props : \mathbb{E} \rightarrow 2^{\mathbb{I}}$	entity properties	(6)
$\pi \in props(e) \iff name(\pi) \in suppNames(type(e))$		

Figure 4.4: Domain functions for entities and properties. With 2^S (eq. 5 and 6) we denote the power set of S (i.e., $\mathcal{P}(S)$).

In our previous example (shown below), we can identify 4 symbolic entities (i.e., Test, View, Model, Controller).

```

1 Test = Package with name: "com.app.Test"
2 View = Package with name: "com.app.View"
3 Model = Package with name: "com.app.Model"
4 Controller = Package with name: "com.app.Controller"
5 Test, View can only depend on Model, Controller

```

The entity s described in the first line has $name(s) = "Test"$ and is of $type(s) = Package$. This entity is associated to a property π which has $name(\pi) = name$ and $val(\pi) = "com.app.Test"$. Since $name \in suppNames(Package)$, we can say that π is a valid attribute of s ($\pi \in props(s)$).

Entity properties are used to define a declarative mapping between symbolic and concrete entities. The mapping logic used in Prob δ is described in Figure 4.5.

$str_1 \sim str_2$	string matching predicate	(7)
str_1 matches regular expression in str_2		
$\pi_1 \approx \pi_2$	property compatibility predicate	(8)
$\pi_1 \approx \pi_2 \iff name(\pi_1) = name(\pi_2) \wedge val(\pi_1) \sim val(\pi_2)$		
$\phi: S \rightarrow C$	entity mapping function	(9)
$c \in \phi(s) \iff c \in \text{"concrete entities in target system"} \wedge$		
$(\forall \pi_s \in props(s))(\exists \pi_c \in props(c))(\pi_c \approx \pi_s)$		

Figure 4.5: Domain functions for entity mapping.

In our example, the entity s (named “Test”) is mapped to a concrete entity c (named “com.app.Test”) with compatible properties ($\pi' \in props(c)$, $\pi \in props(s)$ and $\pi' \approx \pi$). A property is compatible with another if the value of the first matches the regular expression defined as the value of the second.

$mode: \mathbb{R} \rightarrow \mathbb{R}_m$	rule mode	(10)
$subjs: \mathbb{R} \rightarrow 2^S$	rule subjects	(11)
$objs: \mathbb{R} \rightarrow 2^{S \cup V}$	rule objects	(12)
$pred_n: \mathbb{R} \rightarrow \mathbb{P}_n$	predicate name	(13)
$subs: \mathbb{R} \rightarrow 2^R$	sub-rules	(14)
$r' \in subs(r) \iff subjs(r') \subseteq subjs(r) \wedge objs(r') \subseteq objs(r)$		
$\wedge mode(r') = mode(r) \wedge pred_n(r') = pred_n(r)$		
$\mu: \mathbb{R} \rightarrow 2^R$	normalized rules	(15)
$r' \in \mu(r) \iff r' \in subs(r) \wedge$		
$ subjs(r') = objs(r') = 1,$ if $mode(r) = M/C$		
$\langle subjs(r') = 1 \wedge objs(r') = objs(r),$ if $mode(r) = CO$		
$subjs(r') = subjs(r) \wedge objs(r') = 1,$ if $mode(r) = OC$		

Figure 4.6: Domain functions for: Rule (\mathbb{R}). The following abbreviations have been used: M/C = must/cannot; CO = can-only; OC = only-can.

Rules are described by a mode, a set of subjects and optional objects (Figure 4.6). Our example contains a single rule r , with $mode(r) = can\text{-}only$. The rule has two subjects ($subjs(r) = \{Test, View\}$) and two objects ($objs(r) = \{Model, Controller\}$). The predicate name of the rule is $pred_n(r) = depend\text{-}on$. Rule subjects are the symbolic entities for which the rule needs to be evaluated. Subjects and objects will be used to form the predicates derived from the rule (Equation 19).

Since rules can vary in complexity (*i.e.*, the number of subject and object entities is not constrained), they need to be broken down into smaller more manageable

rules (called *normalized rules*). User-defined rules are equivalent to the conjunction of all the normalized rules derived from them. In our example, the normalized rules obtained from the original rule are the following:

```
1 Test can only depend on Model, Controller
2 View can only depend on Model, Controller
```

Normalized rules are obtained in different ways depending on the mode of the original rule (Equation 15). They share the common property of being a sub-rule of a common ancestor rule (Equation 14). This means that all share a subset of the subjects and objects associated to the rule they are derived from.

$name : \mathbb{P} \rightarrow \mathbb{P}_n$	predicate name	(16)
$t_1 : \mathbb{P} \rightarrow \mathbb{E}$	predicate term 1	(17)
$t_2 : \mathbb{P} \rightarrow (\mathbb{E} \cup \mathbb{V})$	predicate term 2	(18)
$preds : \mathbb{R} \rightarrow 2^{\mathbb{P}}$	rule predicates	(19)
$p \in preds(r) \iff name(p) = pred_n(r) \wedge$ $\exists s \in subs(r), o \in objs(r):$ $t_1(p) \in \phi(s) \wedge (t_2(p) \in \phi(o) \text{ or } t_2(p) \sim o),$ if $mode(r) = M/C$ $\langle t_1(p) \in \phi(s) \wedge (t_2(p) \notin \phi(o) \text{ or } t_2(p) \sim o),$ if $mode(r) = CO$ $t_2(p) \notin \phi(s) \wedge (t_2(p) \in \phi(o) \text{ or } t_2(p) \sim o),$ if $mode(r) = OC$		
$preds_{\cap} : \mathbb{R} \rightarrow \mathbb{P}$	common sub-rule predicates	(20)
$p \in preds_{\cap}(r) \iff \forall r' \in subs(r) : p \in preds(r')$		

Figure 4.7: Domain functions for: Predicate (\mathbb{P}). The following abbreviations have been used: M/C = must/cannot; CO = can-only; OC = only-can.

Normalized rules are eventually transformed into predicates (Figure 4.7). Predicates are generated to further simplify the evaluation process. In fact, adapters can safely accomplish their task ignoring the original rule defined by the user. Their logic simply has to cope with boolean predicates generated by Prob \ddot{o} . These predicates, if evaluated correctly, provide sufficient information to derive whether the original rule has been violated or not.

Let's consider a sub-rule derived from the first of previously mentioned normalized rule: *Test can only depend on Model*. If we assume that our system is made of 5 packages (View, Test, Model, Controller, Util), we obtain the following predicates:

```
1 depend-on(com.app.Test, com.app.View)
2 depend-on(com.app.Test, com.app.Controller)
3 depend-on(com.app.Test, com.app.Test)
4 depend-on(com.app.Test, com.app.Util)
5 depend-on(com.app.Test, com.app.Model)
```

Predicates contain up to two terms and have a name. The first term is a concrete entity to which one of the subjects of the normalized rule has been mapped (or not mapped) to. The second term may be either a concrete entity corresponding (or not corresponding) to an object of the same rule or a simple primitive value. The first predicate p in our example has a $name(p) = depend-on$ and two terms $t_1(p) = com.app.Test$ and $t_2(p) = com.app.View$. It was obtained by taking the subject (Test) and object (Model) of the given rule and deriving all permutations existing between the concrete entities corresponding to the first and the concrete entities not corresponding to the second. This process varies according to the rule mode.

The predicates in our example are defined to prove the existence of relationships between two given entities. The first predicate is true if *com.app.Test* depends on *com.app.View*, and false otherwise. If the second term is not an entity, it means that the evaluation implies the verification of a property (e.g., *have-latency(MyWebsite, 10ms)*, *contain-code-clones(MyPackage)*).

Equation 20 is an auxiliary function that is used during the evaluation of *can-only* and *only-can* rules (See equation 22). This function returns the intersection of predicates derived from the sub-rules of a given (normalized) rule. In our case, $preds_{\cap}(r)$, where $r = Test\ can\ only\ depend\ on\ Model, Controller$, equals to:

```

1 depend-on(com.app.Test, com.app.View)
2 depend-on(com.app.Test, com.app.Test)
3 depend-on(com.app.Test, com.app.Util)

```

Two predicates (*depend-on([..].Test, [..].Controller)* and *depend-on([..].Test, [..]. Model)*) are not included in the set, since they can only be generated from one of the sub-rules derived from r .

4.3.4 Semantic Interpretation

After describing the semantic domain of our model, we describe how user-defined statements (conforming to the schema in Figure 4.2) are transformed into domain objects and how rules get eventually evaluated. The semantic equations in Figure 4.8, are a complete abstract specification of the interpretation algorithm implemented in Probō.

The *spec* equation takes a full user specification as input and returns the evaluation results computed for every interpreted rule. Rules are defined in the *stmt* equation. *subj* and *obj* are used to evaluate entities and values declared in a rule and link them to entities defined beforehand by the user. We assume that all statements used to define symbolic entities precede rule declarations. Entities are similarly interpreted using the *stmt* and *prop* equation.

spec: $2^{\{\top, \perp\}}$	stmt: $\mathbb{R} \times \mathbb{S} \rightarrow \mathbb{R} \times \mathbb{S}$	
prop: Π	subj: \mathbb{S}	obj: $\mathbb{S} \cup \mathbb{V}$
<p>(a) $\text{spec}[\![\mathbb{S}]\!] = \{ \text{eval}(r), r \in \text{rules} \}$ where: $(\text{rules}, \text{entities}) = \text{stmt}[\![\mathbb{S}]\!](\emptyset, \emptyset)$</p> <p>(b) $\text{stmt}[\![S_1; S_n]\!](r, e) = \text{stmt}[\![S_n]\!](\text{stmt}[\![S_1]\!](r, e))$</p> <p>(c) $\text{stmt}[\![\text{NAME} : \text{TYPE} \text{ with PROPS}]\!](r, e) = (r, e')$ where $e' = e \cup s, s \in \mathbb{S}, \text{props}(s) = \text{prop}[\![\text{PROPS}]\!], \text{name}(s) = \text{NAME},$ $\tau(s) = \text{TYPE}$</p> <p>(d) $\text{stmt}[\![\text{only S can P O}]\!](r, e) = \text{stmt}[\![\text{S only-can P O}]\!](r, e)$</p> <p>(e) $\text{stmt}[\![\text{S can only P O}]\!](r, e) = \text{stmt}[\![\text{S can-only P O}]\!](r, e)$</p> <p>(f) $\text{stmt}[\![\text{S T P O}]\!](r, e) = (r', e)$ where $r' = r \cup \text{rule}, \text{rule} \in \mathbb{R}, \text{subj}(\text{rule}) = \text{subj}[\![\mathbb{S}]\!](e), \text{obj}(\text{rule}) = \text{obj}[\![\text{O}]\!](e),$ $\text{kind}(\text{rule}) = \text{T}, \text{predType}(\text{rule}) = \text{P}$</p> <p>(g) $\text{prop}[\![P_1, P_n]\!] = \text{prop}[\![P_1]\!] \cup \text{prop}[\![P_n]\!]$</p> <p>(h) $\text{prop}[\![\text{NAME} = \text{"VALUE"}]\!] = \pi$ where $\pi \in \Pi, \tau(\pi) = \text{NAME}, \text{value}(\pi) = \text{VALUE}$</p> <p>(i) $\text{subj}[\![S_1, S_n]\!](e) = \text{subj}[\![S_1]\!](e) \cup \text{subj}[\![S_n]\!](e)$</p> <p>(j) $\text{subj}[\![\mathbb{S}]\!](e) = s$ where: $s \in e, \text{name}(s) = \mathbb{S}$</p> <p>(k) $\text{obj}[\![O_1, O_n]\!](e) = \text{obj}[\![O_1]\!](e) \cup \text{obj}[\![O_n]\!](e)$</p> <p>(l) $\text{obj}[\![\text{"O"}]\!](e) = o$ where: $o \in \mathbb{V}, o \sim \text{O}$</p> <p>(m) $\text{obj}[\![\text{O}]\!](e) = o$ where: $o \in e, \text{name}(o) = \text{O}$</p>		

Figure 4.8: Semantic transformations.

The user-defined rule in our example (*Test, View can only depend on Model, Controller*) would define a rule object associated to two subject entities, two object entities and having a specific rule mode. A new rule model entity would be defined in function f (invoked by a through e), which invokes i and j to define its subjects and k and l to define its objects.

Once we obtain a full semantic model out of the initial user specification, we can evaluate all the rules by using the two functions defined in Figure 4.9.

The *eval* function iterates over all the rules obtained through the previously described interpretation process and evaluates them. The evaluation produces a positive outcome if none of the normalized rules derived from the given rule satisfies the condition prescribed for its rule mode. The condition is tested through a λ function, which will be executed using the best matching adapter capable of handling the given predicate.

In our example, none of the predicates in $\text{preds}_{\cap}(n)$ (where n is a normalized rule derived from the evaluated rule) are allowed to evaluate to true. We assume that the results for the predicates derived from our normalized rule *Test can only depend on Model, Controller* are evaluated as follows:

```

1 depend-on(com.app.Test, com.app.View) = false
2 depend-on(com.app.Test, com.app.Controller) = true

```

$$\lambda : \mathbb{P} \rightarrow \{\top, \perp\} \quad \text{predicate evaluation} \quad (21)$$

*evaluate predicate through best matching adapter
based on user-provided project configuration.*

$$\text{eval} : \mathbb{R} \rightarrow \{\top, \perp\} \quad \text{rule evaluation} \quad (22)$$

$$\text{eval}(r) = \top \iff \nexists n \in \mu(r) :$$

$$p \in \text{preds}(n) \wedge \lambda(p) = \perp, \quad \text{if } \text{mode}(r) = M$$

$$\langle p \in \text{preds}(n) \wedge \lambda(p) = \top, \quad \text{if } \text{mode}(r) = C$$

$$p \in \text{preds}_{\cap}(n) \wedge \lambda(p) = \top, \quad \text{if } \text{mode}(r) = CO/OC$$

Figure 4.9: Rule and predicate evaluation functions. The following abbreviations have been used: M/C = must/cannot; CO = can-only; OC = only-can.

```

3 depend-on(com.app.Test, com.app.Test) = false
4 depend-on(com.app.Test, com.app.Util) = true
5 depend-on(com.app.Test, com.app.Model) = true

```

Predicates presented in bold are common to all the sub-rules of the considered rule. Since the fourth rule belongs to all the sub-rules and evaluates to true, we can derive that the evaluated basic rule fails. The original user-defined rule (*Test, View can only depend on Model, Controller*) is the conjunction of its normalized sub-rules. Since one of them (here discussed) fails, Prob \bar{o} can conclude that the original rule is not correctly enforced in the target system.

4.4 Prototype Implementation

The approach, as described in section 4.2 and section 4.3, has been implemented in a proof-of-concept prototype (available on our website¹⁰). The prototype is implemented in Pharo Smalltalk¹¹, a modern Smalltalk dialect, and currently supports 7 types of conformance rules (Table 4.1).

While building this prototype we chose to implement adapters for tools commonly used by practitioners and belonging to different analysis domains. In its current implementation, the prototype supports rules related to maintainability (dependen-

¹⁰<http://scg.unibe.ch/dicto/>

¹¹<http://pharo.org>

¹²<http://www.moosetechnology.org>

¹³<http://pmd.sourceforge.net>

¹⁴See chapter 5

¹⁵<http://babelfish.arc.nasa.gov/trac/jpf>

¹⁶<http://jmeter.apache.org>

¹⁷<http://zn.stfx.eu/zn/index.html>

¹⁸<http://www.unix.com/man-page/A11/0/grep/>

¹⁹<https://github.com/sputnick-dev/saxon-lint>

Rule	Tool
Package [must, cannot, ..] depend on Package	Moose ¹²
Class [must, cannot, ..] invoke Method	Moose
Method [must, cannot, ..] be invoked	Moose
Package ₁ , .., Package _n [must, cannot, ..] be layered	Moose
System [must, cannot, ..] contain dead methods	Moose
Class [must, cannot, ..] have method "String"	Moose
Method [must, cannot, ..] be named "String"	Moose
Method [must, cannot, ..] have annotation "@String(..)"	Moose
Method [must, cannot, ..] have annotation name "String"	Moose
Method [must, cannot, ..] have annotation type "String"	Moose
Class [must, cannot, ..] implement interface "String"	Moose
Method [must, cannot, ..] have method parameter Class	Moose
Method [must, cannot, ..] throw Class	Moose
Method [must, cannot, ..] catch Class	Moose
Class [must, cannot, ..] be caught	Moose
Class [must, cannot, ..] have empty catch block	Moose
Class [must, cannot, ..] be thrown	Moose
System [must, cannot, ..] contain code clones	PMD ¹³
System [must, cannot, ..] contain cycles	Marea ¹⁴
Class [must, cannot, ..] lead to deadlock	JPF ¹⁵
WebResource [must, cannot, ..] have latency < Integer ms	JMeter ¹⁶
WebResource [must, cannot, ..] handle load from Integer users	JMeter
WebResource [must, cannot, ..] have content "String"	Zinc ¹⁷
WebResource [must, cannot, ..] have content type "String"	Zinc
WebResource [must, cannot, ..] have content length > Integer	Zinc
WebResource [must, cannot, ..] have status code Integer	Zinc
File [must, cannot, ..] contain text "String"	grep ¹⁸
File [must, cannot, ..] exist'	Probo
XMLTag [must, cannot, ..] have text "String"	saxon-lint ¹⁹
XMLTag [must, cannot, ..] have child "String"	saxon-lint
XMLTag [must, cannot, ..] have attribute "String"	saxon-lint
XMLTagAttribute [must, cannot, ..] have value "String"	saxon-lint
XMLTagAttribute [must, cannot, ..] have name "String"	saxon-lint

Table 4.1: rule types supported in Dictō. Each rule is checked through the tool listed on the right hand side of the table.

cies, code clones), performance (response time, throughput), compatibility (data structure) and reliability (deadlock-freeness, availability).

To define a new adapter, we followed these steps:

- Task definition: gather requirements for the adapter based on the properties that need to be tested.
- Tool selection: search for the best tool that fits the identified needs.
- Tool analysis: learn how to specify a valid test input in order to satisfy the identified needs.
- Adapter implementation: implement an adapter that is capable of checking

basic invariants derived from user-defined rules by interacting with the selected tool.

The effort required to implement an adapter for a well-understood tool is relatively modest. The average size of an adapter class is 64 lines of code. The size mostly varies depending on the verbosity of the input schema prescribed by the adapted tool.

Adapters are programmed to decide the truth value of a set of predicates that they agreed on handling. To better understand how this happens, let's consider the following predicate:

```
1 have-latency-less-than(http://www.xyz.com, "100 ms")
```

This predicate, in our current implementation of Probō, will be assigned to an adapter that relies on JMeter¹⁶. The adapter generates an XML file (88 lines of code) containing the specification of a JMeter test plan. The adaptor also defines a set of pre-specified commands that allow the execution of the generated test-case. The output resulting from the execution will be analyzed by the adapter through a specific function that decides whether a given predicate is actually verified or not. In this adapter, test results are traced back to the corresponding predicates using alphanumeric identifiers defined in our model.

Other adapters are implemented using similar approaches. Some (*e.g.*, the ones relying on JPF¹⁵ and PMD¹³) don't require the generation of an input specification since all configuration options are defined as command line parameters. Others (*e.g.*, the ones relying on UNIX command line utilities: ping^{??} and grep¹⁸) generate a UNIX shell script which is then invoked by during execution.

4.5 Discussion

Dictō limits the cost of conformance checking by fulfilling the requirements presented in section 4.1. We here discuss how our solution addresses the proposed requirements.

4.5.1 Scattered Functionality

We propose an integrated solution that employs the functionality of a variable number of tools to test a wide range of rules. The heterogeneity of the supported tools is hidden behind a single uniform coordination framework called Probō. Support for new tools is defined through adapters. Adapters are not built to directly expose the features offered by a given tool to the end user. They are rather designed to exploit

the functionality offered by a tool to obtain information that can be used to evaluate rule-derived predicates.

This approach allows us to decouple Dictō, the high level language used for rule specification, from the semantic and operational model associated to a specific tool. Adding an additional level of indirection between users and tools also implies that less control can be exercised on the configuration of the evaluation tool. Adapter developers can choose which kind of parameters should be exposed to designers (*e.g.*, in a load test performed with JMeter, the number of concurrent connections) and which, for the sake of simplicity and tool-independence, should be pre-defined by the adapter (*e.g.*, in the same test, “Use KeepAlive header” option defined in the generated test case). Adapter developers are also encouraged to build parametrized adapters, in which secondary configuration values that influence the outcome of the analysis can be adjusted based on the designer’s needs.

Hiding the specifics of the tools used for evaluation has the advantage of saving designers from discovering, learning and comparing tools. On the other hand, these tasks still need to be performed to build adapters. In our approach we hope to reduce the cost of these activities by curating an open repository of contributed tool adapters. By adopting this strategy we hope to grow a collection of reusable components that can be directly installed to support new functionalities. Users adopting a contributed adapter are not required to learn about the operational details related to the analysis tool used.

We plan to evaluate the possibility of sharing reusable adapters by running a case study in which we examine the actual overlap between user requirements. This study will be conducted by supporting various practitioners from different organizations on defining a comprehensive set of rules for an active project in which they are involved. Participants will partially be selected among the people involved in our previous empirical study (see chapter 3). As we proceed, we will gradually be able to assess whether the number of adapters needed to satisfy the user’s requirements grows or stabilizes over time.

4.5.2 Specification Language Heterogeneity

Dictō was designed to offer a single coherent specification language for expressing architectural rules to software architects. This language is independent from the specification mechanisms supported by existing tools. Tool-specific input is generated by adapters on the basis of a simplified model (consisting mainly of predicates) derived from a more complex originally defined set of rules. Tool-specific notations are indirectly supported through a well coordinated generative process partially managed by adapters. Rule designers are not required to have any knowledge regarding the tools that are employed to check their statements. This allows them to focus on the task at hand without being distracted by arbitrary implementation choices taken by tool providers. Similarly, adapter developers do not need to

cope with the full complexity of user-defined specifications. In fact, each rule defined through the DSL is broken down into more manageable predicates, which can be checked by evaluating the existence of simple relationships or properties in the code base of the target system.

This level of indirection allows the user to avoid dealing with more technical notations (*e.g.*, XML, Java) while using a friendlier high-level language. This allows for a wider range of stakeholders to take part to the design process. In fact, very little technical skill is required to read and write rules in Dictō. This may partially limit the control of the designer over the final specification. It is the responsibility of the adapter developer to expose the right amount of configurability to the end-user.

Dictō also provides support for model-to-code traceability. Traceability is achieved through declarative mapping directives defined together with symbolic entities. This lightweight mechanism has the advantage of being mostly unintrusive and comprehensible to untrained users. Concrete entities are automatically resolved by Probō, thus not requiring adapters and tools to provide support for any kind of resolution strategy.

A generative approach also allows us to minimize the amount of redundant information that needs to be maintained. Tool specifications are mainly built based on the information contained in a single uniform model that encodes the architectural rules defined by the user. Additional configuration attributes (*e.g.*, project source folder, source code language) are specified per project and are also shared among all adapters.

4.5.3 Specification Language Understandability

Dictō is a DSL designed to reflect how architectural rules are actually specified in practice. The language, as discussed in subsection 4.3.1, resembles basic specification patterns commonly encountered in industrial documentation artifacts. Dictō is sufficiently expressive to enable multifaceted modeling. The syntax of the language can be extended by installing new adapters or defining new concepts in Probō. This guarantees support for a wide range of highly diversified rules belonging to different domains and viewpoints.

Martin Fowler suggests that non-technical stakeholders (“business people”) should become more involved in the design decision process of a software system²⁰. He suggests that software rules should at least be read and understood when presented to a non-technical audience. Pruijt *et al.* [106] conclude on a similar note, recommending to “Minimize the difference between logical rules, as perceived by the architect, and technical implementation in the tool”. The DSL presented in this chapter may be used as an effective step towards achieving this objective. The syntax of our DSL is largely consistent with other solutions [115, 109] and formalisms presented

²⁰<http://www.martinfowler.com/bliki/BusinessReadableDSL.html>

in academic literature [106]. In the future we plan to evaluate the actual usability of our language by involving different users in an experiment and asking them to write, understand and adapt a pre-defined set of rules. By involving people with different backgrounds and measuring the success rate for solving these tasks, we aim at finding out how well the DSL matches practitioners needs from a usability and knowledge management standpoint.

We are also currently involved in a project that aims at integrating Dicto into the development process of a major open-source web-based learning management system (Ilias²¹). Our partners are interested in monitoring the architectural integrity of their system and supporting developers in the process of identifying relevant candidates for reengineering [28]. Throughout the project, we will have the chance to verify to which extent rules can be defined and understood by the numerous stakeholders involved in the project.

4.6 Related Work

Our approach is designed to evaluate declaratively defined rules by using third party analysis tools. We here review existing architecture conformance tools and ADLs.

4.6.1 Architecture conformance tools

Architecture conformance tools have been analyzed and compared in various studies. De Silva *et al.* [26] proposes a taxonomy for categorizing existing techniques and approaches. Prujit *et al.* [106] and Passos *et al.* [103] compare multiple tools by evaluating their capabilities through an experiment. In both studies the authors conclude that existing tools offer complementary features and none of them can be considered as a perfect replacement for all the others. In a previous study (see chapter 3) we run a survey to discover which tools practitioners use to test architectural constraints.

Table 4.2 shows an overview of the most prominent conformance testing tools that accept a textual specification as input. Many solutions (*e.g.*, DCL, inCode.Rules) verify constraints on relationships between classes and modules (*e.g.*, access, declaration, extension). Some languages (*e.g.*, SOUL, LogEn, SCL) focus more on structural properties of classes and methods (*e.g.*, identifiers, keywords, constructs). inCode.Rules [80] also detects code smells (*e.g.*, God class, Data class).

²¹<http://www.ilias.de>

²²<http://classycle.sourceforge.net>

²³<http://www.ndepend.com>

²⁴<https://semml.com>

	DCL [115]	TamDera [48]	inCodeRules [80]	SOUL [93]	LogEn [32]	SCL [56]	ArchFace [118]	ArchJava [3]	Classycle ²²	Ndepend/CQLinq ²³	Semmler.QL ²⁴
constraint types											
- relationships	✓	✓	✓	✓			✓	✓	✓		
- elements				✓	✓	✓				✓	✓
- code smells			✓								
detected RM relations											
- convergence	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
- absences	✓	✓	-	✓	-	✓	✓	✓	-	✓	-
- divergencies	✓	✓	✓	-	✓	-	-	-	✓	✓	✓
DSL extensibility											
- new predicates	-	-	-	✓	✓	-	-	-	-	-	-
programming language [J: Java; S: Smalltalk; P: Prolog; D: Datalog; C: C++]											
- analyzed system	J	J	J	J/S	J	J/C	J	J	J	.net	J
- tool implem.	J	P	J	S	D	J	J	J	J	.net	J

Table 4.2: Comparison among conformance checking tools based on a textual DSL.

The large majority of these solutions, with the exception of DCL[115] and TamDera[48], are only able to detect one of the following violations: absences or divergencies[96]. Only two of the reviewed solutions offer support for language-level extension (*i.e.*, SOUL [93] and LogEn [32]). Both are logic programming languages in which new predicates can be defined by composing existing predicates. The general lack of support for extensibility limits the expressiveness of the solution. Almost all techniques, with the exception of ArchFace [118] and ArchJava [3], assume that architectural constraints are specified in a separated text file. ArchFace and ArchJava require the user to define constraints directly in the source code by using special constructs that are checked at compile time.

4.6.2 Architecture description languages

ADLs allow us to describe the architecture of a system in a formal, declarative and human-readable way. Existing ADLs cover a wide range of use cases and fulfill various practical needs (*i.e.*, analysis, customization, *etc.*). Despite this fact, most ADLs are completely ignored by practitioners [65, 79].

Some languages allow the user to implicitly define constraints by supporting the specification of meta-annotations on first-class model entities. AADL [36] has a pre-

defined catalogue of properties for its different component types. xADL 3.0 [24] allows the user to define new entity attributes by customizing the XML schema. ACME [42] supports the specification of arbitrary named attributes for both components and connectors. Unicon [111] can handle a pre-defined set of attributes introduced to constrain the structure and relationships of components. MetaH [9] allows the user to define timing-related constraints through component attributes. Rapide [75] is one of the few ADLs that provides a rich vocabulary of well documented constraints over observable events. Each constraint is defined as a set of boolean conditions that is expected to hold or not hold when a specific event occurs. SADL [95] allows run-time invariants on the state of a component to be defined. Wright [4] supports the definition of architectural styles which may include constraints over the defined configuration. UML [47] models can be enriched through OCL [100], a textual declarative language used for defining rules regarding elements and relationships of a model.

According to various studies, ADLs fall short in fulfilling the following requirements:

Extensibility: “Most ADLs are quite restrictive and impose a particular architectural model on the architect, which often isn’t appropriate” [121]. In a study by Malavolta *et al.* “about 68% of respondents extended the ALs [(architectural languages)] they used by adding new views (about 48%) or constraints (13%) or both” [79].

Usability: ADLs “need to be simple and intuitive enough to communicate the right message to the stakeholders involved in the architecting phase, but shall also enable formality so to drive analysis and other automatic tasks” [79]. “Heavyweight and complex ALs often deter practitioners. A good combination of features fulfilling practitioners’ needs is crucial for adoption, and closing the gap between industry and academia” [79].

Multifaceted modeling: In a large study “about 85% of respondents declare to use multiple views for architectural description” [79]. The type of views mentioned are: “structural (76%), behavioral (48%), physical (45%) and conceptual (41%)” [79]. Unfortunately, “many ADLs do not support multiple viewpoints” [54].

In our approach we propose Dictō, a DSL that dynamically adapts to the features offered by the adapted evaluators. Usability concerns are addressed by offering a compact and intuitive language that reflects current practice. The possibility of modeling different heterogeneous aspects of a system is guaranteed by the generality of the language. In fact, Probō can be extended to reflect concerns related to various domains.

4.7 Conclusion

We presented a novel approach that aims at optimizing the cost of architectural conformance checking. Software architects have the possibility to declaratively define

and automatically check architectural rules without directly dealing with the idiosyncrasies of currently available tools. With our approach we reduce the effort required to describe and maintain rules, involve a larger number of stakeholders in the design process and effectively test system conformance reusing the functionality offered by state-of-the-art tools.

5

Assisted Quality Improvement

Studies have shown that developers are often disappointed by the results produced by static analyzers [61, 108]. The information provided to the end-user is considered insufficient and only partially helpful to solve the issue at hand [98, 61]. Many practitioners also express interest in being assisted with refactoring suggestions and quick fixes [61].

In the previous chapter we introduced an approach that aims at reducing the overall cost of conformance checking by offering increased usability for the end-user while enabling rich extensibility for contextual adaptation. At the end of this process, the user expects to be confronted with all the violations that invalidate the specified architecture. Violations can be presented with a varying degree of detail. Depending on the complexity of the system and the nature of rule leading to the violation, the effort to fix a violation might be bearable or exceeding the allocated budget.

In this chapter we investigate how we can help users to assess and react to architectural violations by providing actionable results. In our analysis, we focus on a specific type of constraint which typically leads to expensive violations entailing considerable costs for refactoring. This type of constraint, predicating the absence of cyclic dependencies among packages, is relatively easy to check but hard to sustain. Existing tools provide details regarding the elements causing a violation, but fail in supporting the user in devising a resolving strategy. By complementing the results with a semi-interactive quality improvement tool, we may offer actionable suggestions that help to quantify the effort required to eliminate the violation and save the user from the cost of evaluating multiple inconclusive refactoring alternative strategies.

To validate the feasibility of our idea, we developed a tool that provides developers with advises on how to remove package cycles. Our tool, Marea, automatically suggests the most cost-effective sequence of refactoring operations that will break the unwanted cycle. The optimal refactoring strategy is determined based on a custom profit function. Our approach has been validated on multiple projects and was successfully integrated in Probö.

5.1 Dependency cycles

In this chapter, we explore the advantages of having actionable suggestions as part of the results produced by a conformance checking analysis. We focus on a specific type of architectural flaw, *i.e.*, dependency cycles among packages.

Dependency cycles are a typical symptom of bad design [81, 82, 39] and are often linked to architectural erosion [78] and defect-proneness [101]. Two or more packages belong to a cycle if they contain classes which are circularly dependent (See subsection 5.2.1 for a more precise definition).

Empirical studies show that cycles can be found in almost any medium to large object-oriented software system [89, 87]. Dependency cycles introduce deployment constraints, forcing developers to bundle packages that are logically uncoupled, and generally increase maintenance costs. Excessive coupling amongst packages reduces the overall modularity of the project, precluding the possibility to homogeneously distribute the development effort between the members of the team. Scarce modularity has also a negative impact on testability, since isolating the functionality of low granularity units becomes more complicated. Martin defines the Acyclic Design Principle [82] as one of the rules that govern the structure of object-oriented software systems.

Given the proven importance of this architectural anti-pattern, many tools have been developed to detect dependency cycles. Most of them are commercial tools (*e.g.*, Structure101¹, Lattix LDM², SonarGraph³) and are commonly used by industrial practitioners. The main functionality offered by most of these tools consists in presenting a rich visualization of the package cycles existing within a given project. Other tools (*i.e.*, JooJ [90]), prevent the introduction of new cycles by monitoring the development environment and offering real-time warnings.

One fundamental limitation of all the existing techniques is the absence of a convenient support for removing the detected cycles. Automatic fixes and refactoring suggestions are highly appreciated by developers [108, 61]. Based on our interaction with practitioners, we found that developers are often forced to undergo multiple stages in order to eliminate a cycle. Refactoring actions are repeatedly interleaved with reverse engineering steps, during which the user checks the impact of the applied modification. This can lead to a highly ineffective non-linear process that contributes to frustration and higher maintenance costs.

Some tools (*e.g.*, Pasta [52]) have tried to cope with this limitation by introducing support for simulating basic refactoring operations over a reverse engineered model of the analyzed project. Users can drag and drop code elements (*i.e.*, classes, methods) from one container (*i.e.*, package, class) to another and immediately see how this impacts package-level dependencies. Unfortunately, this kind of process is only

¹<https://structure101.com>

²<http://lattix.com>

³<https://www.hello2morrow.com/products/sonargraph>

a slight improvement over the previously described one. In fact, the user still needs to perform subjective choices with little guarantee that the outcome of his action will eventually lead to the complete removal of the cycle. In addition to that, the refactoring operations supported by these tools are very elementary. Other more sophisticated techniques (e.g., dependency injection) often used in practice are simply ignored.

5.2 Basic Concepts

In this section we briefly introduce the main concepts characterizing the domain of application of Marea. In order to make the description as concrete as possible, we choose to restrict the scope of the discussion to systems developed in Java.

5.2.1 Terminology

Marea has been designed with the purpose of detecting and removing package-level dependency cycles. This form of cycle is detected by representing a system as a graph where the nodes are packages and the edges are the dependencies between them. Such a graph might contain strongly connected components (SCCs) that are composed by one or more cycles. A cycle is a closed walk with no repetitions of vertices and edges allowed.

If an entity x depends on another entity y , we write $x \rightarrow y$.

We categorize the dependencies in a system as follows:

- Class Dependency (CD): concrete dependency relating a class to another. $A \rightarrow B$, if the class A contains a reference to the class B .
- Package Dependency (PD): conceptual dependency between packages resulting from the aggregation of one or more CDs. In more concrete terms, suppose class A in package P_A depends on class B in package P_B . Then P_A depends on P_B , and the CD $A \rightarrow B$ is contained in the PD $P_A \rightarrow P_B$.
- Shared Package Dependency (SPD): package dependency that is present in more than one cycle.

Class dependencies are further classified into:

- Inheritance Dependency: $A \rightarrow B$, if A is a direct subclass of B or A implements B (in case B is an interface).
- Reference Dependency: $A \rightarrow B$, if
 - a class field of A is of type B (Class Field).

- a class field of A is initialized with an object of type B (Initialized Class Field).
 - a variable defined in a method in A is of type B (Local Variable).
 - a variable defined in a method in A is initialized with an object of type B (Initialized Local Variable).
 - a parameter of a method in A is of type B (Parameter).
 - the return type of a method in A is of type B (Return Type).
- Invocation Dependency: $A \rightarrow B$, if a method in A invokes a method in B .

5.2.2 Refactoring Strategies

In our approach we use 4 distinct refactoring strategies. In the remainder of this section we describe the different strategies and their applicability constraints.

Move Class (MC) : This refactoring strategy moves a class from one package to another. This refactoring strategy has previously been used by Shah *et al.* [110] to untangle dependency cycles.

In Figure 5.1 (Before), the package *components* \rightarrow *control* (we will ignore the specifics of this dependency in the interest of simplicity) and the class *Button* \rightarrow *Light*, which induces the reverse dependency *control* \rightarrow *components*. This cycle could be broken by simply moving *Button* to the package *components*. This refactoring strategy is very simple but may not be semantically consistent with the overall design of the system. In fact, in this case, the class *Button* should not be moved to *components*.

We will use the notation “MC: A to P ” to describe the operation where class A is moved to package P .

Move Method (MM): This refactoring is similar to Move Class. It has been previously investigated by Tsantalis *et al.* [117] as a means to remove Feature Envy bad smells.

Let’s assume that the method *Button.press()* depends on the class *Light* (in Figure 5.1 (Before)). This invocation dependency could be removed by moving the method from its original class (*Button*) to the target of the dependency (*Light*). We will use the notation “MM: M to A ” to describe the operation where method M is moved to class A .

Abstract Server Pattern (ASP): This refactoring, described by Martin [83], is inspired by the following principle: “Depend upon Abstractions. Do not depend upon concretions”.

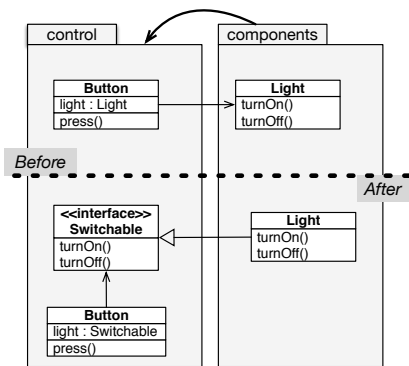


Figure 5.1: The 'Abstract Server Pattern' refactoring strategy

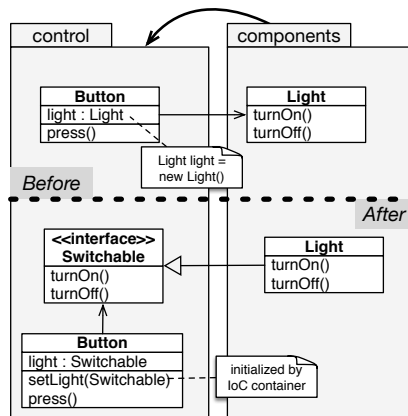


Figure 5.2: The 'Abstract Server Pattern + Dependency Injection' refactoring strategy

The refactoring can be used to invert the direction of a dependency in case its target is a concrete class. In Figure 5.1, *Button* has a field of type *Light*. This dependency can be inverted by creating a new interface, *Switchable*, in the package containing the class from which the dependency is originated. This interface will then be implemented by the class on the other end of the dependency (*Light*). By applying this simple operation we inverted the dependency from *control* to *components*, and we eliminated the cycle.

We use the notation "ASP: *SourceElement* for type *TargetClass*" to describe an instance of this refactoring (in our example, we would use: ASP: control.Button. light for type components.Light).

Abstract Server Pattern + Dependency Injection (ASP+DI): This refactoring⁴ is an extension of the previously introduced ASP refactoring. We use dependency injection to eliminate initialization code responsible for dependencies. In our example (Figure 5.2), we first apply the ASP refactoring. Since *Button.light* is originally initialized using the default constructor of *Light*, we remove the assignment in the field declaration (*Light light = new Light()*) and declare a new method (*setLight()*). This method will eventually be invoked by an Inversion of Control (IoC) container (e.g., Spring⁵) when the object is created. The actual value used for the initialization could be defined in a configuration file and the field might be annotated with a framework-specific annotation.

Using this technique has a positive impact on maintainability, as dependencies can

⁴<http://www.martinfowler.com/articles/injection.html>

⁵<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html>

be easily changed to adapt to new requirements or a different runtime environment. On the other hand, the code will contain implicit references and will require some sort of configuration logic in order to function properly. Fortunately, many applications (e.g., J2EE, Spring) already rely on a dependency injection framework and therefore would not be required to introduce drastic architectural changes in their system. Moreover, the introduction of semi-implicit indirection may have a negative impact on understandability.

Since ASP+DI is essentially a more refined version of ASP, this refactoring can be used to handle all the cases that are handled by its simpler counterpart. Nevertheless, every refactoring technique presented in this section has a different cost (i.e., implementation time) and semantic validity associated to it. The choice between one refactoring or another cannot be automatically determined and must be taken by the user. In our approach we simply test the applicability of each refactoring in the context of the analyzed system.

We will use the notation “ASP+DI: *SourceElement* for type *Class*” to denote an instance of this refactoring.

5.2.3 Strategy Applicability

Each refactoring strategy can be applied to break a variable number of class dependencies. We provide an exhaustive overview showing the applicability of the various strategies for each type of dependency (See Table 5.1).

dependency type	MC	MM	ASP	ASP + DI
Inheritance	✓	-	-	-
Class Field	✓	-	✓	✓
Initialized Class Field	✓	-	-	✓
Local Variable	✓	✓(*)	✓	✓
Initialized Local Variable	✓	✓(*)	-	✓
Parameter	✓	✓(*)	✓	✓
Return Type	✓	✓(*)	✓	✓
Invocation	✓	✓(*)	-	✓(*)

Table 5.1: Applicability of the refactoring strategies (asterisk stands for limited applicability)

The MM refactoring strategy is not applicable in case the dependency is caused by an inheritance relationship (e.g., *Button* extends *Light*) or a class field dependency (e.g., *Button.light* is of type *Light*). In those cases the refactoring cannot be applied as no method is involved in the dependency.

The ASP refactoring strategy cannot be used in case the dependency is caused by an initialized class or local variable (e.g., *Button.light* is initialized to a concrete instance of *Light*). In those cases, one should opt for the ASP+DI refactoring. This strategy

is also not applicable to inheritance and invocation dependencies, as its application would not completely invert the original dependency responsible of the violation.

The ASP+DI refactoring cannot be used in presence of an inheritance dependency (e.g., if *Button* is a subclass of *Light*). In this case, the refactoring operation would break desired properties deriving from the dependency (e.g., behavior reuse).

Special conditions apply in the circumstances marked with an asterisk (Table 5.1). MM cannot be applied to remove the indicated dependencies if the method presents one of the following properties:

- The method is a constructor.
- The method returns *this*.
- The method accesses a variable with class-scope.
- The method has an invocation to a static method defined in the same class.

Furthermore, ASP+DI cannot be used if the dependency is caused by an invocation and the invoked method is static, a constructor, or *super()*, as none of these can be defined in an interface.

5.3 Our Solution

Marea executes in three phases (see Figure 5.3): (A) Initially it analyzes the input system to detect cycles; (B) then it explores all possible refactoring sequences; (C) and finally it suggest the most cost-effective refactoring sequence to the user. In the remainder of this section we describe each phase in more detail. The prototype is available for download on the web⁶.

5.3.1 Analyze Cycles

Detect Cycles – In order to start the process we have to identify the dependencies in the target system. We do so by running a fact extractor⁷ based on Eclipse JDT⁸. The extractor visits all the nodes of the AST reconstructed by the underlying Eclipse platform and uses that information to build a FAMIX [27] model. This model will later be imported in Moose [99], an analysis platform designed to simplify the task of querying and manipulating FAMIX models. Once imported, the model is ready to be analyzed. To detect the cycles, we define a graph $G = (V,E)$, where V are all the packages contained in the obtained model and E are the package dependencies (as defined in subsection 5.2.1) existing among them. We use Tarjan’s algorithm

⁶<http://smalltalkhub.com/#!/~caracciolo/Marea/>

⁷<https://gforge.inria.fr/projects/verveinej/>

⁸<http://www.eclipse.org/jdt/>

[113] to detect the strongly connected components (SCC) in the graph. Each SCC is subsequently untangled into individual cycles.

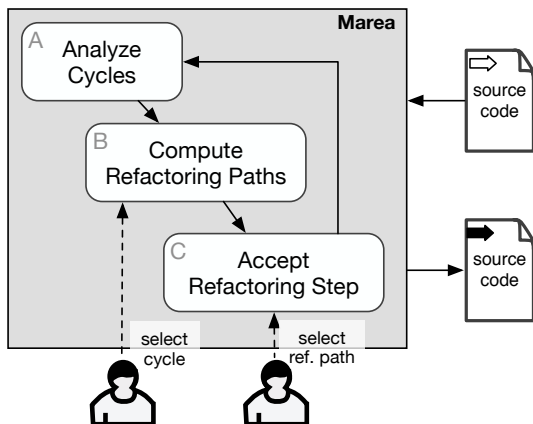


Figure 5.3: The three main process phases of our solution

This phase can be executed in linear time (*i.e.*, the computational complexity of Tarjan’s algorithm is $O(|V| + |E|)$).

Rank Cycles – A typical project may contain a high number of cycles. To help the user to optimize the refactoring effort, we rank the cycles based on their overlapping degree and their size. This means that smaller cycles containing dependencies that are shared among more cycles will be presented before larger ones with fewer shared dependencies. To define our ranking, we sort all the cycles contained in the system (in descending order) based on the following formula:

$$rank(cycle) = \frac{|SPD_{cycle}|}{|CD_{cycle}| * |PD_{cycle}|} \quad (5.1)$$

where CD_{cycle} , PD_{cycle} , SPD_{cycle} are the sets of all the CDs, PDs and SPDs contained in $cycle$.

In the example illustrated in Figure 5.4, we have three cycles. Cycle 2 ranks best because it has two SPDs and a relatively low number of PDs and CDs.

Rank Package Dependencies – Cycles are formed by at least two package dependencies (PD). PDs are aggregations of multiple class dependencies and can be sorted based on the following function:

$$rank(pd) = \frac{|\{cycle \mid pd \in SPD_{cycle}\}|}{|CD_{pd}|} \quad (5.2)$$

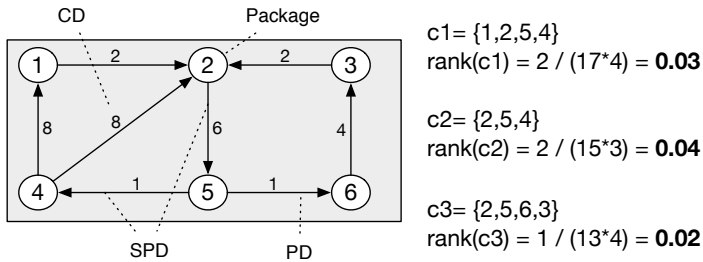


Figure 5.4: Example: ranking of 3 cycles

where CD_{pd} is the set of all the CDs contained in the package dependency pd and SPD_{cycle} is the sets of all the SPDs contained in a *cycle*.

This ranking function favors package dependencies that are shared among many cycles and are composed by a low number of class dependencies. These dependencies have the highest probability to break multiple cycles at the lowest cost (*i.e.*, refactoring steps are usually proportional to the number of CDs). An alternative method for ranking package cycles is discussed in a recent work by Falleri *et al.* [33]. In their approach, they assign a higher rank to cycles that involve packages that are more distant in the package containment tree (*e.g.*, $\text{rank}(\{x, x.y.z\}) > \text{rank}(\{x.y, x.y.z\})$; where $x, x.y, x.y.z$ are the packages forming the ranked cycles). They also assume that cycles with PDs with a lower number of CDs are more likely to have been introduced accidentally and therefore should be ranked higher. The last assumption is also reflected in our function.

5.3.2 Compute Refactoring Paths

User action: select Cycle – When the analysis of the target system is complete, the user is asked to select the cycle she wants to remove. Her choice may be strongly influenced by her expertise, past experience and priorities. For example, a cycle that involves a core module is most probably more important than one between utility packages.

Simulate Refactoring Paths – To find the best refactoring sequence, we build a decision tree where each node represents a mutation of the model originally extracted from the system (Phase A). Each model variation is actually computed by cloning an existing model and applying the modifications prescribed by one of the supported refactoring strategies. The memory footprint of every new model is limited by the fact that only the changes are saved (and not a full new model).

Figure 5.5 illustrates an hypothetical situation in which a class dependency, part

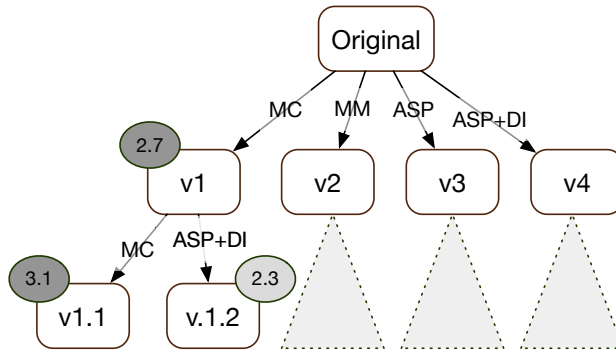


Figure 5.5: Decision tree representing possible refactoring sequences

of the currently analyzed cycle, can be broken using all four refactoring strategies⁹. Since all refactorings are applicable, we create 4 variations of the original model (v1-4). For each new model we search again for cycles and recompute the dependencies that still need to be removed in order to eliminate the cycle. In the model v1, we might have to remove a dependency of type *initialized class field*. Since this dependency can only be removed using MC or ASP+DI, we obtain 2 new variations of v1 (v1.1-2).

The depth of the tree will grow depending on the level of accuracy that needs to be reached. Sometimes class dependencies might be impossible to break (*e.g.*, the suggested refactorings are semantically inconsistent with architectural design choices). In this case more elaborated design changes are required. In other cases, the removal of a class dependency may introduce new cycles in the system, causing an overall negative effect on the quality of the system. If this happens, the resulting refactoring sequence should be applied with caution or avoided.

The construction of a tree terminates when the cycle at hand has been completely removed. This means that every path connecting the root node to any leaf represents a complete refactoring sequence that leads to the elimination of the given cycle. The termination of the tree construction process is guaranteed to terminate, because at least one refactoring strategy (*i.e.*, MC) can always be applied. The process can be prematurely terminated for performance reasons by defining a maximal tree depth.

Suggest Optimal Path – To distinguish between effective and potentially harmful refactoring operations, we define a profit function that summarizes the gain obtained by applying that operation. The function should favor measurable quality improvements and should penalize high-effort refactoring operations. The function is defined as follows :

⁹This might be the case when the class dependency is of the type: *local variable; parameter; return type; or invocation*. See Table 5.1

$$P = \frac{w_0}{cycles + 1} + \frac{w_1}{depth} + w_2 \frac{(1 - I_{from}) + (1 - I_{to})}{2} + w_3 \frac{A_{from} + A_{to}}{2} \quad (5.3)$$

where:

- *cycles* is the total number of cycles in the system.
- *depth* is the depth of the node within the decision tree.
- $I_{from/to}$ quantifies the instability [83] of the package from/to which the class dependency is directed. This metric is an indicator of the package's resilience to change. The value can range from 0 to 1, where 0 indicates a completely stable package.
- $A_{from/to}$ quantifies the abstractness [83] of the package from/to which the class dependency is directed. This metric indicates the percentage of abstract classes contained in the given package. The range for this metric is 0 to 1, with 1 indicating a completely abstract package. According to the Stable Abstractions Principle¹⁰, "packages that are maximally stable should be maximally abstract".
- w_i are constants that can be tuned to assign less or more weight to the single components of the equation. In our experiments, we chose to have all weights set to 1.

The profit function was designed to guide the user towards the best result (*i.e.*, low number of dependency cycles and high structural quality) by minimizing operational costs (*i.e.*, low number of required refactoring steps). The profit value of a refactoring sequence is generally inversely proportional to the number of cycles existing in the system and the number of operations composing the given sequence. Improvements in structural quality (decrease of instability or increase in abstractness) contribute to increase the profit. Weights (*i.e.*, w_i) have been introduced to accommodate project/organization specific customization. These parameters could be tuned manually or automatically (based on previous interaction sessions).

This profit function is used in our approach to calculate the utility value of the single nodes composing our decision tree. As the tree is completely constructed, we will review all the leaf nodes and select the one with maximum profit. The refactoring sequence corresponding to the path connecting the root of the tree to this node is the sequence that will (according to our profit function) remove the cycle at the lowest cost. In our example (Figure 5.5), the best refactoring sequence is Original \rightarrow v1 \rightarrow v1.1. This refactoring sequence has a profit value of 3.1. In this example we assume that all the sub-trees below v2-4 lead to lower profit leaf nodes.

¹⁰<http://www.objectmentor.com/resources/articles/stability.pdf>

5.3.3 Accept Refactoring Path

User Action: Select Refactoring Path – The optimal refactoring sequence might be composed of a large number of refactoring steps. Some of these steps might be applicable to the system under analysis while others might be semantically inconsistent with the overall design of the project (as intended or formally specified by the stakeholders). To opportunely guide the user in the process, we present her only with the refactoring actions that compose the optimal path. Other sub-optimal refactoring sequences can be examined on request.

Update Dependency Graph – Once the user has chosen to apply a specific refactoring sequence, the process updates its internal model and starts again from the beginning (*i.e.*, phase A). All the selected refactoring operations are only simulated and not applied to the code. Refactoring sequences could be exported in a textual format and serve as an input for another tool that will perform the actual modifications to the source code.

5.4 Evaluation

To test the applicability of our approach, we used our proof-of-concept prototype to analyze two projects (one open-source and the other commercial). In this section we describe the outcome of our experiments.

5.4.1 JHotDraw

JHotDraw¹¹ is an open-source project developed by Gamma *et al.*, often chosen as a reference object-oriented system for its sound design and rich use of design patterns. We analyzed version 6.0 beta 1, consisting of 485 Java files and a total of 28,000 non-comment lines of code. During our analysis we detected 44 package cycles. Six of them could probably be considered irrelevant, as they only involve packages belonging to the test modules (*i.e.*, `org.jhotdraw.test.**`). The remaining ones are more likely to be considered harmful and often involve packages (*i.e.*, `org.jhotdraw.contrib`, `org.jhotdraw.util`) that, at first sight, should not belong to cycles. A subset of cycles encountered in our analysis is presented in Table 5.2.

To test our prototype we followed the steps described in section 5.3. Since we could not find an expert user who could help us discarding sub-optimal refactoring options, we blindly followed the suggestions offered by the tool without considering the semantic implications that this could have on the overall design of the system. As a result, we managed to discover a refactoring that allowed us to break all 44 cycles in 7 refactoring steps. This result was achieved through the following steps

¹¹<http://www.jhotdraw.org>

(summarized in Table 5.2):

cycle #1 (*{contrib, samples.javadraw, contrib.zoom}*¹²): This cycle was removed by inverting the dependency *contrib* → *samples.jhotdraw*. This dependency has been selected as best candidate for removal because of its relatively smaller number of comprising class dependencies (2 compared to the 18 dependencies existing in the opposite direction) and its high number of dependency sharing (14 other cycles share the same dependency). The cycle was removed by moving the class *SVGDrawApp* from *contrib* to *samples.jhotdraw*. This was not only the optimal refactoring solution, but also the only one applicable. In fact, the class *SVGDrawApp* inherits from *org.jhotdraw.samples.javadraw.JavaDrawApp*. Therefore the only applicable refactoring strategy is Move Class (see Table 5.1). Moving the class to *samples.jhotdraw* automatically removes the second dependency, consisting of an invocation to the parent constructor (using the super construct). After this step, the system still contained 26 unresolved cycles.

cycle #2-5 (2: *{contrib, contrib.zoom}*, 3: *{test, test.samples.pert}*, 4: *{standard, contrib, samples.javadraw, framework}*, 5: *{standard, contrib.dnd}*): As the analysis continues, more complex cycles are analyzed. Cycles 2, 3, 5 offer two refactoring options while cycle 4 offers only one. In all four cases, the MC refactoring appears to be the more convenient option (according to our profit function Equation 5.3). Proposed alternatives feature a profit score that is very similar to the chosen optimal counterpart. This means that, according to our profit function, all non-chosen refactorings would have been good candidates for the refactoring. In cycle #2, the second best refactoring option suggested by our tool was ASP+DI on *contrib.CustomSelectionTool.showPopupMenu(..)* for type *contrib.zoom.ZoomDrawingView*. This option scored 1.35 profit points, compared to the 1.36 of the chosen refactoring strategy. Also in cycle #3, the alternative solution (*i.e.*, MM *test.AllTests.suite()* to *test.samples.pert.AllTests*) had a similar score compared to the optimal solution (1.44 compared to 1.52). After this step, the system still contained 3 unresolved cycles.

cycle #6 (*{framework, util}*): The weaker package dependency in this cycle consists of nine class dependencies. For performance reasons, we decided to limit the depth of the simulation tree to a maximum of three. The suggested refactoring reported in Table 5.2 was obtained by combining the optimal paths computed during three subsequent simulation phases. During each simulation step, the tool evaluated an average of almost 30 different scenarios. The whole process had to be split into three phases because of the considerable memory requirements required to compute the individual simulation trees. This was done by applying the suggested refactorings

¹²A dependency cycle is described as a set of packages, where the element in position N depends on N+1 and the last depends on the first. The common package name prefix “org.jhotdraw” has been removed for readability purposes.

#	Refactoring Sequence
1	MC: <i>contrib.SVGDrawApp</i> to <i>samples.javadraw</i>
2	MC: <i>contrib.CustomSelectionTool</i> to <i>contrib.zoom</i>
3	MC: <i>test.AllTests</i> to <i>test.samples.pert</i>
4	MC: <i>standard.StandardDrawingView</i> to <i>contrib</i>
5	MC: <i>contrib.dnd.DragNDropTool</i> to <i>standard</i>
6	MC: <i>framework.Handle</i> to <i>util</i> ; MC: <i>framework.DrawingView</i> to <i>util</i> ; ASP: <i>framework.HandleEnumeration.nextHandle()</i> for type <i>util.Handle</i> ; ASP: <i>framework.DrawingEditor.getUndoManager()</i> for type <i>UndoManager</i> ; MC: <i>framework.Tool</i> to <i>util</i> ; MC: <i>framework.DrawingEditor</i> to <i>util</i> ; MC: <i>framework.Locator</i> to <i>util</i> ; MC: <i>framework.Figure</i> to <i>util</i> ; MC: <i>framework.Connector</i> to <i>util</i> ; MC: <i>framework.ConnectionFigure</i> to <i>util</i> ; MC: <i>framework.Drawing.findFigureInsideWithout(int,int,Figure)</i> to <i>util</i> ; MC: <i>framework.DrawingChangeEvent.DrawingChangeEvent(Drawing,Rectangle)</i> to <i>util</i> ; ASP: <i>framework.DrawingChangeListener.drawingInvalidated(...).e</i> for type <i>framework.DrawingChangeEvent</i> ASP: <i>framework.DrawingChangeListener.drawingTitleChanged(...).e</i> for type <i>framework.DrawingChangeEvent</i> ASP: <i>framework.DrawingChangeListener.drawingRequestUpdate(...).e</i> for type <i>framework.DrawingChangeEvent</i> MC: <i>framework.Locator</i> to <i>util</i> ;
7	MC: <i>util.UndoableCommand</i> to <i>standard</i> MC: <i>util.UndoableTool</i> to <i>standard</i> MC: <i>util.Figure</i> to <i>standard</i> MM: <i>util.ConnectionFigure.startFigure()</i> to <i>util.Figure</i> MM: <i>util.ConnectionFigure.endFigure()</i> to <i>util.Figure</i> ASP: <i>util.GraphNode.node</i> for type <i>standard.Figure</i> MC: <i>util.GraphLayout</i> to <i>standard</i> MC: <i>util.ConnectionFigure</i> to <i>standard</i> MC: <i>standard.RedoCommand</i> to <i>standard</i> MC: <i>standard.UndoCommand</i> to <i>standard</i> MC: <i>util.JDOStorageFormat</i> to <i>standard</i> MC: <i>util.UndoableHandle</i> to <i>standard</i>

Table 5.2: JHotDraw: detected cycles and refactoring sequences applied to remove them.

in intermediate stages and restarting the analysis taking the new version of the system as input.

The refactoring steps reported in Table 5.2 show that our tool tried to break the dependency from *framework* to *util* by applying the MC and ASP refactorings in a precise sequence. The first operation consists in moving the class *framework.Handle* to *util*. This operation introduces a new dependency between the two packages *framework* and *util*, since *framework.HandleEnumeration.nextHandle()* has a return type *Handle*, and *Handle* is now in package *util*. To address this issue, the tool proposes to apply ASP on *framework.HandleEnumeration.nextHandle()* (operation 3). The subsequent operations contribute further to reducing the number of dependencies. Only the 8th operation (i.e., MC: *framework.Figure* to *util*) appears to be problematic. In fact, by moving *Figure* into its new package, we introduce new dependencies. This happens because another class contained in *framework* is heavily coupled with *Figure* (i.e., many methods return *Figure* or require arguments of type *Figure*) and moving *Figure* to *util*, we automatically add 23 new dependencies from *framework* to *util*. This contributes to increasing the effort required to remove the cycle. The new dependencies are slowly removed by further applying the MC and ASP refactorings. After

this step, the system still contained 1 unresolved cycle.

cycle #7 (*framework, util*): This last iteration also had to be split into multiple phases. The number of steps within the optimal refactoring sequence is relatively contained (12 refactoring operations), but the memory resources required to compute all the possible alternative paths was considerable. Also in this case the tool preferred to resort to MC in most of the steps. However, in few cases, it also suggested to use the MM and ASP refactoring. MM could be seen as a variant of MC with a lower impact on the design of the system (since a smaller part of functionality is moved across packages). The reason why MC is often preferred over MM is that it provides a means to remove dependencies in a smaller number of steps. If no other class depends on the moved class, then no negative side effect will result from the refactoring.

5.4.2 Industrial Project

To further evaluate our tool, we approached a team working in one of the largest IT companies in Switzerland. The team, composed of four people, was actively developing a module that was part of a larger project. The version that we could analyze consisted of 865 Java files distributed across 159 packages for a total of 50.000 non-comment lines of code.

Our contact person explained to us that the team prided itself for dedicating special care to good object-oriented design practices. The quality of their software was, according to them, superior to that of other modules developed within the same project. The team also regularly used SonarQube¹³, a quality assessment tool that, among other things, offers a report of all the cycles contained in a project.

During our analysis, Marea found 25 cycles formed across 22 packages. We asked our collaborator (a member of the development team) to select the most relevant cycles. From those, we chose to analyze the cycle represented in Figure 5.6. This cycle involves two packages (*i.e.*, *[..].scout.client* and *[..].scout.server.services.process.stubs*) and is caused by a total of four dependencies involving three classes. Any detail that may reveal the identity of the company has been omitted as explicitly requested. Packages and classes have been labeled for convenience.

To break this cycle, Marea ranks the package dependencies based on Equation 5.2. Since the dependency between *C1* and *C2* is caused by one single invocation, the tool proceeds with the computation of all refactoring paths that may lead to its removal. The result of the simulation phase is shown in Figure 5.7. As we can see, Marea evaluates 4 alternative refactoring sequences. The first one consists in moving the class *C1* to the package *P2*. This simple refactoring not only eliminates the dependency from *P1* to *P2*, but also the one from *P2* to *P1*. In fact, at the end of the

¹³<http://www.sonarqube.org>

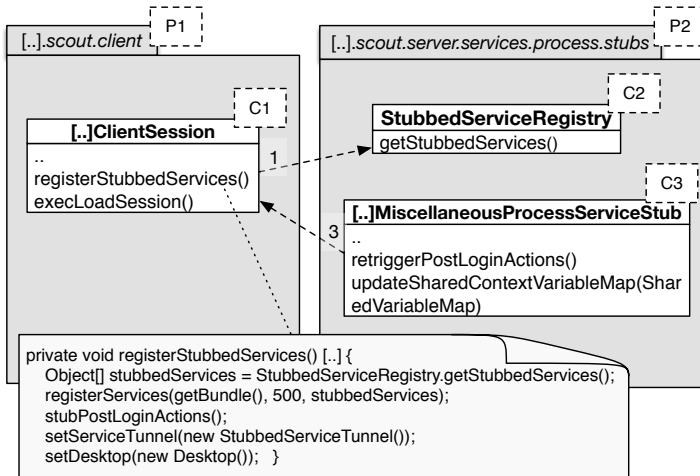


Figure 5.6: One of the cycles found in the industrial project

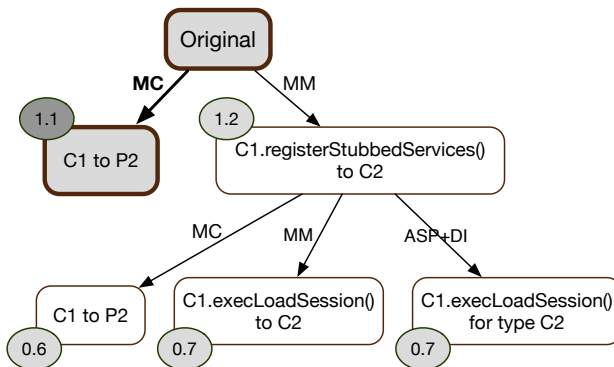


Figure 5.7: Simulation tree for the cycle found in the industrial project

refactoring, all the classes involved in the cycle will be in one single package. This solution obtains a profit score of 1.1, which makes it the best refactoring sequence. An alternative solution, would have been to move the method *C1.registerStubbedServices()* to C2. This operation is similar to the previously described move class, with the difference that only the method involved in the dependency needs to be moved. Unfortunately, another method in C1 (i.e., *execLoadSession()*) depends on the moved method. Because of that, the dependency from P1 to P2 is still not broken and further steps must be taken. The tool evaluates 3 options: MC of C1 to P2; MM of *C1.execLoadSession()* to C2; ASP+DI of *C1.execLoadSession()* for type C2. The first refactoring simply replicates the approach taken in the optimal solution. The second one moves the method that is now causing the dependency between the two packages to its target class. This solution breaks the cycle since *C1.execLoadSession()* has

no incoming dependencies. The last refactoring inverts the dependency between the two packages by removing the explicit reference to *C2* in *C1.execLoadSession()* using ASP+DI.

To test the full applicability of the obtained solutions, we performed all the proposed refactoring operations using a common IDE. We initially opted for Eclipse¹⁴, but eventually had to find an alternative. In fact, Eclipse supports all the required refactoring operations but failed in performing MM of *C1.registerStubbedServices()* to *C2*. The reason is that the move method refactoring “cannot be used to move potentially recursive methods”. Despite the fact that *C1.registerStubbedServices()* is by no means recursive, we could only complete the operation manually. In a second attempt, we chose to use IntelliJ IDEA¹⁵. This IDE correctly performed the MM refactoring by first adding a static modifier to *C1.registerStubbedServices()* and then moving it to its target class. All subsequent refactoring operations were also correctly applied. In the end we verified that all proposed solutions are sound and lead to the complete removal of the cycle.

We asked one of the developers to comment on the proposed solutions. After an attentive analysis of the code, he explained that:

- Solution 1 (MC), is simple and might be applicable if there were no restriction on how the two packages are deployed. Unfortunately client and server are deployed separately. Moving the class *C1* to the package *P2* would require to include client libraries into the server bundle.
- Solution 2 (MM + MC), is meaningless since it reaches the same result as in Solution 1 but in two steps instead of one.
- Solution 3 (MM + MM), is not a complete solution since the moved method (*execLoadSession()*) overrides a method of its superclass. The method also calls itself using the super keyword (i.e., *super.execLoadSession()*). Moving the method to another class would break the functionality of the method. This particular solution should have been discarded automatically by our tool. The discovery of similar edge cases can be considered acceptable during the initial trial period. We will address similar issues in following development iterations.
- Solution 4 (MM + ASP/DI), is the solution of choice of our subject. This refactoring is more time consuming and complex compared to all other ones, but cleanly separates the concepts contained in the two packages. The functionality remains where it has been originally placed and a newly introduced interface serves as a contract between the two classes causing the dependency.

One additional solution, that might apply only in this particular case, suggested by our user was to move *P2* under *P1*. In fact, *P2* only contains lightweight classes with few dependencies and no business logic. These classes have been created for testing

¹⁴<https://www.eclipse.org>

¹⁵<https://www.jetbrains.com/idea/>

purposes and do not necessarily need to be deployed with any of the two packages. Making $P2$ into a sub-package of $P1$ would quickly remove the cycle and only introduce a minor semantic inconsistency. On one hand, the classes contained in $P2$ should logically belong to the *scout.server* package hierarchy, as they relate to the server domain. On the other hand, moving them to $P1$ would be acceptable because they are exclusively used for testing purposes. This solution would not completely remove the cycle, but, for our user, cycles contained inside architectural components (*i.e.*, *scout.server*, *scout.client*) are considered to be of secondary importance.

Solution 4 has been chosen as best refactoring strategy because it reflects a standard refactoring adopted by the team to remove dependencies. In fact, many classes deployed separately on client and server, share common interfaces. Concrete dependencies are resolved through a dependency injection mechanism based on AOP and the exchange of serialized classes over HTTP. The preference expressed by the user for this refactoring strategy could be explicitly factored into the profit function (*i.e.*, in Equation 5.3, add another term: $w_4 \times \text{number_of_ASPDI}$).

Solution 2 showed that the algorithm might lead to a meaningless solution when the proposed refactoring path reaches a previously explored subgraph. This point can be used to optimize the construction of the decision tree.

Solution 3 brought up another corner case that needs to be treated with special care. This case will be addressed by implementing a dedicated guard in the algorithm responsible for building the decision tree.

5.5 Discussion

5.5.1 The Package Blending Problem

Our approach is always guaranteed to reach a solution, as long as enough time and memory are provided. In fact the MC refactoring can be applied in any circumstance, and will therefore always be used to approach an optimal solution. The problem is that the MC refactoring might often not represent the most desirable type of refactoring from a semantic point of view. The change implied by this refactoring is only justifiable if the behavior described in the moved class is consistent with the category described by the target package. If we ignore this reasoning and blindly move classes from one package to another (as we intentionally did in subsection 5.4.1), we will end up gradually dismantling the modularity of the system. This behavior may possibly lead, in the extreme case, to the complete blending of one package into another. If that happens we will have eliminated a package dependency causing a cycle but we would have significantly altered the high-level structure of the project. This contrasts with the separation of concerns design principle, which advocates the isolation of cohesive functionality.

This general tendency towards the unification of packages should primarily be prevented by the user. The profit function often suggests MC as an optimal refactoring strategy (See Table 5.2), as it offers a fast and uncomplicated way to get rid of a dependency. Another approach to control the abuse of the MC refactoring strategy, could consist of extending the profit function (Equation 5.3) with an additional metric that measures package cohesion. One such metric could be an opportunely adapted variation of LCOM [55], a metric that quantifies the number of responsibilities of a given functional unit, or CRSS [88], a metric for good package design.

A further option to control the execution of the simulation algorithm could be to support the specification of structural invariants that define the boundaries of allowed refactorings. These invariants could explicitly forbid the relocation of classes contained in specific modules or the separation of entities sharing some semantically relevant property. The specification of the architectural components (in terms of sets of packages) would already be sufficient to prevent design breaking changes (*e.g.*, moving classes between a *client* and a *server* component).

5.5.2 Prototype Limitations and Tradeoffs

Simulating multiple refactoring scenarios, as we have seen in section 5.4, has its cost. One of the main issues encountered during our experimentation is related to the amount of memory required to store each evaluated simulation step. In our prototype we tried to optimize memory consumption by using in-memory object models that can be evolved by only saving the incremental changes that separate one version of the model from another. This was possible thanks to a third-party framework called Orion [71]. Despite the advantage of incremental change memorization, we still encountered several cases in which the simulation exhausted the available memory resources. This issue could probably be addressed by improving the Orion framework or opting for a more scalable data management approach for storing our models (*e.g.*, a graph database). Another viable solution consists in optimizing the simulation process by enhancing the profit function (subsection 5.5.4) or by taking into account complementary information (*e.g.*, architectural rules, build configuration files) that leads to the definition of structural constraints.

During our experiment we decided to cope with the above mentioned limitation by pruning the simulation tree at a pre-fixed depth. This choice might have a negative impact on the accuracy of our technique. In fact, a partial sub-optimal refactoring sequence could theoretically develop into an optimal solution in further steps of the simulation. By varying the maximum length of a simulation path, we can vary the tradeoff between the overall cost of the analysis and its level of precision.

Another significant aspect that needs to be considered when implementing our approach is the level of correctness of the simulated refactoring operations. In our prototype, as mentioned, we delegated the versioning of our simulation models to

a dedicated framework. This framework allows us to evolve existing models by applying predefined basic change operations (*e.g.*, create/delete class, create/delete method). The refactoring strategies discussed in this chapter (subsection 5.2.2) had to be implemented by combining several of those change operations. This implies investigating and managing all possible edge cases, updating references and maintaining the system in a generally consistent state. This whole spectrum of complexity is well managed in commercial refactoring engines, but requires considerable effort to be implemented from scratch. In our prototype, we did our best to address all encountered issues and to handle the most recurring cases. Despite our effort, we recognize the fact that implementing a fully correct refactoring operation requires a considerable engineering effort. Furthermore, the completeness of the implementation also depends on the level of detail of the meta-model used to represent the system under analysis. If certain details of the system are only partially captured in the model, then those details are not guaranteed to be correctly updated to reflect the applied refactoring. We plan to address the above mentioned limitations by engaging in further experimentation and performing comparative studies with current implementations of refactoring algorithms (*e.g.*, Eclipse JDT refactoring engine¹⁶).

5.5.3 Refactoring Application

As explained in subsection 5.4.2, applying the refactoring operations suggested by Marea is not always easy. Our approach simulates the refactoring operations on models that are only partially as complex as the reality they represent. This means that all the details necessarily omitted in our models may play a role in the actual applicability of the suggested operations. We try to prevent complex situations by analyzing the properties of the involved code elements (*e.g.*, MM cannot be applied on methods that contain invocations to other static methods defined in the same class). Our approach can be considered safe and unobtrusive. A possible evolution of our approach could be based on the speculative application of refactoring step in a code sandbox. This approach would be more pragmatic but might possibly require more computation time, as the analysis model (on which decisions are taken) needs to be reverse engineered after each iteration.

5.5.4 Profit Function

The profit function used in our approach and described in Equation 5.3, is a simple attempt at quantifying the effect of a refactoring strategy over a project. It was designed to guide the user in choosing a better solution based on objective measurements. Given that the optimal solution can only be decided by the user, we assume that this function should only be used as a general heuristic for comparing competing alternative solutions. The metrics employed in the calculation are well known

¹⁶<http://www.eclipse.org/jdt/>

indicators of the structural stability of software packages. The function could be improved by adding further metrics and changing their relative impact by modifying their weight coefficients. A more adequate combination of weights could be inferred by performing empirical studies and recording the paths typically chosen by the majority of the users.

5.6 Related Work

Providing automatic support for the removal of dependency cycles is a complex problem. We here report on the main research directions that have developed around this topic.

5.6.1 Refactoring Candidate Identification Heuristics

Some approaches are specialized in detecting the most critical elements in a dependency cycle. These approaches do not advise explicitly on how to remove a cycle, but rather provide hints on where to look in order to devise a proper refactoring strategy.

Melton *et al.* present Jepends [89], a tool that identifies the classes that should be refactored in order to remove a cycle. Each class in the system is analyzed and ranked based on its number of incoming/outgoing dependencies and on the number of cycles it is involved into. Classes that most contribute to cycles and with higher coupling are considered to be the best starting point for further inspection and consequent refactoring.

Jooj [90] is another tool by Melton *et al.* that warns the user about the existence of cycles as soon as they appear. The warnings are displayed within the IDE and further instructions may be provided to remove critical dependencies. The main assumption behind this solution is that, as long as the user is aware of the impact of his actions, new cycles will not be introduced. The authors declare their intent to implement refactoring suggestions based on patterns described by Lakos [68] (*e.g.*, escalation, demotion, dumb data, manager class). No further information is provided regarding the challenges involved in adding this specific feature.

Laval *et al.* introduce CycleTable [70], a visualization technique that should guide the user in the identification of critical code elements involved in cycles. CycleTable does not focus on a single solution to break cyclic dependencies. It rather groups classes based on their coupling profile. This approach could be compared to the one used in Jepends [89], as both aim at classifying code units based on structural metrics.

Laval *et al.* also present another tool, Ozone [72]. The tool suggests which dependencies should be removed in order to obtain a layered package structure. The target

architecture is inferred automatically based on the analysis of the source code and a set of optional user-defined dependency constraints. The refactoring steps required to concretely obtain the target architecture are left to the user to investigate.

All the mentioned approaches only provide small hints regarding what concretely needs to be done to break a dependency cycle. In fact, presenting a list of candidates for refactoring without further instructions on how to perform the actual refactoring task only partially contributes to solving the problem of cycle removal.

Oyetoyan *et al.* [102] propose a new heuristic metric that can be used to guide the removal of intra-class cycles. Their algorithm identifies an optimal refactoring strategy that maximizes a set of structural metrics (*e.g.*, coupling) and promotes specific implementation choices (*e.g.*, class attribute static modifier). The resulting refactoring is then applied to update an annotated graph model representing the system. The algorithm can be applied multiple times until all class cycles have been removed. The main difference between this approach and Marea is the fact that Marea explores multiple refactoring options and evaluates combinations of multiple refactoring strategies during each analysis step. Instead of simply applying a pre-configured default refactoring, Marea simulates complete refactoring scenarios and is therefore capable of measuring the impact of the overall refactoring instead of just focusing on devising a solution based on step-wise optimization. Marea also focuses on package cycles, instead of class cycles.

5.6.2 Refactoring Simulation

Many commercial tools provide support for removing cyclic dependencies by simulating refactoring operations.

In Structure101¹⁷, the user can move classes, methods, fields and packages using drag-and-drop actions. No guidance is provided during the process. Refactoring operations, such as move method, seem to be always easily applicable without considering the effects that such an operation would involve when applied on the corresponding code elements. The dependency graph of the analyzed system is hard to navigate and cannot be reduced to isolate single cycles.

SonarGraph-Architect¹⁸ allows the user to identify the dependencies that cause a cycle. These dependencies are ranked and described at the granularity of classes. No hints are provided on how to remove the identified dependencies. The user can move classes from one package to another and observe the impact of the operation on the dependency structure of the system.

Lattix LDM¹⁹ visualizes cyclic dependencies using a Dependency Structure Matrix [109]. Besides finding cycles, the tool also supports basic structural editing features

¹⁷<https://structure101.com>

¹⁸<https://www.hello2morrow.com/products/sonargraph>

¹⁹<http://lattix.com>

such as the renaming, moving and deletion of packages. This limited set of operations may help removing certain cycles, but will also most probably encourage users to reduce the modularity of their system.

Pasta [52] is a tool developed by Compuware. Pasta, like Ozone [72], tries to derive the best layering configuration for a given system and presents the user with all the dependencies that need to be removed to implement that configuration. Pasta also offers a graphical interface that supports the simulation of several refactoring operations by drag-and-drop: move package, move class. Simulated changes can eventually be automatically applied to the code. The author claims that a future version of the application will also support advanced refactoring operations described by Martin [82].

All the presented tools deal with dependency cycles in the same terms as one would approach a graph problem. Each node composing the cycle can be moved around regardless of the complexity of its underlying implementation. Little or no guidance is provided on which operation may offer the best compromise between effort and benefit. The user has the responsibility to decide between many refactoring options that can only often be performed only on larger granularity elements (packages, class). The gap between the simulated change operations and the actual refactorings that might eventually be applied on the corresponding code elements remains large.

If the editing features provided by the current commercial solutions could be extended with more sophisticated refactoring operations and supported by an intelligent decision support system, we might have a complete solution for dealing effectively with cyclic dependencies.

5.7 Conclusion

In this chapter we introduce a novel approach to guide developers in the task of removing cyclic dependencies among packages. We propose a tool that simulates various refactoring operations and identifies the optimal change sequence based on a profit function. We also report on the challenges that we encountered during the implementation and evaluation of the tool.

We conclude that assisting a user during the removal of an architectural violation (in this case a cyclic dependency) is possible and worthwhile. Our prototype illustrates the basic phases that such a process should support.

The addition of the described technique within the approach proposed in chapter 4 can help to further reduce the cost of conformance checking. The user would be relieved from the burden of deriving a refactoring strategy by his own and will benefit from having sufficient information for assessing the effort required to solve the issue at hand.

6

Industrial Validation

In chapter 3, we investigated the type of constraints that software architects are interested in checking and discovered a wide range of requirements. Only a small fraction of them is well supported by existing tools, and where tools exist only a smaller part of the developer community is aware of them. We discovered that attempts at automating architectural conformance checking often ended with failure, given that the resources invested in the task often exceeded the allocated budget. Practitioners are open to adopt quality assessment tools, but are not willing to pay the cost of deployment and maintenance activities.

To relieve them from this additional cost, we developed a solution that allows users to formulate architectural rules using a simple high-level domain specific language (DSL) and automatically have them checked by third-party analyzers (See chapter 4). This solution has the potential to aggregate the functionality of most existing quality assessment tools under the umbrella of a single uniform and readable language. By providing a familiar specification interface to the user and augmenting the analysis results with operationally relevant recommendations (chapter 5), we can reduce the overall cost of the process by relieving the user from automatable tasks.

To evaluate the effectiveness of our solution we applied our tool suite in the context of three distinct industrial projects. In this chapter we describe and analyze the main results of our study. The case studies show that our approach has the potential to engage stakeholders in discussions that would otherwise probably never have taken place. Dictō, the DSL proposed as part of our solution, becomes a powerful instrument for expressing and communicating architectural rules. Relying on a highly extensible analysis platform allows developers to specify rules without taking directly into account the limitations of a particular analysis tool or the challenge of maintaining rules written in multiple definition languages for multiple analysis tools. Instead, they can quickly prototype rules and benefit from the reusability offered by adapters developed in other contexts.

6.1 Our Approach

Our goal is to streamline the process of validating architecturally relevant quality constraints. This is done by offering Dictō – a common declarative specification language as the main interface for the definition of rules and Probō – providing a highly automated and extensible platform for the integration of heterogeneous off-the-shelf analyzers. Dictō and Probō have already been described at length in chapter 4. In this section we briefly summarize their key characteristics.

Dictō is a DSL whose design is based on requirements collected in a previous empirical study (see chapter 3). It can be used to define entities and rules as in the following example:

```
Test = Package with name:"org.*.test.**"
only Test can contain dead methods
```

A schematic representation of the language grammar is presented in Figure 6.1. Further documentation can be found on our website¹.

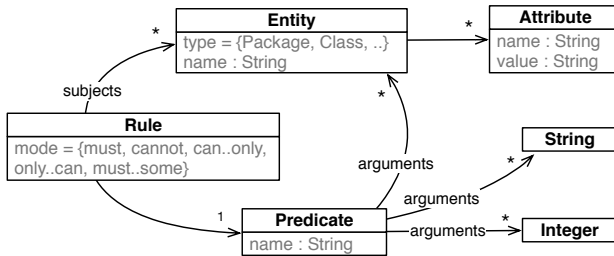


Figure 6.1: Dicto grammar (instantiated on previously mentioned example).

Probō: User-defined rules are evaluated by Probō. The application is based on a pipeline architecture that can be described by the following sequential phases:

- **Parsing**: In this phase we analyze the provided source code, extract all the necessary information and create an in-memory model of the target system.
- **Transformation**: All user-defined entities and rules are normalized and broken down into more manageable predicates. Those are forwarded to the most appropriate adapter, which generates a specification for the tool it supports. Adapters are lightweight data transformers that are built by technically specialized developers with deeper knowledge of the configuration and operation of a given target tool.

¹<http://scg.unibe.ch/dicto/>

- Analysis: External tools are launched using the generated specification. It is a tool's responsibility to evaluate the given predicates and provide the information necessary to identify the violations for the originally defined rules.
- Reporting: The output generated by the external tool needs to be interpreted and processed to separate failing rules from passing ones. A report file summarizing the outcome is eventually generated.

6.2 Case Studies

We evaluate our architectural monitoring solution in case studies with three development teams working on distinct projects in two different companies (See Table 6.1).

#	Organization domain (n. employees)	Project tech. - size	Team size
C1	Transportation (1.000+)	J2EE - 50 K	5
C2	Transportation (1.000+)	J2EE - 0.5 M	30+
C3	e-Learning (12 vendors)	PHP - 1 M	25

Table 6.1: Summary of case studies.

In the first case study (**C1**) we dealt with a group of senior developers responsible for the development of a small, but architecturally very relevant, project within a large swiss transportation company. The project consisted in a framework used by the majority of projects developed in the company. The team was already using SonarQube², a popular quality monitoring tool for overseeing some general aspects of the evolution of the system (*e.g.*, common coding anti-patterns, dependency cycles). They showed genuine interest in our solution, but, after a first pilot project, they abandoned the idea of actively supporting a full integration of our solution inside their development process.

In **C2** we worked together with another team within the same company as **C1**. This time we had the chance to interact with a person who championed our solution until full deployment. During this second case study we had numerous iterations in which we discussed rules and reviewed the resulting violations. In the end we deployed our tool on one of the development workstations used in the project.

In **C3** we collaborated with 18 developers working for 12 vendors of an open-source learning management system called ILIAS³. Those developers were all members of a special interest group (SIG) established to discuss reengineering opportunities for improving the system at the architectural level. The underlying motivation for

²<http://www.sonarqube.org>

³<http://www.ilias.de/>

founding this group stemmed from the fact that the system has evolved for over 18 years without the guidance of a person responsible for defining and enforcing a sustainable architectural policy. After multiple iterations, we integrated our tool into Teamcity⁴, a continuous integration server used to build the core module of the ILLAS application.

Each case study was organized around the following phases:

1. **Endorsement seeking & Process definition:** The first step is getting in touch with a contact person from the organization and persuading her of the value of the offered solution. This is typically done by setting up an introductory meeting during which we present the main features of Dictō, some sample rules currently checked by Probō, and an outlook over possible integration options with currently employed monitoring solutions. After gaining the support of our contact person, we attempt to discuss a deployment strategy for our solution. This step must be tailored to the specific practices and needs of the organization. The chances of success grow if the solution is introduced in an unobtrusive, transparent and gradual way.
2. **Rule elicitation & formalization:** Stakeholders have typically different requirements regarding which kind of architectural rules need to be defined. They typically vary depending on the technologies adopted in the project and the domain of the system. After outlining the requirements, we define a set of rules that reflect all identified constraints. Rules are directly specified by the user using Dictō and are iteratively refined to maximize their readability and reusability.
3. **Feedback automation:** All defined rules need to be checked automatically on a regular basis.

To enable this behavior, we develop the necessary Probō adapters (or reuse existing ones) and integrate our solution into the existing quality control system (e.g., continuous integration server, dashboard). To make the solution effective, we need to stimulate the interest of the developers working in the organization. This can be done by raising awareness (inviting users to acknowledge the current violations and warning developers upon the introduction of new violations) and rewarding users performing corrective maintenance.

In **C2** and **C3**, we successfully deployed our solution within the organization. In **C1** we only reached phase 1. In this case the team failed to obtain the support of management to fully deploy the solution in the context of their project.

The total duration of the case studies **C2** and **C3** was almost 1 year each. **C1** ended prematurely after 1 month.

⁴<https://www.jetbrains.com/teamcity/>

6.3 Evaluation

6.3.1 Endorsement Seeking & Process Definition

In all the three case studies, we initially interacted with one person who later supported us in introducing the concept of our solution to the rest of the organization.

In **C1**, we started a pilot project with the support of our primary contact person. This person was a user of the framework being developed in the project taken into consideration. As a user, he knew which kind of constraints needed to be enforced on the developed code. These constraints were partially documented in an internal wiki and partially derived from direct experience and orally shared knowledge. The contact person was genuinely interested in the evaluation the proposed solution and thought that the team working at the project could well appreciate our effort. To guarantee a successful introduction of the proposed solution in the context of the project, we suggested to integrate the results produced by our tool into the software quality monitoring dashboard already in use within the team. As we presented our results to the leaders of the team, the general idea was well-received. Unfortunately the extent of the presented rule set (in Listing 1) failed to convince them of the full utility of the solution. The people attending the meeting commented that most rules were, to some extent, already checked by other tools. Despite the flaws described in section 6.4, they preferred not to invest any additional resources into improving their current quality monitoring infrastructure. Their focus was also primarily on structural aspects of the source code. They were skeptical towards introducing rules that were not already tested (either manually or using commercial tools). The rule set derived from the pilot project (Listing 1) is ultimately representative of some of the constraints that needed to be checked in the project. Further cooperation could have led to a more exhaustive and representative sample of rules.

```
1 SYSTEM cannot contain cycles
2 PersistencePackage cannot depend on ServicePackage
3 ImplicClass must have annotation "@πService"
```

Listing 1: Pilot rules defined for case study C1 (π is the name of the project).

For anonymization purposes, we will use the symbol π as a way to implicitly refer to the name of the projects analyzed in **C1** and **C2**.

Similarly, in **C2**, we started our collaboration through a pilot project. Our contact person was a developer working full time on the development of the project being examined. He suggested to start by re-evaluating rules that were already tested by another commercial tool currently employed within the project (Sonargraph⁵). After assessing the effectiveness of our tool, he started proposing new rules that were either defined in documented guidelines or that he, based on his experience in the

⁵<https://www.hello2morrow.com/products/sonargraph>

project, suspected of being important for maintaining the architecture of the system. His main interest was in revealing existing architectural flaws and simplifying the tasks involved in performing qualitative maintenance. The rule set presented to the team consisted of 17 rules, mostly focused on code dependencies (See Listing 2).

As we presented our results, we agreed that the definition of the rules could be undertaken by any member of the team, while the development or refinement of new or existing adapters would initially require our intervention. The task of maintaining adapters could eventually be transferred to a selected member of the team following a personalized training sessions.

In C3, we established contact with a person who had interest in introducing a solid quality monitoring solution within his organization. He is a co-founder of a special interest group (SIG) established to promote and discuss project-wide reengineering tasks that would improve the maintainability of the project. In order to implement any new design specification, the SIG needed a mechanism to control which aspects of the new architecture were correctly implemented and which part of the source code still needed to be refactored towards the new design. Our solution offered the help needed to define and check the actual realization of the prospective architecture. The idea of adopting our solution required the approval of the SIG, the head of development and some key members of the community (*i.e.*, mostly representatives of the various service providers). The SIG was easily convinced of the utility of the tool. They acknowledged the technical benefit but were worried about the political implications of introducing and maintaining such a solution. The issue was discussed with the top management of the organization several months later, as we eventually obtained permission to deploy the tool on a global scale. During this last meeting, it was decided that new rules, discussed within the SIG, would need to be approved during the bi-weekly physical meeting moderated by the head of development. Our contact person would be involved in maintaining the necessary adapters and would share his duties with other members of the development team as a means of disseminating his expertise.

6.3.2 Rule Elicitation & Formalization

In each case study, we specified a set of rules that reflected some major architectural concerns identified within the projects taken in consideration. Those rule sets were defined in Dictō based on initially elicited requirements. In the interest of space, we omit the definition of entities. The complete specifications can be found on our website⁶.

In C2, we defined the rule set in Listing 2.

```
1 ClientScoutPackage can only depend on SharedScoutPackage
2 ServerScoutPackage can only depend on SharedScoutPackage, ServicePackage
```

⁶<http://scg.unibe.ch/dicto/case-studies.php>

```

3 ServicePackage can only depend on BusinessPackage
4 BusinessPackage can only depend on ServicePackage, PersistencePackage
5 CoreProject cannot depend on StammdatenProject
6 BetriebProject can only depend on AngebotProject
7  $\pi$ Project can only depend on AngebotProject, BetriebProject
8 ServiceUiMethods, ServicePublicMethods must throw
9      $\pi$ ServiceException
10 ServiceImplClasses must have annotation "@ $\pi$ RemoteService"
11  $\pi$ Batch cannot depend on  $\pi$ UiImpl
12  $\pi$ Batch cannot depend on  $\pi$ PublicImpl
13 Persistence cannot depend on Service
14 Batch cannot depend on Persistence
15 ScoutClient cannot depend on ScoutServer
16 Util, Model can only depend on Util, Model
17  $\pi$ Project can only depend on  $\pi$ Project, CoreProject, StammdatenProject, AngebotProject
18 ModelClasses, DTOClasses must implement "java.io.Serializable"

```

Listing 2: Rules defined in case study C2 (π is the name of the project).

The rule set is largely based on documented guidelines and previously checked constraints. The definition process required multiple iterations that took place over a period of about 6 months. Each iteration allowed us to identify erroneous violations and discuss over the specification of new rules or new language constructs.

In C3, we defined the rule set in Listing 3.

```

1 WholeIliasCodebase cannot invoke triggerError
2 WholeIliasCodebase cannot invoke exitOrDie
3 WholeIliasCodebase cannot invoke setErrorOrExceptionHandler
4 WholeIliasCodebase cannot invoke eval
5 WholeIliasCodebase cannot depend on SuppressErrors
6 ilExceptionsWithoutTopLevelException can only depend on ilExceptions
7 GUIClasses cannot depend on ilDBClass
8 GUIClasses cannot depend on ilDBGGlobal
9 only GUIClasses can depend on ilTabsClass
10 only GUIClasses can depend on ilTabsGlobal
11 only GUIClasses can depend on ilTemplateClass
12 only GUIClasses can depend on ilTemplateGlobal
13 IliasTemplateFile cannot contain text "on(blur|change|click|dblclick|focus|keydown|keypress|keyup|
    load|mousemove|mouseup|mousedown|mouseenter|mouseleave|mouseout|mouseover|mousewheel|resize|
    select|submit|unload|wheel|scroll)"
14 IliasTemplateFile cannot contain text "<script*>"
15 WholeIliasCodebase cannot invoke raiseError
16 IliasTemplateFile cannot contain text "javascript*:"

```

Listing 3: Rules defined in case study C3.

In C3, the rules were initially specified by our contact person. As soon as they were made public, other members of the SIG started to propose their own rules. They suggested three new rules (*i.e.*, line 4, 5, 15 in Listing 3). Their suggestions often consisted in syntactically valid specifications posted in the community forum together with questions like: "Can this be checked?". All the rules were formulated based uniquely on previously presented examples. No formal training was required. The rules defined within the SIG were later discussed in a physical meeting during which additional rules (*i.e.*, 7-14, 15 in Listing 3) were proposed. In this case, all rules were

proposed by members of the community that have never been in any way exposed to Dictō.

Some of the specified rules reflect general best practices (*e.g.*, avoid the invocation of disrupting functions – line 4) while others define constraints related to new architectural concepts that need to be implemented over time (*e.g.*, new exception handling policy – lines 1-3). The remaining rules are mostly there to ensure a correct separation of concerns (*e.g.*, MVC pattern – lines 7-12).

All rule sets presented here are in their final form which was reached after multiple refinement iterations. All major changes applied during this process are discussed in section 6.5.

6.3.3 Feedback automation

Our tool has been successfully integrated with pre-existing monitoring and continuous integration solutions. In particular we managed to integrate with the SonarQube dashboard in **C2** and with TeamCity continuous integration server⁷ in **C3**.

To integrate with SonarQube, we developed a plugin that evaluates user-defined rules in Probō and transforms all reported violations into “issues”. SonarQube was already used as an issue tracker within the team to define and assign development tasks. By silently adding architectural violations as issues, we were hoping to unobtrusively deliver our results to the stakeholders involved in the trial evaluation. Unfortunately, due to the company policy, we were not able to customize the SonarQube installation used by the whole team. Instead we installed our integrated solution (*i.e.*, SonarQube with the Probō plugin) on the workstation of a developer.

In **C3**, we integrated our tool suite with TeamCity, a newly introduced continuous integration service that was made accessible to the whole community. By developing a plugin, we managed to expose analysis results in a separate view inside the web dashboard. Users could view which violations had been added or removed since the last build and obtain a list of all those that were currently unresolved. If a developer introduced or solved a violation she would receive an email notification remaining her of the event. We also introduced a leaderboard where all contributors are ranked according to the number of fixed violations. This stimulated users to contribute more and to pay attention to previously ignored quality concerns.

6.4 Results

At the end of our studies we measured how rule violations were introduced or removed over time.

⁷<https://www.jetbrains.com/teamcity/>

In **C2** we detected a total of 270 violations. These violations were treated as follows: 27 (lines 8,9 and 17 in Listing 2) were classified as critical and fixed immediately; 158 (lines 3, 7) were considered of secondary importance and listed in the issue tracker; 85 (lines 1, 6, 14) were not fixed. Not addressed violations were mainly ignored because of the high complexity involved in the refactoring task. In fact, two rules (lines 1, 14) involved user interface dependencies, while another rule (line 6) concerned a module which was no longer actively maintained. Nine out of twenty-seven rules were correctly observed in the implementation and did not lead to violations.

In **C3** we monitored the violations introduced and removed over an arc of two months. During this time the total number of violations decreased from 606 to 600 (*i.e.*, 10 violations were introduced and 16 removed). Given the size and age of the system (1M lines of code and 18 years of development), we consider that to be a positive outcome. During this initial trial period, we contacted several developers who either introduced or removed a violation. Contributors responsible for introducing violations reported different reasons for their action, such as intrinsic complexity of the context (*i.e.*, making the contribution violation-free would have required major changes) or general lack of time. One user said that the feedback “definitely leverages the discussion about architecture and separation of concerns”. They also considered the rule that they violated to be reasonable and legitimate. Users who removed violations were mostly concerned with enhancing the quality level of a module they developed or to increase their score on the leaderboard.

In **C3**, the analysis of some violations led to the discovery of repeating anti-patterns. For the rule on line 8 (Listing 3), for example, our collaborators observed that developers consistently referenced a global variable defined for database access in GUI classes. This was done to pass the reference down the invocation chain to model classes. The identification of this common practice led to internal discussions and to the decision to evaluate alternative dependency injection strategies. This case shows how our solution supports complete feedback loops and enables dynamics that were previously unattainable.

During the evaluation we found several cases in which quality assurance tools were already employed by the organization. In those cases we re-encoded the rules defined for the pre-existing tool into Dictō and discovered several divergences in the results. Sometimes we found false positives (*i.e.*, spurious violations reported by our tool). Those were typically due to imprecisions in the specification and could be quickly removed. More often we found false negatives (*i.e.*, violations not reported by the reference tool). False negatives can be symptomatic of a less precise analysis technique. Since the specifications were equivalent, it could be that precision is sacrificed for the sake of performance and scalability. The analysis of the encountered false negatives helped us to uncover some possible limitations of the previously adopted tools.

In **C1** we found that only three of the 18 package cycles identified in our analysis were actually reported by the employed tool (SonarQube). All the 15 cycles

ignored by the preceding tool were manually validated and categorized as actual violations.

Based on our analysis, SonarQube failed to detect cross-module dependencies. This means that if two classes located in two different projects reference each other, no cycle will be detected. Our case study project is organized into 46 Maven modules. This configuration reduces versioning conflicts and simplifies maintenance and deployment. In our experiment, SonarQube ignored cycles like $\pi.service.code \rightarrow \pi.service.i18n \rightarrow \pi.service.code$ (caused by dependencies among classes belonging to the respective packages but contained in different build modules).

SonarQube also seems to ignore indirect cycles (*i.e.*, cycles among more than two packages). In fact, a cycle like $\pi.service.code \rightarrow \pi.service.i18n \rightarrow \pi.service.i18n.code \rightarrow \pi.service.code$, failed to be detected as a violation.

Our analysis is based on hypotheses drawn from an end-user perspective. Many of the encountered false negatives could not be linked to any of the above mentioned conditions. To completely understand the reasons behind these errors, one should have access to the full details of the analysis algorithm.

In C2, developers were using Sonargraph to monitor dependency constraints. In our analysis we discovered 5 false negatives (Sonargraph reported 2 violations out of the 7 detected by Dictō). One possible explanation that could explain this inconsistency is related to the strategy used to reconstruct the dependency graph for the analyzed project. Based on our analysis, we suspect that Sonargraph detects dependencies by parsing the import statements contained at the beginning of each source file. Our tool relies on a parser that extends Eclipse RCP. This allows for a more sophisticated dependency resolution strategy that traverses indirect references and locates the true endpoints of a dependency.

In our case studies the technical leaders of the projects did not suspect any incompleteness in the previously obtained results. Both tools employed by our partners have a solid reputation. SonarQube is the de-facto standard for lightweight technical debt management and its large user-base is typically seen as a proof of its reliability. Sonargraph is one of the leading solutions for checking dependency violations and is often seen as a primary choice for monitoring architectural quality.

False negatives are particularly hard to discover. End users are typically not aware of them since the complete validation of the analysis results is practically infeasible and would require the inspection of the whole code base (not just the reported violations). Discrepancies among results produced by different tools could be detected by developing and running multiple adapters for the same type of constraint and automatically comparing the violations reported by each analyzer. Providing a mechanism that supports cross-checking of results produced by different tools, greatly simplifies the task of comparing the accuracy of competing quality assurance solutions.

6.5 Discussion

6.5.1 Expressiveness in Practice

The expressiveness of Dictō evolved throughout the course of the case studies. Minor additions were made to the language itself. Other changes had to be performed at the level of single adapters, to support more precise specification mechanisms. We here provide a non-exhaustive list of changes that were discussed and implemented during the case studies:

I. Rule type: In **C3**, one of the users involved in the development of the system defined the following rule:

```
ilExceptions = PhpClass with name:"il*Exception*"
ilTopLevelException = PhpClass with name:"ilException"
ilExceptionsWithoutTopLevelException = {ilExceptions} except {ilTopLevelException}

ilExceptionWithoutTopLevelException must depend on ilException
```

This rule requires all application-specific exception classes (*i.e.*, all classes named “il*Exception*” except “ilException”) to depend on all the classes described by the entity ilException. This rule was semantically wrong, since any Exception class not depending on all ilException classes resulted in a violation. To fully express the user’s intentions, we introduced a new kind of rule that only failed when the rule subject entity did not depend on some of the elements described as the rule argument. The rule could thus eventually be rewritten in the following form:

```
ilExceptionWithoutTopLevelException must depend on some ilException
```

By revising the rule using the “must .. some” construct, we can express the condition that every element in ilExceptionWithoutTopLevelException must have a dependency directed to at least one element in ilException.

II. Entity specifiers: At the beginning of our case studies, package entities were described through selection attributes including simple wildcards (*i.e.*, “*”). In **C2**, we soon realized that this specification approach was not precise enough. In fact, a pattern like “org.*.x” was designed to greedily match any package name starting with “org” and ending with “x”, while our user wanted to have the option to either match a single identifier (*e.g.*, org.foo.x) or multiple ones (*e.g.*, org.foo.bar.x). To address this limitation we introduced a double wildcard character “**” (the syntax was inspired by Apache Ant⁸). This allowed us to properly describe entities in the following form:

⁸<http://ant.apache.org/manual/dirtasks.html#patterns>

```
BusinessPackage = Package with name:"π.*.business.**"
```

Similarly in **C3**, one of the users asked to define a rule that required the definition of a more complex argument value as a means to detect compliance to a specific naming convention (line 13 in Listing 3). This request could be addressed by specializing the adapter responsible for checking the rule and adding support for regular expressions.

III. Entity selection attribute modifiers: At the beginning of our evaluation, entities could only be defined by specifying a set of inclusive filters that described expected characteristics exhibited by the target elements. Soon enough, we were asked to also include the possibility of defining an exclusive filtering mechanism. This was done by introducing a negation modifier for the assignment operator used to define entity attributes. In **C1**, our user wanted to define a logical entity mapping to all the packages matching the following expression: "`π.*.persistence.**`". After analyzing the results we found out that many of the elements resolved for that entity were correctly matching the expression but were irrelevant in the context of the analysis. After introducing the new modifier (*i.e.*, "`!:`"), the rule could be rewritten as follows:

```
PersistencePackage = Package with name: "π.*.persistence.**", name!:"π.*.service.**"
```

IV. Entity grouping construct: To further support the definition of more complex entities, we also introduced a new language construct that enabled the conjunction and disjunction of sets derived from the combination of previously defined entities. This feature became a valid complement to the previously described selection attribute modifiers. Complex entities, such as the above defined *PersistencePackage*, could now be defined through the combination of other entities. In **C3**, for example, the user could define a new entity by combining other previously defined entities:

```
ilExceptionsWithoutTopLevelException = {ilExceptions} except {ilTopLevelException}  
WholeIliasCodebase = {ilClasses, assClasses}
```

V. Rule argument separators: In **C3**, we defined the following rule:

```
SetErrorOrExceptionHandler = {SetExceptionHandler, SetErrorHandler}  
WholeIliasCodebase cannot invoke SetErrorOrExceptionHandler
```

This rule clearly states that the argument is a disjunction of two different logical entities. The same pattern could be found in **C2**, where the user needed to specify that a package could not depend on multiple other packages:

```
ServerScoutPackage cannot depend on SharedScoutPackage
ServerScoutPackage cannot depend on ServicePackage
```

To support a simpler definition of such rules, we introduced a conjunctive (*i.e.*, “,”) and a disjunctive (*i.e.*, “/”) rule argument separator. The rules could be rewritten as follows:

```
WholeIliasCodebase cannot invoke SetExceptionHandler / SetErrorHandler
ServerScoutPackage cannot depend on SharedScoutPackage, ServicePackage
```

VI. Entity exclusion: As we analyzed the results obtained in **C2**, we quickly discovered that many true positives (correctly reported violations) were not relevant. These violations either referred to test classes or external libraries. Test classes are typically not reviewed for quality and are simply regarded as secondary artifacts with low maintenance priority. External libraries, on the other hand, are obviously out of the scope of the project and as such should not be checked against architectural rules. To exclude such exceptions, we initially thought of introducing a pre-parsing step where the user can define (through a script) which classes and libraries should be copied or ignored before building a model of the system. This solution was later on discarded because it reduced the overall accuracy of the parsing process. We eventually decided to specify the excluded artifacts in a project configuration file through the following property:

```
IGNORE-ENTITY:"org.eclipse.**; **.zlr**; **Test"
```

This filter was effectively used to exclude all the entities that were previously identified as noise in the results produced by our analysis.

6.5.2 Expressiveness in Theory

To further test the expressiveness of Dictō, we performed a literature survey and tried to encode architectural rules reported in other papers using our DSL. We collected 44 rules by reviewing various sources [120, 74, 103]. The full list can be viewed on our website⁹. All rules but three could be successfully expressed in Dictō. The rules that could not be specified (listed in Table 6.2) presented some characteristics (*e.g.*, conditional constructs, interrelation between multiple entity groups) that will be discussed in the following paragraph.

Dictō was successfully employed and evolved to accommodate the specification needs of the users participating in our case studies. Despite our best efforts, several rules (reported in Table 6.2) could not be encoded without introducing major changes to the language and to the underlying model. One type of rule that is currently not supported by Dictō is the one which predicates an invariant that may

⁹<http://scg.unibe.ch/research/arch-constr/eval/Expressiveness>

#	Rule
U1	The classes implementing interface <code>Tool</code> must implement method <code>activate</code> if method <code>isUsable</code> returns true.
U2	Calling method <code>getLocation</code> requires cloning the instance (calling method <code>clone</code>) to avoid that the receiver of <code>getLocation</code> can change the internal behavior of a <code>LocatorHandle</code> .
U3	The names of the attributes of class <code>FigureAttributeConstant</code> should be used as suffixes of the attributes of class <code>ContentProducer</code> starting with the prefix <code>ENTITY</code> .

Table 6.2: List of rules discovered during literature survey that cannot currently be specified using Dictō

apply only upon the fulfillment of a condition (U1-2 in Table 6.2). This kind of conditional rule did not appear during our case studies, but seem to be required in other contexts. To support such rules, we not only would need to add a new construct to the language but would also have to introduce the concept of conditional statement to our model. This addition would imply the implementation of a catalogue of parametrized conditional expressions for each supported entity type (since each adapter would need to provide the necessary functionality to evaluate the validity of the specified expression).

To better understand the impact of such a change, let's suppose that we decide to support this feature by extending our DSL in such a way that U1 could be written as follows:

```
ToolClasses = Class with superClass:"**.Tool"
isUsable = Method with name:"**.isUsable"
ToolClasses must have method "activate" (if "isUsable" returns true)
```

Probō would recognize that the subject entity `ToolClasses` is of type `Class` and would determine that the corresponding code element fulfills the conditional expression defined between parentheses. This feature is not currently planned for implementation, since we have no concrete evidence that it might be of interest for our end-users. One of the challenges of implementing such a construct would be to find a way to specify the relationship existing between terms included in the conditional block with entities mentioned in the rule. In fact, in our example, "`isUsable`" is not explicitly related to the subject element of the rule.

Another limiting factor, is the lack of support for expressing correlation between properties of the subject entity and values used as rule arguments (See U3 in Table 6.2). In **C1**, we encountered the following rule:

```
All classes ending by "Impl" must implement an interface that has the
same name as the class, but without the suffix "Impl".
```

This kind of rule could have been implemented in a very *ad hoc* fashion by defining a predicate that expresses exactly the above mentioned constraint. We decided to

avoid this solution and tried to conceive a more flexible rule that could express arbitrary naming patterns. Since at the time we could not come up with an adequate solution, we decided to ignore the rule.

Later on, after completing the case studies, we introduced a new language feature that allows the user to specify capture groups in entity selectors. The captured values can then be symbolically referenced in the predicate argument of a rule as shown in this example:

```
ImplClass = Class with name: "***.(*)Impl"  
ImplClass must have interface named "$1"
```

This additional feature allows us to create interrelations between subject elements and predicate arguments. Given its late introduction, we cannot comment on its practical applicability. In retrospective, we think it would have served as an elegant way for expressing that particular type of rule.

6.5.3 Usability

In our case studies we tried to establish short iterations in order to acquire as much feedback as possible. Our users often asked for minor non-functional changes that could potentially improve their specification from the point of view of readability and learnability. During the evaluation period we implemented the following requested features:

I. New entity types: In C3, we used a fact extractor called `PhpDependencyAnalysis`¹⁰ in order to extract the information needed to resolve user-defined entities and to test the required rules. This tool generates a two dimensional collection representing all the binary dependency relationships existing in the analyzed code base. After the first iteration we introduced a new type of entity that we called “*PhpDependency*”. Entities of this type correspond to nodes of the graph derived by combining the relationships identified by our extractor. The user could define entities such as:

```
eval = PhpDependency with name: "eval"  
ilDBGlobal = PhpDependency with name: "ilDB"  
ilExceptions = PhpDependency with name: "il*Exception*"
```

After presenting these rules to other people involved in the project, we decided to introduce other equivalent entity types with more suggestive names. This addition was functionally inconsequential but greatly improved the readability of the rules. The same entities could be re-defined as follows:

¹⁰<https://github.com/mamuz/PhpDependencyAnalysis>

```
eval = PhpFunction with name:"eval"  
ilDBGGlobal = PhpGlobal with name:"iLDB"  
ilExceptions = PhpClass with name:"iL*Exception*"
```

To implement this change, we had to add the introduced types to our list of reserved language keywords. The resolution algorithm implemented to retrieve the entities with the new types was the same as the one used for “*PhpDependency*”.

II. Javadoc-like rule comments: In **C3**, we soon realized that rules defined by a smaller group of users had poor chances of being fully understood by the community at large participating in the development of the project. Users reading the results produced by the analysis questioned the rationale hidden behind the rule, asked for examples of violations or for a reference to a more detailed source of information. As a consequence, we introduced a new language feature that allowed users to provide documentation for single rules in the form of comments. This feature was used to briefly summarize the intent of the rules, suggest possible fixing strategies and point users to more detailed reference pages hosted on the project website.

During our evaluation we were also asked to make results actionable. In fact, in **C1** and **C2**, we were able to detect cyclic or otherwise unwanted dependencies between packages but we neglected to provide useful information on how to remove them. In a second iteration we extended our analyzer and included a detailed description of all the concrete dependencies (*e.g.*, invocations, variable references) that actually caused the violation. This made it possible to quickly validate the results and to find the real source of the problem. To further improve the usefulness of our report, we also added an additional error message attribute to each single violation description. This new attribute contained a free form error message as returned by the employed analysis tool.

All this information improved the understanding of the problem. To further support the user in the resolution of the violation, we also introduced support for reporting optional suggestions on how to actually proceed in the task. In a parallel project, we developed a tool that not only detects cycles but also computes all possible refactoring strategies that could be adopted to break them (See chapter 5). We integrated this tool in Probō and successfully managed to provide detailed advice on how to concretely eliminate the detected cycles.

To implement this change, we had to extend our model representation and adapt the language parser.

6.5.4 Performance and Scalability

Our prototype has been designed to be highly scalable and proved to perform adequately even when confronted with large industrial projects.

To evaluate the performance of our tool, we measured the time required to complete the phases described in section 6.1 (“Transformation” and “Reporting” were combined as they both have minimal impact over the overall performance of the system). The results are presented in Table 6.3. Execution times for each phase were measured on a 2.6 GHz machine with 16 GB of memory and are expressed in seconds. The projects taken into consideration are JHotDraw 6.0b1 (JHD) and the projects from **C1** and **C2**. The analyzed projects are ordered by increasing size: JHotDraw has 28,000 NLOC; **C1** has 55,000 NLOC; **C2** has 460,000 NLOC.

#	Project / rules	Parsing (sec / % total)	Transf.+Re- porting (sec / %)	Analysis (sec / % total)	Total (sec)
1	JHD / 2	10.8 (46%)	1.1 (5%)	11.4 (49%)	23.4
2	JHD / 3	0	0.2 (17%)	1.43 (83%)	1.6
3	JHD / 4	10.7 (93%)	0.7 (6%)	0.1 (1%)	11.5
4	JHD / 2	0	0.3 (60%)	0.2 (40%)	0.6
5	CS1 / 3	9.8 (35%)	1.4 (5%)	17.0 (60%)	28.3
6	CS2 / 2	38.1 (100%)	0.3 (0%)	0 (0%)	382.0
7	CS2 / 67	38.1 (97%)	0.2 (0%)	10.7 (3%)	392.6

Table 6.3: Performance measurements for different phases of execution while analyzing seven rulesets on three projects.

The first phase, parsing, is typically the most costly one. In some cases (*i.e.*, cases 2 and 4 in Table 6.3) it is not needed, as the entities specified by the user do not need to be resolved to elements in the source code. This happens when the entities refer to concepts that are not in the code (*e.g.*, web resources) or the analysis tool does not need to resolve them explicitly (*e.g.*, the entity refers to the whole code base). Parsing time is strongly correlated with the size of the target system and its complexity (*e.g.*, coupling, depth of inheritance). This phase typically takes between 35% and 100% of the total execution time. The time required by this task also depends on the level of detail expected from the resulting information. If a more coarse-grained model is sufficient to support subsequent tasks, times could be reduced considerably.

Transformation+Reporting typically takes less than one second to be executed. The only exceptions are in cases 1 and 5. In both cases the excess in execution time is explained by the increase in the amount of information that is written to the report file (both rule sets contain a constraint on the presence of package cycles). Based on these results, we can conclude that the processing overhead of rules and results generated by external tools is minimal and not necessarily correlated with the number of user-specified rules.

Analysis is a phase that requires a highly variable amount of time to be completed. It strongly depends on the precision and inherent complexity of the task at hand. The choice of an efficient analysis strategy and of an adequate level of granularity in the result are crucial for making the tool usable. In **C3**, the largest project we analyzed so far, we spent a large amount of the time optimizing the analysis algorithm. We reduced its execution time from more than one hour down to three minutes. The optimization consisted mostly in introducing new caches, replacing the PHP interpreter (HHVM¹¹ instead of Zend Engine¹²) and refactoring output printing statements. This incredible improvement made the difference between a unacceptably slow solution and one that could be periodically run after each commit. Most of the applied optimizations are at the platform level and can be reused in other analysis contexts. The execution time of this phase is also correlated with the size of the input system and the number of rules that need to be checked. Its overall impact on the overall execution time varies from 0% to 83% (in case 2 the analyzer needed to measure the latency of a remote web resource).

In general, we can conclude that our approach is reasonably scalable and can be applied to large industrial applications. The validation of a realistic rule set against a project with half a million lines of code took 6.5 minutes (case 2). Smaller systems can be analyzed under a minute (all remaining cases).

6.5.5 Portability and Reusability

The tool can be adapted to support other languages. This can be done by adding a new parser for extracting the information necessary to resolve user-defined entities in the code. Analyzers are also typically language specific and therefore need to be modified to support the chosen technology. In **C3** we adapted our toolchain to support rules designed for PHP applications. The effort to do so was relatively modest and, due to the pipeline architecture, heavily localized. To fulfill the needs of other organizations, we plan to provide support for C and C++.

We also analyzed the reuse potential of the adapters developed thus far. We observed that the choice of rules in the three case studies is quite homogeneous (*e.g.*, dependency constraints are defined in all rule sets). Adapters developed to check these rules could be reused across organizations as far as the underlying technologies were the same (**C3** required a new adapter for checking dependencies within its PHP project).

6.5.6 Extensibility

Our solution was designed to be easily extensible. Support for new analysis tools can be added by developing relatively simple adapters that act as data transformers

¹¹<http://hhvm.com/>

¹²<http://php.net/>

between Probō and the chosen external tool. This mechanism allowed us to support most of the rules encountered in our case studies with little effort.

During our evaluation we also encountered some rules that could not be supported in a straightforward way. Some of these rules were simply too ambitious and would have involved the use of very specific and extensive analysis techniques that were not available in common off-the-shelf tools. These rules typically concerned behavioral aspects of the system, such as inter-process communication (every test case must use the local database for testing), execution time (test cases should be executed within a given time interval) and application state (test methods have to perform a rollback at the end of their execution). All these rules would require some form of instrumentation and the definition of a clearly defined application-specific testbed for exercising the properties of interest. This would probably result in a more complex execution pipeline and changes to the architecture of our tool. Upon discussion, we decided to limit the scope of our tool to rules that can be tested using non-invasive analysis techniques that do not require any upfront preparation or modification of the target system. We use static analysis to check rules that concern source code properties and on-demand query-based tools for evaluating other rules (e.g., latency, file structure).

#	Rule
X1	The method <code>init</code> should be called after creating or loading a <code>CompositeFigure</code> , that is, after calling the method <code>new</code> or <code>read</code> .
X2	Calls to the method <code>addInternalFrameListener</code> should occur before calling the method <code>addToDesktop</code> in the class <code>MDIDesktopPane</code> .
X3	The status line must be created (i.e. call to <code>setStatusLine</code>) before a tool is set (i.e. call to <code>setTool</code>).
X4	After calling <code>viewDestroying</code> on an object you cannot do anything else on that object (seen in class <code>ViewChangeListener</code>).
X5	If you call <code>activate</code> or <code>deactivate</code> from the class <code>Tool</code> you should call <code>isActive</code> before (seen in class <code>DrawApplication</code>).
X6	If method <code>mouseUp</code> of class <code>AbstractTool</code> is overridden, the last statement should be a super call.
X7	If method <code>mouseDown</code> of class <code>AbstractTool</code> is overridden, the first statement should be a super call.

Table 6.4: List of rules discovered during literature survey which can currently not be checked by Probō

Also some of the rules found during our survey would be hard to check using currently known analysis tools. For example, the rules X1-5 in Table 6.4 could theoretically be checked by extending one of the tools supported by our platform. Unfortunately this would require a more fine-grained parsing analysis than the one adopted at the moment. The parsing algorithm should take into account the order in which invocations occur within methods. Implementing this feature would require a major engineering effort, since it would impact multiple components of the analysis tool. A similar limitation was identified for rules that require details regarding the order in which statements appear within a method body (i.e., X6-7). To support such rules, one should again extend the analysis tool to also analyze the single statements

occurring in a source code file. The current implementation only considers coarse grained structural elements (such as classes, fields, methods) and ignores anything beyond that. This limitation is inherent to the current implementation of FAMIX [27], the meta-model used to represent the system under analysis.

6.6 Related Work

In this chapter we discuss a series of case studies that show how our approach (described in detail in chapter 4) can be applied in an industrial context.

Quality assurance tools: There exist various tools that can be employed to evaluate architectural conformance. Murphy *et al.* [96] introduced the idea of reflexion models, a verifiable representation of the logical dependencies expected to exist in a given target system. This technique has been widely exploited to build a consistent number of academic [31] and commercial tools (*e.g.*, Sonargraph, Structure101¹³, Semmle¹⁴). Some of these solutions offer additional complementary features. Some allow the definition of rules through a textual DSL [115, 48]. Others contributed new visual representations that support the reverse engineering of large systems [109].

All these techniques have been compared with respect to their functional capabilities in multiple studies [103, 63, 106]. As a result, we know that existing tools offer complementary features and none of them can be considered to subsume all the others. Pruijt *et al.* [106] also identify a set of conformance rules that are not supported by any of the analyzed tools (*e.g.*, naming conventions, subclass inheritance). Such rules could be checked using tools such as SOUL [93], uContracts [74], LogEN [32] or SCL [56]. Unfortunately such solutions are mostly proofs of concept and are rarely tested in a realistic industrial environment. These solutions, despite offering valuable support for the task they aim to support, suffer from several flaws on aspects that range from the usability of the specification to the scalability of the rule checking algorithm. Some commercial counterparts, .QL[25] and CQLinq¹⁵, have managed to address those aspects. Despite the availability of these solutions, dealing with the singularities of multiple tools is often considered as a significant inconvenience when dealing with quality assurance solutions. We propose an approach that hides the operational details of such tools behind a uniform and readable high-level DSL called Dictō.

Empirical evaluation: Other researchers have focused on the empirical foundations of existing techniques. Weinreich *et al.* [120] presents a case study in which rules are tested on a reverse engineered model of a banking system. Lozano *et al.* [74] present a collection of rules encountered while analyzing source code comments in JHotDraw. They also present a survey of structural relationship rules specified in

¹³<https://structure101.com>

¹⁴<https://semmle.com>

¹⁵<http://www.ndepend.com/docs/cqlinq-syntax>

previous literature. Passos *et al.* [103] validates existing static conformance checking tools by comparing their ability to test a given selection of rules. All the rule sets encountered in these studies are typically used by the authors as a baseline for validating an approach or a tool. In our work we tried to encode the reported constraints in Dictō as a means to test the expressivity of our language. The results are discussed in subsection 6.5.2.

Albuquerque *et al.* [2] evaluate the usability of their DSL borrowing techniques coming from the human-computer interaction domain. Their approach consists in comparing competing languages based on quantitative experiment. This strategy works well in case one is interested in artificially proving the superiority of one DSL towards another in terms of language features, but does not answer the question whether the language is capable of dealing with real world specification requirements. Since our main interest is in evaluating the capabilities of Dictō within an industrial context, we chose to adopt a more empirical approach.

Ganea *et al.* [40] evaluate their quality assurance tool by defining a non-comparative experiment involving industrial users. The subjects were asked to perform analysis tasks with and without the evaluated tool. The results show that tool-assisted users are more efficient at solving quality related tasks. In our work, we assume that this finding can be extended to any tool that provides contextual information regarding specific properties of a system.

6.7 Conclusion

In this chapter we show the effectiveness of our approach for monitoring architectural quality (introduced in chapter 4) through 3 case studies. Our results reveal that our approach can be applied in an industrial context. It is sufficiently usable to allow the definition of new rules even by untrained users. Results can be conveniently integrated into existing monitoring solutions (*e.g.*, dashboard) enabling short feedback loops that advance the understanding of the system and encourage proactive behavior. Scalability can be ensured by limiting the extent of the analysis required for checking rules. The language we designed for supporting specification of rules (Dictō) could be evolved to accommodate emerging requirements and successfully managed to fulfill the needs of our users. The limitations discovered during our study appear to be minor and will be possibly addressed in future iterations. Since our goal is to reduce the cost of architectural conformance checking, we value simplicity over completeness. In conclusion, we claim that the possibility of easily integrating the capabilities offered by existing analysis tools, and of specifying rules without the need of acquiring specific knowledge over the tool used for checking it, are two deciding factors that may well contribute to building an effective quality monitoring solution.

7

The Path to Industrial Adoption

In this chapter we aim to empirically assess the dynamics involved in choosing and adopting an automated conformance monitoring solution. We investigate the criteria that need to be taken into account when designing, adapting and/or deploying an automated quality monitoring solution within a company. We analyze a set of 14 interviews conducted with practitioners involved in architectural duties working in four distinct Swiss companies to derive a list of criteria that influence the decision of whether to adopt a conformance monitoring solution.

These criteria describe the different priorities that professionals take into account when evaluating the possibility of transitioning to a new quality assessment tool for conformance checking. Understanding which forces are involved and how to maximize one factor over another can make a difference between setting up a solution that everybody ignores and one that actually contributes to improving the quality of a system. To further understand how these criteria are valued in an industrial context, we planned 3 case studies. We developed our own prototype and tried to lead various professionals to adopt it within one of their projects.

Thanks to the acquired experience we identified five phases that might be encountered on the path towards successful adoption of a conformance checking solution. Each phase is related to multiple judgement criteria that, based on our experience, are worth taking into account in order to maximize the chances of higher acceptance and impact of the conformance monitoring solution. Different criteria may be considered to be more or less important depending on various socio-technical aspects that characterize the target organization.

7.1 Background

During our previous in-the field study (chapter 3), we observed that software architects are clearly aware of having little control over the implementation of their architecture. They typically express design decisions in terms of guidelines or specification constraints. These are supposed to be read and periodically checked by developers but this hardly happens. As a project grows, quality checks become less

frequent and more arbitrary. Knowing whether a certain architectural invariant is actually correctly reflected in the implementation often becomes a matter of trust.

To overcome this issue and properly assess the conformance of a software system with respect to a set of formerly defined architectural constraints, practitioners typically resort to specialized commercial tools (*e.g.*, SonarQube¹, SonarGraph²). Unfortunately the setup and maintenance costs related to such tools often outweigh the benefits that can be derived from their results. In fact, most of these tools suffer from three main limitations:

Poor maintainability: Tools are typically hard to set up and configure. Practitioners are often afraid of adopting new solutions because of the effort typically required for customization and integration. This effort is hard to estimate and to justify to higher management.

Narrow scope: The tools currently available on the market are very specialized and typically offer support for checking at most a couple of rule types. This means that practitioners are likely to need multiple tools, based on heterogeneous conceptual models and with low chances of integration.

Low usability: Introducing a new solution typically involves training dedicated personnel, and producing explanatory documentation. The specification of rules is typically a non-collaborative process and which requires specific technical knowledge.

In our study, 4 out of 14 participants said they used tools in the past and later on decided to switch to manual inspections due to the excessive costs involved in the process. 6 participants did not even attempt to set up an off-the-shelf automatic solution. The reasons behind these choices could often be related to missing support from key stakeholders, lack of availability of an adequate tool and limitations in formalizing and testing the desired invariant. Participants reported that the value of conformance checking is often underestimated by decision makers. The “cost of misalignment is not perceived” (A) and the “return of investment [of a conformance checking solution] is not clearly seen” (A). “Developers do not care about non-functional requirements” (E). “Automatic validation [of architectural constraints] would be useful and will eventually probably be implemented, but is not feasible at the moment” (J) since “it’s not a high priority and nobody would exactly know how to do it” (I). In other cases practitioners recognize that “there are things which can’t be verified and decided automatically” (B). Sometimes, “verification is not heavily used because most of the concerns cannot be formalized” (E). In those cases “it’s better to delegate to humans (*e.g.*, pair-programming, checklists)” (E). This

¹<http://www.sonarqube.org>

²<https://www.he1lo2morrow.com/products/sonargraph>

might be due to mismatches between a specific architecture and an existing tool (*e.g.*, F could not use JDepend because the tool did not allow him to identify components over package naming). Sometimes professionals recognize that “all their rules could be checked with static analysis” (J) and that “if a tool existed it would be very useful” (I). Unfortunately “the tools available today are not enough” (I) and nobody feels qualified to contribute a new one on his own.

After collecting a considerable number of opinions, we reached the conclusion that an automated conformance checking solution was typically desirable but considered to be too expensive to set in place. To solve this problem we started developing a solution that could at the same time address most of the identified requirements and minimize the drawbacks encountered by the interviewed practitioners. The prototype that we eventually developed reflected our understanding of this multitude of viewpoints and experiences. We developed an informal and intuitive model for describing the needs and expectations of our ideal user matured further as we started using the prototype in multiple case studies. During almost two years of collaboration, we started discovering new concerns, similar adoption patterns and unexpected requirements across different organizations. This led us to explicitly analyze and characterize all different factors that influence the way an automated conformance checking solution is evaluated by professional users. In the remainder of the chapter, we try to reconstruct and analyze the process that needs to be followed in order to establish a quality assessment solution within an industrial organization.

7.2 Study design

In this chapter we report on a blocked subject-project study [10]. Our goal is to observe different case studies and generalize the adoption process through common characteristics. To carry out our case studies we developed a prototypical solution which was refined in response to emerging requirements and feedback. The aim of this study is not to prove intrinsic properties of the tool, but rather classify the events that happen around its introduction process.

Before planning our case studies, we interviewed 14 professionals working on tasks related to software architecture (See chapter 3). We recorded approximately 18 hours of conversation and collected several project documents (*e.g.*, architecture specification, developer guidelines). The qualitative data collected in this preliminary phase were analyzed using coding techniques [94]. Our goal at this stage was to identify important factors that may have a role in deciding on the selection, adoption and maintenance of a quality assurance tool for monitoring architectural conformance. As a result, we identified most of the criteria described in section 7.3.

In a subsequent phase, we developed a prototypical tool for monitoring architectural conformance (see chapter 4).

Finally, we carried out five industrial case studies (see chapter 6 and Table 7.1) with the intension of introducing our tool in the context of a live development project.

In the first case study (*i.e.*, C1) we dealt with a consortium of vendors of an open-source PHP application called Ilias³, an e-learning platform used internationally by millions of users. In C2 and C3 we approached two teams working for a company with one of the largest IT divisions in Switzerland. The team in C2 counts more than 30 developers working full-time on a 10+ year migration project (from Cobol to J2EE) of a B2B application used for managing orders and coordinate traffic. C3 was a smaller team responsible for the development of a J2EE basic framework employed in almost all the hundreds of projects running in the company. In C4 we approached a medium-sized consulting company with public sector contracts. In C5 we tried to establish a collaboration with a branch of one of the largest Swiss insurance companies.

#	Organization domain (employees)	Project tech. - size (team)	Phase Reached
C1	E-learning (12 vendors)	PHP - 1 M (25)	5
C2	Transportation (1.000+)	J2EE - 0.5 M (30+)	4
C3	Transportation (1.000+)	J2EE - 50 K (5)	1
C4	E-government (1.000+)	J2EE - 50 K (5)	1
C5	Insurance (1.000+)	J2EE - n/a (n/a)	1

Table 7.1: List of project teams participating to our case studies.

The study was conducted over the span of almost one year. All the companies were using some sort of commercial quality assurance tool. In C1, the collaboration was carried out through a special interest group (SIG) responsible for the proposal and design of new architectural concepts that could improve the overall quality of the system.

The case studies reached various stages of maturity (See section 7.4). In the first case study (C1), the tool was fully deployed and integrated into the production environment. In the second, we got it installed on single workstations and have it used in isolation. In all other case studies, the tool was officially presented to the team but was never fully adopted. In C3, we introduced the tool by showing some violations that we knew were relevant and partially already checked by the team. The inability to continue to further stages highly depended on a general lack of trust and low motivation. In C4, we interacted directly with higher management and immediately gained interest and willingness to collaborate. Unfortunately the company was subsequently acquired and restructured by a larger company. Our agreement, which was in the process of being formally defined by the legal department of one of their clients, was never made official. In C5, we similarly got in touch with higher management and presented our tool. After the initial meeting we tried to propose a project, but we didn't hear back from them.

³<http://www.ilias.de>

7.3 Decision Factors

In this section we characterize the decision factors that play a decisive role in driving decisions when discussing over the adoption of a conformance monitoring solution. This classification is non-exhaustive and is entirely based on our direct experience. It includes factors that were discovered by analyzing 18 hours worth of interviews applying coding techniques (as described in section 7.2). Additional categories (*i.e.*, performance, accuracy, feature set, analytics support) were discovered during the case studies.

7.3.1 Product

Cost – As in almost any industry, cost is often a primary concern. Embracing a new quality assurance solution typically entails new licensing costs and often requires skilled labor for adapting and maintaining the acquired solution. “The automation of conformance tests is expensive” (interviewee A) and “budget resources allocated to quality related tasks is limited” (G) in most of the cases. “Automatic checking [of architecturally relevant constraints] would be a big advantage” (G), but is not always a “high priority” (J), given that the customer and non-technical management often think that “the only important thing is that the product is delivered with the right functionalities” (G). This leads to situations where architectural conformance is checked “manually” (G), through generic tools (*e.g.*, SonarQube (H)) or “is evaluated [on the client side] on the basis of produced specification documents” (A). In general, software quality “is hard to quantify and management is always skeptical because it concerns issues out of its domain”.

Usability – A usable software product should be easy to understand, learn and use [58]. “Software architects usually do not care about [implementation] details, they reason on a more abstract level” (K). They typically find themselves in the situation of taking decisions over architectural invariants which, in order to be checked, would require them to deal with variably complex tools. Some architects “prefer visual representations when it comes to understand architectural structure” (H). Some may even develop their own toolchain in order to check dependencies as specified in an easily maintainable Excel spreadsheet (D). This shows that non-technical declarative specifications are normally preferred over tool-specific configurations.

Performance – Software efficiency can be measured in terms of time and resources consumed to complete a given task [58]. Analysis execution time is an important feature of a quality assessment tool. If architectural invariants need to be checked in near real-time (*e.g.*, at commit-time, on request), performance becomes key in providing a usable experience.

Accuracy – This quality is ensured if the software product is able to provide results with the needed degree of precision [58]. Analysis results need to be as correct and complete as possible. In our studies, we spent a considerable amount of time in

validating the results of our tool. This process was crucial to increase the reliability of our solution and to proceed with our evaluation.

Feature set – Users select and compare software products according to the features they support [58]. In our case, the ability to describe and check multiple characteristics of a system was relevant to our collaborators. It influenced the requirements elicitation phase and was a clear discussion point when deciding on the adoption of the tool. Features are often used for comparison with other commercial products.

Integrability – Software products don't exist in isolation, but need to co-exist and interact with other, independently-developed products in the target environment [58]. Conformance checking solutions are typically introduced into established contexts and have to harmonize with pre-existing processes and tools. Practitioners expect that a new solution can non-intrusively enrich their experience by providing information when and where required. In fact, G advocated a conformance checking tool with "integration with Word or Enterprise Architect" while J said that "it would be nice to have rules checked by an IntelliJ plugin". Some organizations have some kind of periodic reporting mechanism already in place (*e.g.*, email reports generated during the nightly build (G)) and would appreciate if those could be extended instead of being replaced or replicated.

Proactiveness – To effectively enforce guidelines and guarantee architectural invariants, one needs to be proactively reminded of relevant anomalies and opportunities for improvement. "Constraints cannot be enforced if they are simply described in a document; they either need to be implemented in a framework or a verification tool must check them" (K). In fact, "people forget about rules or don't even know of their existence" (I). "An architectural specification is almost worthless" and "it's important to have continuous feedback (*e.g.*, checks integrated in the continuous integration server)" (E). Developers need to be reminded of their mistakes and tools should support them to prevent accidental violations.

7.3.2 Process

Transparency – Architects and developers are keen to have an accurate and up-to-date overview of particular aspects of their system. More transparency over quality related issues leads to "easier risk assessment and reduced hidden costs" (N). For example, "detecting a dependency violation early during development [...] would be very helpful" (K). Delivering convenient reporting reduces the risk of incurring technical debt and "exposes conceptual fallacies" (K).

Analytics support – Rules and analysis results may have managerial value (*e.g.*, estimate relevance/effort of tasks, validate new design concepts). In our studies we observed that violations can be used to show progress over maintenance or migration processes. This was achieved by describing the desired target architecture in the form of constraints and considering the number of deriving violations as a metric for estimating the time required for completing the transition towards the defined

goal. Similarly, rule specifications can be used as a means to assess the feasibility and effort involved in realizing complex design changes.

7.3.3 User

Engagement – Another important factor that contributes to the success of a conformance checking solution is its level of acceptance. As we have seen, developers need to be properly motivated and encouraged to contribute. If they don’t feel sufficiently involved in the process of defining and following common objectives, the solution is likely to be soon ignored and eventually abandoned.

7.4 Adoption phases

In this section we describe the phases that we encountered while introducing an actual prototypical solution in various IT companies. Each phase is linked to one or more decision factors that must be taken into consideration during that particular phase of the process. The single phases and the associated decision factors are described in Figure 7.1 and in the subsequent sections.

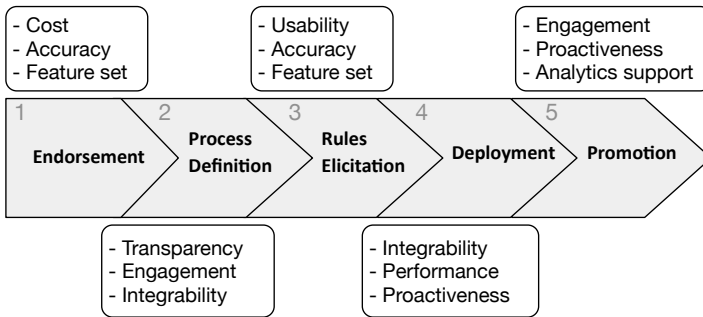


Figure 7.1: Adoption of an automated conformance monitoring solution: process phases and influencing decision factors.

7.4.1 Endorsement

Introducing a new technical solution within a company requires support from a competent and motivated person who understands the value of the proposed tool and the implications that this has on a chosen project. We typically approached this phase by setting up an introductory meeting with our contacts during which we presented the main features of our solution. The presentation also included some

exemplifying rules, and an outlook over possible integration options with currently employed monitoring solutions.

In our experience we tried to approach different kind of users: developers and architects. In C1, our first case study, we interacted with a special interest group (SIG) consisting of 18 members founded to discuss architecturally relevant refactorings. This community was clearly aware of the benefits that can be derived from extra-functional maintenance activities. They have a distinctively proactive mindset, as their group has set the goal to propose new tasks aimed at improving the overall quality of the system. In this context we could easily convince them of the advantages of our prototype. Most of the discussion that followed concerned the possible political implication that the introduction of a new tool would have had on the organization.

In C2 we also interacted with a quality-aware proactive senior developer who believed in the benefits of quality assurance tools. He recognized the limits of the current monitoring environment and was willing to experiment with other solutions. In this case, the major concern was regarding the capabilities and the accuracy of the proposed solution. As a developer, he was very interested in uncovering new existing flaws and inconsistencies. Since his role did not entitle him to take any organizational decision, we had to follow a long procedure to define a pilot project. During this procedure, we had to negotiate the terms of our collaboration with the legal department of the company and subsequently discussed the project with the leader of the team our contact person was part of.

In C3 we interacted with the heads of a team responsible for a smaller, yet very strategic, project run within a larger organization. These 4 people maintained a framework that was used as a foundation by most of the 200 projects developed in the same company. Unfortunately, they were less inclined to consider the adoption of a new quality monitoring solution. We analyzed their code base and showed them new violations that they were not capable of finding with their current toolset. Despite that, they dismissed the idea of refactoring the uncovered flaws and preferred to maintain their code base in its current state. Their attitude towards code quality was more reactive. If somebody reported a major architectural violation, they would have looked into the problem and discussed a solution. Introducing an automated tool that supported this task was in contradiction with their approach. The effort invested in preventive maintenance had to be kept to the minimum.

In C4, we had the chance to discuss the adoption of the solution with two company-wide branch managers responsible for all major architectural decisions. Having the opportunity to discuss the subject with technically competent decision makers clearly facilitated our task. We could easily convince them of the utility of our prototype and we could quickly define a pilot project for testing it out. Getting in touch with these persons was less complicated because of the limited size of the company (100-150 employees). The advantage of encountering less resistance during the first encounter was lessened by the fact that both stakeholders were often very busy and tended to schedule subsequent meetings at longer time intervals (compared to the

other case studies).

Relevant Decisional Factors

Cost: One of the primary concerns when discussing the adoption of the new technical solution was its cost. In our experience, especially when dealing with people having little or no decision power, we encountered appreciation for our choice of relying on open-source analyzers. On the other hand, the interest seemed to decrease when discussing analysis features that were offered by equivalent commercial tools already in use in the company. In C3, the architects were already using a rather expensive tool for checking dependency constraints. The possibility of integrating the results produced by the existing solution with the information produced by our tool seemed to be less appealing since money had already been invested in the competing solution. Also in C2 developers were using a commercial solution for checking dependencies. In that case, the person responsible for maintaining and operating the tool clearly admitted that the costs related to the use of that solution (*i.e.*, licensing, training) were clearly not proportional to the benefits offered. Given the tight budget typically allocated for quality analysis related tasks, it is important to define a price which is reasonably contained and proportional to the amount of distinguishing features offered.

Accuracy: The results produced by an automated quality monitoring solution should be sufficiently precise in order to be considered useful. In C2 and C3 we discovered that our tool found between 400% to 500% more violations than the competing tool currently in use within the project (C2: our tool found 7 illegal logical dependencies, SonarGraph only 2; C3: our tool found 18 dependency cycles, SonarQube only 3). This difference in accuracy is due to different analysis strategies employed by the different tools. In C2, accuracy played a significant role in deciding on the utility of the proposed solution. In C3, accuracy was overshadowed by the cost of setting up and integrating a new tool, justifying its existence to management and investing effort into dealing with the detected violations.

Feature Set: One of the differentiation factors that distinguishes our prototypical solution from competing alternatives is the support for a wider variety of constraint types. The possibility of specifying rules concerning multiple design facets in the same specification was a key deciding factor in C1. During the first meeting, participants immediately started to think about the type of invariants that could be useful to check in their project. Several people even proposed new types of rules that were not described in the initial presentation. Learning about the extensibility of our approach and the option of designing new custom analyses with a relatively modest effort was clearly one of the main arguments that convinced them to invest in the solution. In C5, one of the software architects participating in the initial meeting asked about the possibility of defining and checking cross-project invariants. Also

in this case we can see how functionality is an important factor when discussing the high-level characteristics of a conformance monitoring solution.

7.4.2 Process Definition

If an organization agrees on supporting the introduction of the proposed solution, one must define how this can be done in concrete terms. In our case studies, we typically discussed various aspects. We analyzed how the solution was supposed to be integrated with the current infrastructure as well as the way future stakeholder would have to interact with it. Technical aspects are easily outlined and should be quickly sorted out by the service provider. Changes to the process require more careful analysis, since they may have a deeper impact on the performance of the organization. If a tool heavily interacts with current procedures and does not provide obvious advantages to its users, this tool will soon be neglected. Our solution provides contextual information that exposes anomalies in the developed code. By delivering our information through an existing information medium, we reduce the chance of altering established procedures and minimize the training costs.

In C2, we were asked to implement rules for expressing constraints described in an internal documental repository (*i.e.*, wiki website). All developers were asked at one point in time to read them, but few of them managed to keep them in mind and to periodically check whether they were updated. To reduce the overhead caused by a potentially useless activity, we decided to combine the existing documentation with executable rules that could be used to check the consistency of the developed system. This would have required them to delegate the task of defining new rules to the author of the guidelines. Rules written by this person would have, where necessary, required the intervention of a technical facilitator trained to integrate third party analyzers to support the checking of the defined rules. This technical facilitator would have initially been assisted by the original author of the tool. At the end of an initial training phase, the facilitator should have been capable of dealing with common integration scenarios. More complex cases, requiring deeper changes in the evaluation process, would have still required the knowledge of the original developers. The users ultimately inspecting the violations resulting from the validation of the rules, were expected to autonomously understand and react to the reported anomalies. Results were expected to be displayed through a pre-existing dashboard and handled through an integrated ticket system (See Section 7.4.3).

In C1, we discussed the target process together with the participants of the refactoring SIG. These people warned us of the risk of developing a solution that could not be fully accepted inside the community. In the past, another user deployed a continuous integration server that periodically built the core module of the project. This service was largely ignored because it was badly advertised to the community. Our solution had the potential of supporting vital quality assurance tasks but needed to be promoted in a convincing way. The tool had thus to be silently deployed and revealed as a complementary feature of a new, yet to be set up, continuous integration

server. The plan consisted in assigning the responsibility of deciding on new rules to the SIG. New rules would have been announced during a bi-weekly physical meeting that involved representatives of the major vendors involved in the community. People participating in this meeting should be gradually sensitized towards non-functional issues and should have the right to veto the proposed rules. The discussion of rules should not require too much time, as the meeting is mostly designed to propose and discuss over functional requirements. Users should eventually be encouraged to look into the current violations by reporting the results produced by our tool. The details of each violation can be found in the continuous integration web front-end.

Relevant Decisional Factors

Transparency: In C1, users were positively inclined towards policies that supported and encouraged transparency. The fact of being part of an open community with a flat hierarchy made them more prone to engage in public assessment activities. Despite this, we decided to report on the introduction of new violations upon commit sending only private emails to the developers responsible for their introduction. During the bi-weekly meeting it was also expected that only the positive interventions (*i.e.*, violation removal) were mentioned publicly. Transparency is a good principle if counterbalanced by respect for the dignity of the involved people. In C2, developers were divided in 2 categories: internal (*i.e.*, developers contracted by the company) and external (*i.e.*, consultants hired through a third-party company). External developers can be easily dismissed and care more about maintaining their reputation. Exposing flaws that could be associated to them was immediately considered as a threat to their position. Based on this consideration, we decided to first test our solution within a smaller group of users exclusively composed of developers. This strategy would have helped to avoid unnecessary tensions and to gradually change the attitude of the team towards quality related issues.

Participation: In C2, we had the chance to interact with the developer responsible for maintaining a previously established dependency checking tool. This person emphasized that providing prompt feedback to the users is a key feature of any quality assessment tool. This must be true for users interested in the results of the analysis as much as for users interested in maintaining the rules checked by the used tool. In our case we aimed at integrating our tool with a dashboard system that was regularly refreshed after each build of the project. Rules defined for our tool would be defined through a dedicated web editor that supported the ability to interactively check their applicability to the target system. New tickets and email notifications would be created after the introduction of new violations. Pre-configurable ticket prioritization may also be used to direct the attention of the user. In C1, participation was seen as a key ingredient of a successful service. To reach a sufficient number of developers, we aimed at maximizing transparency and introducing new incentives

for rewarding active developers. Users would be automatically listed in a leaderboard where the most contributors are ranked based on the number of violations they have removed. Another strategy to promote the involvement of developers consisted in presenting only the latest introduced violations while consulting the online report displayed within the continuous integration service. This reduces the chance that the user may feel overwhelmed and makes the effort required to eliminate the violations more easy to estimate. Remaining violations can still be browsed by expanding the view.

Integrability: Quality assessment tools are typically expected to be naturally streamlined into the process. In C2, the previously adopted dependency checking tool offered support for integrating the results into SonarQube. This feature was fundamental for making the violations visible to the developers. Separately produced reports cannot be regularly inspected without a clear incentive. SonarQube not only provides current measurements of the system but is also used to manage change tasks. This last feature makes it an essential platform of communication, that happens to be visited frequently by all the users involved in the development process. Our choice of creating new tickets for newly introduced violations reduces even more the distance of our solution from the center of attention of our target users. In C1, we chose to integrate our results inside TeamCity. This service already reports failed unit tests. By adding an additional view that shows the results produced by our tool, we emphasize the duality of a software system by displaying non-functional violations side-by-side with functional failures. Also in this case, having all information seamlessly integrated behind one coherent interface is likely to reduce resistance to adoption and increases productivity. The possibility to also link violations to impacted code elements also increases the convenience of the solution.

7.4.3 Rules Elicitation

Architects and developers have a very personal understanding of what constraints regulate the architecture of their system. This interpretation is typically vague and might not always fit the framework proposed by quality assessment tools. To bridge the gap that separates tools from design ideas, we first started by designing a language that could be used to replicate actually observed specification patterns. Putting language before functionality increased our chances of acceptance and played an important role in reducing the barriers to usage of our prototype.

The language that we used in our case studies, Dictō, is based on a simple model (Figure 7.2). Dictō can be used to describe properties and relations on and between entities defined in a system. Rules can be expressed in different modes (*e.g.*, Test-Methods *must* have annotation '@Test' or *only* TestMethods *can* have annotation '@Test'). The DSL resulting from this model proved to be effective in eliminating any distraction related to unimportant technicalities linked to the the tools used for the analysis. The language was generic and abstract enough to accommodate a large

variety of concrete rules. This encouraged our collaborators to explore very different requirements and led us to iteratively adapt and refine the language to meet their requests.

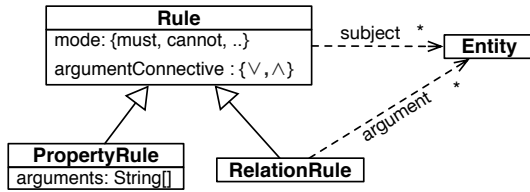


Figure 7.2: Language meta-model

In both case studies, rules were initially defined based on examples provided for illustration purposes. This helped us to gain familiarity with the general concepts governing our language. Later on we encouraged them to engage in the definition of more experimental rules. We suggested to examine current developer guidelines and to formalize rules that were (or that could have been) specified in such a context.

In C2, the user formalized a set of dependency constraints that were currently checked on his project through another tool (*i.e.*, SonarGraph). The specification of these constraints was previously regarded as technically demanding and was managed by one single developer, trained for the task. In its new translated form, the rules were finally under the control of our collaborator. This helped him to get a better understanding of the current architecture and to extend the current set of invariants with new rules concerning the module he was working on.

To enable the analysis of other documented guidelines and constraints, we were required to extend our tool. As our collaborators realized that unsupported requirements could be easily integrated in the solution, they started to discuss additional ideas and to propose constraints that diverged more and more from the initially supported feature set. The proposal of unusual and unpractical rules represented a clear achievement, as it meant that our users could fully concentrate on what they wanted to check without considering how this could be done. In other words, our DSL has been effectively used as a design tool instead of just being considered as a pure specification tool. Relying on a declarative language that can be easily understood by humans as well as tools, reduces the cost of formalizing and communicating knowledge and favors discussion.

In C1, other stakeholders started to propose rules related to their area of expertise or to recent tasks. Some managed to formulate syntactically correct rules by observing previous specifications. Others provided more prosaic descriptions. But eventually everybody could understand the rules written by others without any particular assistance. One of the points that contributed to making this possible was the introduction of inline Javadoc-like documentation in rule specifications. Thanks to

this beneficial outcome, the refactoring SIG started to discuss new refactoring ideas together with enforcement policies. New rules were presented to superior decision organs and could be fully discussed without any previous instruction.

In both case studies we established short feedback loops during which we iteratively defined, analyzed and corrected each respective project ruleset. In each iteration we put care into manually validating the violations reported as a result of our rule analysis. By doing so, in C2, we discovered several infractions that were acknowledged as concrete issues and were ignored by other tools currently employed by the team (e.g., SonarQube, SonarGraph). This increased the trust in our tool and reduced the distance between our solution and other comparable commercial products. In C1, the analysis of some reported violations brought the discovery of a previously unknown design anti-pattern (known as courier anti-pattern⁴). This in turn led to a new refactoring initiative and consequent rules that guarantee its implementation. In this case we observed how our solution could actually concretely support a feedback loop model for continuous quality improvement.

Relevant Decisional Factors

Usability: This was clearly one of the most deciding factors. In C1, stakeholders indirectly involved in the experiment were able to autonomously define new rules (e.g., *IliasCodebase cannot depend on eval*; *IliasCodebase cannot depend on SuppressErrors*). The addition of documentation comments in the specification (used for describing the semantics of rules) reduced the gap between ordinary informal documentation and the more formal syntax of our language. Rules could be easily discussed without the presence of the original author. The overall friendliness of the language was partially undermined by the absence of comprehensive documentation and a proper rule evaluation environment (i.e., sandbox environment). In C2, team members carefully avoided maintaining the rules specified for SonarGraph because of the poor usability of its configuration language. This shortcoming limited participation to a much smaller and less representative group of stakeholders.

Feature set: The capabilities of our prototype were soon questioned when users started to carefully consider their requirements. Questions like “is it possible to define..” or “how can I check if..” started to appear in C1 during physical and virtual discussions as soon as more users were involved in the process. Other users suggested rules (e.g., “IliasCodeBase must be compatible with PHP5.3”) that were syntactically correct but could hardly be checked using any kind of tool currently available on the market. Negotiating the scope of the offered service is a clear responsibility of the solution provider and will help him to focus the requirements elicitation process.

⁴<https://r.je/oop-courier-anti-pattern.html>

Accuracy: In C2 and C3, we defined rules concerning the presence of cycles and dependencies in the project. In both cases, our analysis reported violations that were completely ignored by popular commercial tools used by the organization (e.g., SonarQube, SonarGraph). These tools have a good reputation and a solid user base but, without a means to establish the accuracy of their evaluation, it is hard to question the value of their application. Offering some reference measurements and generally assessing the accuracy of the produced results will increase trust in the proposed solution.

7.4.4 Deployment

Once defined, rules need to be verified automatically on a regular basis. To enable this behavior, we have to integrate our solution with an existing quality control system (e.g., continuous integration server, dashboard). The challenges posed by this process may vary depending on the technical environment, the current development process and the general attitude of the team towards automated feedback mechanisms. In our case studies we opted for a simple setup (see Figure 7.3).

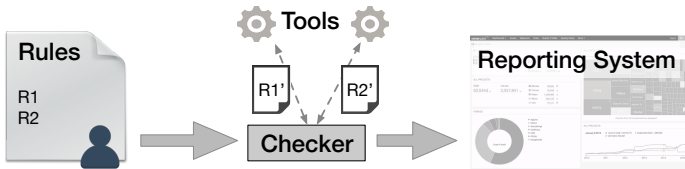


Figure 7.3: Automated conformance checking solution

In short, we analyze the rules defined during the previous phase through a set of pre-adapted off-the-shelf analyzers. The results are then integrated into a visual reporting system (e.g., dashboard).

In C1, our collaborators decided to display the results as a custom view within TeamCity⁵, a continuous integration system. Through this approach they succeed in addressing both functional (i.e., unit test) and non-functional (i.e., architecture conformance rules) aspects within a single interface. This allowed also for skeptical users, who should at least care about functional tests, to be exposed to architectural issues. In our implementation, we put care in not overwhelming the user by only showing the violations introduced in the latest build. All violations, including those introduced in the past, are still available for inspection by switching to a secondary view. This choice was mostly due to the fact that developers may be more interested in fixing issues introduced by recently contributed changes. Users might also be more motivated to address the issues if they see a smaller number of work items. An additional summary report was produced every two weeks and served as basis for

⁵<https://www.jetbrains.com/teamcity/>

discussion during the periodic physical meetings organized by all the organizations participating to the community.

In C2, the team used SonarQube as main issue tracker. We initially aimed at extending the current installation by automatically adding our violations as new issues. This strategy would have allowed for a very smooth transition with no changes in the workflow and full integration of our analysis. Unfortunately all SonarQube installations available in the organization stem from the same customized implementation. This made it almost impossible, given the size of the company and the intricacies of their governance, to extend the reach of our experiment to all the members of the team involved in the case study. As an alternative we decided to deploy our solution locally on one of the workstations used in the project. As a result we could simulate the productive environment and observe how our solution would have been used in the originally planned scenario.

Relevant Decisional Factors

Process Integration: One of the developers working in C2 and assigned to the maintenance of SonarGraph said that “integration is key”. He himself had to integrate the results produced by his tool into SonarQube before and recognized that without that step nobody would have cared about the violations it reported. Extending a familiar and consistently used reporting instrument is essential to reach out to the end-user instead of requiring him to change his practice. In C1, we introduced our results by enhancing a long needed continuous integration application. The appeal of this new application granted sufficient support for our solution.

Performance: The time needed to complete our analysis played an important role during the deployment phase. In C1, we spent considerable effort to bring the time needed to check our rules from hours to minutes. To make this possible we had to optimize our prototype and improve some analysis techniques. Without this engineering effort, the solution would have been useless since developers expect feedback shortly after they commit. Also in C2, we had to deal with a large code base. This meant that multiple iterations were just dedicated to reduce analysis time. In both cases we had to make compromises over the precision of the analysis. This meant discussing with our collaborators over the minimum amount of information needed to describe violations in such a way that they could lead to a concrete action plan.

Proactiveness: A developer in C1 emphasized that that “immediate feedback is important”. There is evidence [64] that providing timely feedback on potentially degrading quality issues can significantly contribute to preventing architectural decay. In C2 we sent email to contributors after each commit to expose violations that she introduced in her last change. Once again we tried to reduce the burden on the user by reaching out to her.

7.4.5 Promotion

One of the main focuses of our solution, as stated by our collaborators, should be to actively contribute to improving code quality and to communicate the value of the implied effort to management. In C1 the reactions towards our tool were initially rather conservative, since the tool had no commercial history and was not believed to be reliable. By involving our collaborators in all the phases of the study and openly discussing possible improvements and limitations we slowly gained their trust. We gladly observed that the representatives of the various sub-communities could be involved in the process of discussing new rules. In fact, our tool-agnostic specification could be used to provide a concrete insight into the activities of the SIG as well as enabling management to take decisions on the concrete aspects that defined them. Our DSL became an effective tool for negotiating work items, expressing new concerns and assessing completed tasks. The support so far provided by management, was essential to establish a wide and legitimated dialogue. The fact that this assembly usually prioritizes the discussion over functional features and typically finds place under tight time constraints, does not guarantee that the discussion will continue to be kept at regular intervals. We assume that backing from a motivated sub-community (*i.e.*, SIG) needs to continue to be provided.

After observing the beneficial contribution of our solution, other members of the community volunteered to deploy our solution in their own environment. In fact, beside the main branch, accessed by the whole community, there were other client specific customizations maintained by single vendors. One of those vendors is currently working towards integrating our solution in his own continuous integration server (*i.e.*, Jenkins⁶).

Relevant Decisional Factors

Engagement: In C1, we carried out a survey and found that developers were positively impressed by our solution. They observed that “it definitely leverages the discussion about architecture and separation of concerns”. They commented that rules were readable and meaningful for the project. To further incentivize them, we also decided to create a leaderboard⁷ that tracks who removed the most violations from the code base. This simple expedient, already exploited in other communities (*e.g.*, StackOverflow⁸), helped us increasing curiosity towards our analysis. One contributor said: “I searched very deliberately for violations within our modules and fixed them, on one hand to get our modules violation-free and on the other hand the leader board influenced me”.

⁶<https://jenkins-ci.org>

⁷<http://ci.ilias.de/DictoStats>

⁸<http://stackoverflow.com>

Proactiveness: In C1, every developer that we managed to contact through meetings or surveys stated that he is strongly in favor of architectural checks. Yet, when it comes to proactive actions, most people do not seem to have the time to work on quality related issues. This led us to focus on the proactive elements of our integration with the continuous integration server. We paid attention to setting up passive mechanisms that required minimal or no effort from the user. Analysis results are generated and communicated automatically and results are easily reachable in a context where the user would already be looking if she is interested in reports related to quality. Also the rule definition process should support proactive thinking. In C1, rules are defined to guarantee the correct implementation of new design ideas. By continuously inspecting and discussing the violations resulting from the analysis of those rules, developers grew their understanding of the system and even discovered new architectural anomalies that needed to be addressed. This unexpected virtuous circle once again confirmed the role of a well engineered conformance solution in high-level design related discussions.

Analytics Support: As previously mentioned, reports produced by our tool were also used to show progress over ongoing development tasks. Violations could show how far the current implementation was compared to a specific target architectural design resolution. This helped at the same time to strengthen the sense of control over non-functional aspects of the system as well as increasing the transparency of the development process.

7.5 Discussion

Our case studies help to gain a practical insight into how an architectural conformance solution can be introduced in an industrial context. We ran several case studies and report on the strategies employed to gain full adoption of our tool. Despite our best efforts, we did not manage to cover all the phases outlined in our model (section 7.4) for every case study. We recognize that our study required a significant commitment by both involved parties and that most industrial organizations are skeptical towards innovative ideas. This explains why most case studies ended during the first phase. At the beginning it is important to gain the support of an active community or a relevant decision maker, and this did not happen in some of those cases. In general, we observed that political tensions that pre-existed within the organization had the biggest influence over the success of our project. Different social aspects (like fear of being discredited, of exposing inconvenient truths, of contradicting a superior or losing credibility in front of the colleagues) played an important role over the decisions that were taken along the way.

Despite the partial completeness of some case studies, we still believe that our experience helped us gain a deeper understanding into the process undergone while introducing a generic architecture conformance monitoring solution. Our case studies show that the users involved in the adoption process might have different priorities

but normally share the same concerns. The decision factors described in section 7.3 were partially identified before the beginning of the study and proved to be important points in our decision making process. Some of the decisions we took had observable effects which could be later on evaluated and discussed. The usability of our prototypical language, for example, appeared to be a relevant discussion point during the elicitation phase and showed its beneficial effects towards the end of the study, when more users started to participate to the definition of rules. Similarly, the integration of our solution within the existing infrastructure was considered relevant as we had to define the overall process required to sustain the solution and appeared to be a crucial aspect in later phases of the study (according to surveys and usage statistics). The decision factors that we analyzed in this chapter should be considered for general guidance when defining a plausible adoption strategy. The analyzed criteria described in our work won't necessarily help in reaching a successful outcome but should contribute in reinforcing the assumptions that one might have towards the general process.

The phases described in section 7.4 and elaborated more in detail in the following sections have been inferred directly from our experience. We compared the different case studies and tried to factor our common activities and processes. As an outcome we obtained a sequence of replicable phases that can be used to break down our case studies. The resulting process model should be sufficiently general to be recognized in almost any context that entails the introduction of a new system for technical support. The main purpose of the model is to create a link between easily observable phases and the deciding factors identified to answer RQ1. The fact that those phases could be used to describe the case studies encountered in our experience provides empirical evidence that those phases can be used to establish a successful quality monitoring solution within an industrial organization.

7.6 Related Work

Several authors report on the application of tools for checking architecture conformance in an industrial context.

Rosik *et al.* [107] describes an industrial application of a technique based on reflexion models [96]. The authors conclude that developers value their solution positively but violations are not fixed in a timely manner. Our studies were considerably more complex (*i.e.*, more developers, legacy code, more type of rules) and partially confirmed the observations reported by Rosik *et al.*. We observed that violations are typically resolved as long that the effort involved is contained and adequate incentives are provided. Herold *et al.* [53] elaborate on the technical details of a rule-based architecture conformance checking tool used in an industrial case study. The approach is conceptually similar to ours but restricted to a particular type of rule (*i.e.*, dependency constraints). The studies show the importance of process integration and the use of a simple and user-friendly formalism to increase maintainability.

Ganea *et al.* [40] evaluate their quality assurance tool by defining a non-comparative experiment involving industrial users. The authors recognize the importance of reducing the number of false positives, seamless integration, unobtrusive feedback, performance and user feedback. All conclusions are drawn from a controlled experiment, but still provide a comprehensive picture of what factors may influence the acceptance of a quality assessment tool. In general all mentioned authors tend to focus mostly on describing their solution instead of analyzing the process followed to introduce it within a specific organization.

Other studies explicitly analyze the impact of introducing a quality assurance tool in an industrial context. Sadowski *et al.* [108] describe a program analysis platform that integrates multiple lint tools and exports all detected issues to a review system. The authors emphasize the importance of actionable results and workflow integrability. Users are particularly sensitive to false positives and value the ability to share their configuration with the other members of their team. Despite the slightly different nature of the tools discussed in their study, Sadowski *et al.* reach similar conclusions as those reported in this chapter. Johnson *et al.* [61] interview 20 practitioners that use static analysis tools on a regular basis. The conclusions drawn from this study are similar to those reported by Sadowski *et al.*. Participants criticize the poor usability of the tools (*e.g.*, result navigation, result understandability, settings sharing, customizability) and stress the importance of quick feedback.

7.7 Conclusion

In this chapter we describe how an automated conformance monitoring solution can be adopted in the context of an industrial project. We describe this process in terms of its composing phases and the deciding factors that influence its course. Our aim is to offer a comprehensive overview of the forces involved in such a delicate course of events.

Our experience shows that a quality assurance solution should be above all customizable and usable. Developers are typically not encouraged to react to quality-related issues and need to be properly informed and motivated. This can be achieved through continuous automated analysis, non-overwhelming reporting and various types of incentives (*e.g.*, reputation points). Architectural inconsistencies need to be communicated as small and easily manageable tasks. Resistance to change can be overcome by integrating the new solution with a pre-established quality control system.

Through our experience we hope to provide guidance to professionals and academics who intend to introduce a similar solution in a company.

8

Conclusions

In this last chapter we summarize the contributions made by this dissertation and point to directions for future work.

8.1 Contributions of this Dissertation

In this thesis we argue that usability and extensibility are crucial aspects of any architecture conformance checking solution. We state that a cost-effective solution should hinge on a unified approach based on an extensible, declarative and empirically-grounded specification language.

We presented Dictō, a DSL for defining architectural rules, and Probō, a rule checking engine designed for extension. By using a semi-formal practice-inspired declarative language, we are able to write tool-agnostic rules that are accessible to untrained stakeholders and, at the same time, can be automatically processed by a conformance checking validator. Dictō is designed to seamlessly adapt to new extensions of the backend. The language can be specialized to the requirements of a specific project without compromising its usability. Our approach has proven to be applicable in the context of various industrial projects.

Our contributions are the following:

1. We surveyed existing approaches (chapter 2) and performed an extensive on-the-field study (chapter 3) to understand the main opportunities, challenges and obstacles faced by practitioners when performing architecture conformance checking.
2. We presented a novel approach to architecture conformance checking (chapter 4). The approach was empirically validated by performing various industrial case studies (chapter 6, chapter 7).

3. As part of our approach, we iteratively designed a language for describing architectural constraints (chapter 4). We analyzed different usability trade-offs and took into consideration practitioner's feedback (chapter 6).
4. We presented a novel approach for enriching the results of a conformance checking analysis by providing refactoring hints for guiding the user in the removal of dependency cycles (chapter 5).
5. We analyzed the social and technical implications of introducing a new architecture conformance checking solution in the context of an industrial project (chapter 6, chapter 7).

8.2 Future Research Directions

Having defined our approach to architecture conformance checking and discussed various opportunities to reduce the overall cost of this quality assurance activity, we identify scope of further work in this area.

Simplified Specification Process: During our on-field study, several professionals expressed interest at having a solution that seamlessly integrated with the current environment used to describe and document software architecture. With a dedicated plugin, Dictō rules could be directly embedded in a Word document or specified in a separate view of an IDE. Developers might also be interested in specifying rules in the code (using specially annotated comments). Investigating and comparing different options might lead to a better understanding of practitioner's needs and expectations.

Reducing the Feedback Loop: Rules written in Dictō can be validated through Probō in a single batch execution step. The solution is designed to be run offline and periodically. To increase the effectiveness of our approach, one could imagine having a real-time solution that notifies the user of new violations at the same moment in which they are introduced. Results could be displayed directly in the IDE and could point directly to the artifacts affected by the detected inconsistency.

Automatic Rule/Fix Mining: In our approach, rules are defined by a user who has a good understanding of the architecture of the system and is capable of formulating rules that reflect the implied constraints. These rules could be also automatically mined from existing software corpora and suggested to the user on the basis of some previous technical choices (*e.g.*, adopted framework). Rules (and relative fixes) could be detected by looking at homogenous set of files (*e.g.*, Xml configuration files, classes with a special annotation) and searching for regularities and outliers. By analyzing a sufficiently large amount of data, we could infer new rules and, by taking into account historical data, we could extract previously applied fix patterns.

Relevant Rules: At the beginning of our work, we performed various studies to find out which rules are normally considered relevant for professionals. While developing and evaluating our prototype we encoded some of these rules and discovered others over time. As of today, we still lack a comprehensive overview of which rules software designers need to enforce on a system’s implementation. Further empirical studies are needed to investigate current requirements and provide solid ground for the development of future analysis solutions.

Logic Programming: Our approach is related to logic programming in general and modal logic in particular. By closely analyzing the similarities between Dictō and existing logic formalisms, one could study alternative ways for expressing rules or exploit existing inference engines for validation. By reimplementing our solution based on a more formal foundation, we could benefit of additional guarantees regarding internal consistency (*e.g.*, absence of logical contradiction) and correctness of the results (tools developed in this field are mature and reliable).

Model Driven Development: Dictō is a declarative textual specification language for encoding architectural rules. In some domains (*e.g.*, automotive, embedded systems), professionals use similar languages for describing the design of their system. Languages like AADL are used as input for MDD (model driven development) processes to define the main architectural features of embedded systems. Studying such languages and their platforms could lead to a better understanding of the requirements commonly encountered by practitioners and the challenges related to their implementation.

8.3 Summary

We propose a novel approach to architecture conformance checking. By providing an extensible, declarative and empirically-grounded specification language to the user and augmenting the analysis results with operationally relevant recommendations, we can reduce the overall cost of the process. This result is achieved by relieving the user from automatable tasks and increasing the overall usability of the task.

Appendices



Qualitative Study - Interview Questions

The quantitative study, described in chapter 3, was based on the questions reported in this appendix.

- Briefing:
 - Describe a recent noteworthy project;
 - What was your role?;
 - Who designed the architecture?;
 - Which process was adopted?
- Architectural requirements:
 - Which were the most important aspects in the architecture and why?;
 - Were they expressed as quality requirements (QR)?
- QR specification:
 - How was architecture documented?;
 - How were QR specified?;
 - What was the specification used for?;
 - Was the specification maintained?
- QR validation:
 - Which QR were tested and how?;
 - When are they validated?;
 - Based on which criteria did you decide what to test?;
 - Which tools did you use?;
 - Who defined and maintained the tests?

B

Quantitative Study - Survey Questions

In this chapter we include some fragments of the survey submitted to the participants of the quantitative study described in chapter 3. The full survey is available in the form of screenshots on our website¹.

Survey on Quality Requirements

In the following survey you will be asked to estimate the relevance of several *quality requirements*. At each step you will be presented with:

- a **quality concern**: an interest which pertains to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders.
- some examples of **quality requirements** related to the specified quality concern.
- **questions** related to the quality concern.

The evaluation is subjective and should be based on your personal professional experience. Before expressing your evaluation, please make sure you carefully read and understand the examples given for each quality concern. All information obtained from this study will be kept anonymous.

Motivation

The architecture of a software system is the result of a number of design decisions that aim at satisfying a given set of **quality requirements**, such as ease of evolution, good run-time performance, and rapid build times. Architecture is rarely explicit in code and alignment between intended and actual architecture is often very difficult to validate and requires consistent human effort. There have been several attempts to address this problem (e.g. component dependencies: JDepend, Sotograph, Structure 101; performance: JMeter) but there are still many aspects that remain ignored. Our goal is to develop new ways for **expressing and validating quality requirements**. We envision a set of tools that help identifying implementation solutions that break specified architectural requirements. This questionnaire will help us to identify the most relevant and commonly specified aspects of an architecture. This will provide a solid starting point for our research project.

Thanks for your collaboration.

Andrea Caracciolo
caracciolo (at) iam.unibe.ch

Start Survey

Figure B.1: Landing page

¹<http://scg.unibe.ch/download/AC/Survey.pdf>

Naming conventions

examples of requirement related to this concern

- Component Interfaces end with the suffix '_CI'.
- Java bean classes end with the suffix 'Bean'.
- File names must conform to the following pattern: [DATE]_[SOME-ID].txt

questions (REQUIRED)

a. FAMILIARITY
Have you ever seen this kind of requirement in a project ?


Yes
 No

b. IMPACT ON PROJECT SUCCESS
How important was it to satisfy this kind of requirement ?

1
 2
 3
 4
 5

No impact on the project Low impact Moderate impact High impact Critical to project success

questions (OPTIONAL)

c. FORMALISM
How was the requirement specified ? 

<input type="checkbox"/> natural language	<input type="checkbox"/> common formal notation: <input style="width: 100px;" type="text"/>
<input type="checkbox"/> semi-formal notation: <input style="width: 100px;" type="text"/>	<input type="checkbox"/> tool-specific notation: <input style="width: 100px;" type="text"/>
<input type="checkbox"/> ad-hoc formal notation	<input type="checkbox"/> other: <input style="width: 100px;" type="text"/>

d. VERIFICATION
How was the requirement validated during development ?

<input type="checkbox"/> no verification	<input type="checkbox"/> whitebox test
<input type="checkbox"/> code review	<input type="checkbox"/> testing tool: <input style="width: 100px;" type="text"/>
<input type="checkbox"/> blackbox test	<input type="checkbox"/> other: <input style="width: 100px;" type="text"/>

e. COMMENT
Do you want to share any comment or personal example for this kind of requirement ?

Back

Next

Figure B.2: Constraint evaluation page



Qualitative Study - Taxonomy examples

The taxonomy, described in section 3.2, consists of a set of quality attributes. Here we provide exemplary constraints for each identified quality attribute.

- Authorization: Access to the service is restricted to users having role A or being part of group B.
- Meta-annotations: Attributes of type DateTime must be annotated with @Date(format = "d-m-Y").
- Response time: The system is time-critical and has to answer each request within 10 ms.
- Authentication: The user has to confirm his identity using the central authentication service.
- Code metrics: Code-coverage for unit tests must be > 85
- Dependencies: Component A cannot invoke method X of component B.
- Signature: A web service must provide the following API: push(Message), pull():Result.
- Communication: Communication must be synchronous. Protocol used must be HTTPS.
- Software update: New security updates must be installed within 1 week from their release.
- Data retention policy: Only the last 4 digits of a credit card number can be stored.
- Availability: The system must be reachable 99% of the times from 6h00 to 20h00.
- Data integrity: Instance values of datatype IBAN must start with an ISO 3166-1 country code.
- Data structure: pom.xml contains: <dependency><groupId>junit</groupId>..</dependency>

- Event handling: Exceptions of type MyEx must be handled in the layer where they were generated.
- File location: web.xml must be located at src/main/webapp/WEB-INF/
- Hardware infrastructure: The server must have 48GB RAM and 2 x 2.6GHz Intel Xeon processor.
- Throughput: The system must be able to execute a certain task 10'000 times per hour.
- Visual design: The web front-end must comply to the standard corporate design guidelines.
- Naming conventions: Java bean classes end with the suffix "Bean".
- Recoverability: The system has to be operational again within 1h after a crash.
- Software infrastructure: Components communicate with each other using CORBA.
- Accessibility: The web front-end must be accessible to color blind users.
- System behavior: The system state has to be consistent with a given state machine diagram.

Bibliography

- [1] B. Aga. Marea — a tool for breaking dependency cycles between packages. Masters thesis, University of Bern, Jan. 2015. URL <http://scg.unibe.ch/archive/masters/Aga15a.pdf>.
- [2] D. Albuquerque, B. Cafeo, A. Garcia, S. Barbosa, S. Abrahao, and A. Ribeiro. Quantifying usability of domain-specific languages: An empirical study on software maintenance. *Journal of Systems and Software*, 101:245–259, 2015. ISSN 0164-1212. doi: 10.1016/j.jss.2014.11.051. URL <http://www.sciencedirect.com/science/article/pii/S0164121214002799>.
- [3] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implementation. In *ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, Orlando, FL, USA, 2002. ACM. ISBN 1-58113-472-X. doi: 10.1145/581339.581365.
- [4] R. Allen and D. Garlan. The Wright architectural specification language. CMU-CS-96-TB, School of Computer Science, Carnegie Mellon University, Pittsburgh, Sept. 1996. URL <http://www.cs.cmu.edu/afs/cs/project/able/ftp/wright-tr.ps>.
- [5] D. Ameller, C. Ayala, J. Cabot, and X. Franch. How do software architects consider non-functional requirements: An exploratory study. In *Requirements Engineering Conference (RE), 2012 20th IEEE International*, pages 41–50, Sept. 2012. doi: 10.1109/RE.2012.6345838.
- [6] N. Ayewah and W. Pugh. A report on a survey and study of static analysis users. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, DEFECTS '08, pages 1–5, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-051-7. doi: 10.1145/1390817.1390819. URL <http://doi.acm.org/10.1145/1390817.1390819>.
- [7] N. Ayewah, D. Hovemeyer, J. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *Software, IEEE*, 25(5):22–29, Sept. 2008. ISSN 0740-7459. doi: 10.1109/MS.2008.130.
- [8] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.
- [9] P. Binns, M. Engelhart, M. Jackson, and S. Vestal. Domain-specific software architectures for guidance, navigation, and control. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):201–227, 1996. doi: 10.1142/S0218194096000107.

- [10] B. Boehm, H. D. Rombach, and M. V. Zelkowitz, editors. *Foundations of Empirical Software Engineering*. Springer-Verlag, Berlin, Germany, 2005. ISBN 3-540-24547-2.
- [11] B. Bokowski. Coffeestrainer: Statically-checked constraints on the definition and use of types in java. *SIGSOFT Softw. Eng. Notes*, 24(6):355–374, Oct. 1999. ISSN 0163-5948. doi: 10.1145/318774.319253. URL <http://doi.acm.org/10.1145/318774.319253>.
- [12] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999. ISBN 0-201-57168-4.
- [13] J. Bosch. Software architecture: The next step. In F. Oquendo, B. Warboys, and R. Morrison, editors, *Software Architecture*, volume 3047 of *Lecture Notes in Computer Science*, pages 194–199. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-22000-8. doi: 10.1007/978-3-540-24769-2_14. URL http://dx.doi.org/10.1007/978-3-540-24769-2_14.
- [14] N. Boucké, A. Garcia, and T. Holvoet. Composing structural views in xADL. In *Proceedings of the 10th international conference on Early aspects: current challenges and future directions*, pages 115–138, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-76810-6, 978-3-540-76810-4. URL <http://dl.acm.org/citation.cfm?id=1783274.1783284>.
- [15] J. Brunet, G. Murphy, D. Serey, and J. Figueiredo. Five years of software architecture checking: A case study of eclipse. *Software, IEEE*, 32(5):30–36, Sept. 2015. ISSN 0740-7459. doi: 10.1109/MS.2014.106.
- [16] A. Caracciolo, M. Lungu, and O. Nierstrasz. How do software architects specify and validate quality requirements? In *European Conference on Software Architecture (ECSA)*, volume 8627 of *Lecture Notes in Computer Science*, pages 374–389. Springer Berlin Heidelberg, Aug. 2014. doi: 10.1007/978-3-319-09970-5_32. URL <http://scg.unibe.ch/archive/papers/Cara14a-SpecifyValidateQualityRequirements.pdf>.
- [17] A. Caracciolo, M. Lungu, and O. Nierstrasz. Dicto: A unified DSL for testing architectural rules. In *Proceedings of the 2014 European Conference on Software Architecture Workshops, ECSAW '14*, pages 21:1–21:4, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2778-7. doi: 10.1145/2642803.2642824. URL <http://scg.unibe.ch/archive/papers/Cara14b-Dicto.pdf>.
- [18] A. Caracciolo, M. Lungu, and O. Nierstrasz. Dicto: Keeping software architecture under control. *ERCIM News*, 99, Oct. 2014. URL <http://ercim-news.ercim.eu/en99/special/dicto-keeping-software-architecture-under-control>.
- [19] A. Caracciolo, M. Lungu, and O. Nierstrasz. A unified approach to architecture conformance checking. In *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 41–50. ACM Press, May

2015. doi: 10.1109/WICSA.2015.11. URL <http://scg.unibe.ch/archive/papers/Cara15b.pdf>.
- [20] A. Caracciolo, B. Aga, M. Lungu, and O. Nierstrasz. Marea: a semi-automatic decision support system for breaking dependency cycles. to appear, Mar. 2016. URL <http://scg.unibe.ch/archive/papers/Cara16b.pdf>.
- [21] A. Caracciolo, M. Lungu, O. Truffer, K. Levitin, and O. Nierstrasz. Evaluating an architecture conformance monitoring solution. to appear, 2016. URL <http://scg.unibe.ch/archive/papers/Cara16c.pdf>.
- [22] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2002.
- [23] J. W. Creswell and Vicki. *Designing and Conducting Mixed Methods Research*. Sage Publications, Inc, 1 edition, Aug. 2006. ISBN 9781412927925. URL <http://www.worldcat.org/isbn/1412927927>.
- [24] E. Dashofy, A. van der Hoek, and R. Taylor. A highly-extensible, xml-based architecture description language. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 103–112, Aug. 2001. doi: 10.1109/WICSA.2001.948416.
- [25] O. de Moor, D. Sereni, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, and J. Tibble. .ql: Object-oriented queries made easy. In R. Lammel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 78–133. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-88642-6. doi: 10.1007/978-3-540-88643-3_3. URL http://dx.doi.org/10.1007/978-3-540-88643-3_3.
- [26] L. de Silva and D. Balasubramaniam. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1):132–151, Jan. 2012. ISSN 0164-1212. doi: 10.1016/j.jss.2011.07.036. URL <http://dx.doi.org/10.1016/j.jss.2011.07.036>.
- [27] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.
- [28] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002. ISBN 1-55860-639-4. URL <http://scg.unibe.ch/download/oorp>.
- [29] A. Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000. doi: 10.1145/352029.352035. URL <http://homepages.cwi.nl/~arie/papers/dslbib.pdf>.

- [30] C. K. Duby, S. Meyers, and S. P. Reiss. Ccel: A metalanguage for c++. Technical report, Providence, RI, USA, 1992.
- [31] S. Duszynski, J. Knodel, and M. Lindvall. Save: Software architecture visualization and evaluation. In *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, pages 323–324, 2009. doi: 10.1109/CSMR.2009.52.
- [32] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th international conference on Software engineering, ICSE '08*, pages 391–400, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368142. URL <http://doi.acm.org/10.1145/1368088.1368142>.
- [33] J.-R. Falleri, S. Denier, J. Laval, P. Vismara, and S. Ducasse. Efficient retrieval and ranking of undesired package cycles in large software systems. In J. Bishop and A. Vallecillo, editors, *Objects, Models, Components, Patterns*, volume 6705 of *Lecture Notes in Computer Science*, pages 260–275. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-21951-1. doi: 10.1007/978-3-642-21952-8_19. URL http://dx.doi.org/10.1007/978-3-642-21952-8_19.
- [34] P. Feiler, D. Gluch, J. Hudak, and B. Lewis. Embedded systems architecture analysis using SAE AADL. Technical Report CMU/SEI-2004-TN-005, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2004. URL <http://www.sei.cmu.edu/library/abstracts/reports/04tn005.cfm>.
- [35] P. Feiler, D. Gluch, and K. Woodham. Case study: Model-based analysis of the mission data system reference architecture. Technical Report CMU/SEI-2010-TR-003, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2010. URL <http://www.sei.cmu.edu/library/abstracts/reports/10tr003.cfm>.
- [36] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical report, DTIC Document, 2006.
- [37] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000. URL <ftp://mirror.transact.net.au/sourceforge/w/project/wa/waspap/waspap/Core/REST-Architecture.pdf>. AAI9980887.
- [38] Forrester Research. The value and importance of code reviews. Technical report, Klocwork, Mar. 2010. URL <http://www.klocwork.com/resources/research/the-value-and-importance-of-code-review-forrester>.
- [39] M. Fowler. Reducing coupling. *IEEE Software*, 18(4):102–104, 2001. ISSN 0740-7459. doi: 10.1109/MS.2001.936226.

- [40] G. Ganea, I. Verebi, and R. Marinescu. Continuous quality assessment with incode. *Science of Computer Programming*, 2015. ISSN 0167-6423. doi: 10.1016/j.scico.2015.02.007. URL <http://www.sciencedirect.com/science/article/pii/S0167642315000520>.
- [41] D. Garlan. An introduction to the Aesop system. Technical report, Carnegie Mellon University, July 1995.
- [42] D. Garlan, R. T. Monroe, and D. Wile. ACME: An architecture description interchange language. In *CASCON'97: Proceedings of the 7th Conference of the Centre for Advanced Studies on Collaborative Research*, pages 169–183, Toronto, Ontario, Canada, Nov. 1997. doi: 10.1145/782010.782017.
- [43] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 3, pages 47–67. Cambridge University Press, New York, NY, USA, 2000.
- [44] E. Gasparis, J. Nicholson, and A. Eden. Lepus3: An object-oriented design description language. In G. Stapleton, J. Howse, and J. Lee, editors, *Diagrammatic Representation and Inference*, volume 5223 of *Lecture Notes in Computer Science*, pages 364–367. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-87729-5. doi: 10.1007/978-3-540-87730-1_37. URL http://dx.doi.org/10.1007/978-3-540-87730-1_37.
- [45] S. Giesecke, W. Hasselbring, and M. Riebisch. Classifying architectural constraints as a basis for software quality assessment. *Advanced Engineering Informatics*, 21(2):169 – 179, 2007. ISSN 1474-0346. doi: 10.1016/j.aei.2006.11.002. URL <http://www.sciencedirect.com/science/article/pii/S1474034606006681>. Ontology of Systems and Software Engineering; Techniques to Support Collaborative Engineering Environments.
- [46] G. Gousios, A. Zaidman, M.-A. Storey, and A. van Deursen. Work practices and challenges in pull-based development: The integrator’s perspective. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 358–368, May 2015. doi: 10.1109/ICSE.2015.55.
- [47] O. M. Group. Unified Modeling Language (version 1.3). Technical report, Object Management Group, 1999.
- [48] A. Gurgel, I. Macia, A. Garcia, A. von Staa, M. Mezini, M. Eichberg, and R. Mitschke. Blending and reusing rules for architectural degradation prevention. In *Proceedings of the 13th International Conference on Modularity, MODULARITY '14*, pages 61–72, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2772-5. doi: 10.1145/2577080.2577087. URL <http://doi.acm.org/10.1145/2577080.2577087>.

- [49] M. Haigh. Software quality, non-functional software requirements and IT-business alignment. *Software Quality Control*, 18(3):361–385, Sept. 2010. ISSN 0963-9314. doi: 10.1007/s11219-010-9098-3. URL <http://dx.doi.org/10.1007/s11219-010-9098-3>.
- [50] J. S. Hammond, N. Yuhanna, M. Gilpin, and D. D’silva. Market overview: Enterprise data modeling: A steady state market prepares to enter a transformational new phase, Oct. 2008.
- [51] M. Hanus. Multi-paradigm declarative languages. In V. Dahl and I. Niemela, editors, *Logic Programming*, volume 4670 of *Lecture Notes in Computer Science*, pages 45–75. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-74608-9. doi: 10.1007/978-3-540-74610-2_5. URL http://dx.doi.org/10.1007/978-3-540-74610-2_5.
- [52] E. Hautus. Improving Java software through package structure analysis. In *International Conference Software Engineering and Applications*, 2002.
- [53] S. Herold, M. Mair, A. Rausch, and I. Schindler. Checking conformance with reference architectures: A case study. In *Enterprise Distributed Object Computing Conference (EDOC), 2013 17th IEEE International*, pages 71–80, Sept. 2013. doi: 10.1109/EDOC.2013.17.
- [54] R. Hilliard and T. Rice. Expressiveness in architecture description languages. In *Proceedings of the Third International Workshop on Software Architecture, ISAW ’98*, pages 65–68, New York, NY, USA, 1998. ACM. ISBN 1-58113-081-3. doi: 10.1145/288408.288425. URL <http://doi.acm.org/10.1145/288408.288425>.
- [55] M. Hitz and B. Montazeri. Measure coupling and cohesion in object-oriented systems. *Proceedings of International Symposium on Applied Corporate Computing (ISAAC ’95)*, Oct. 1995.
- [56] D. Hou and H. Hoover. Using scl to specify and check design intent in source code. *Software Engineering, IEEE Transactions on*, 32(6):404–423, June 2006. ISSN 0098-5589. doi: 10.1109/TSE.2006.60.
- [57] ISO/IEC. ISO/IEC 25010 — Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models, 2010.
- [58] ISO/IEC 9126-1. ISO/IEC 9126-1:2001 Software engineering – Product quality, 2001.
- [59] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*, pages 109–120, Nov. 2005. doi: 10.1109/WICSA.2005.61.

- [60] C. Jaspan, I.-C. Chen, and A. Sharma. Understanding the value of program analysis tools. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion, OOPSLA '07*, pages 963–970, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-865-7. doi: 10.1145/1297846.1297964. URL <http://doi.acm.org/10.1145/1297846.1297964>.
- [61] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, pages 672–681. IEEE Press, 2013. ISBN 978-1-4673-3076-3. URL <http://dl.acm.org/citation.cfm?id=2486788.2486877>.
- [62] P. Klint, T. van der Storm, and J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM '09. Ninth IEEE International Working Conference on*, pages 168–177, 2009. doi: 10.1109/SCAM.2009.28. URL <http://homepages.cwi.nl/~jurgenv/papers/SCAM-2009.pdf>.
- [63] J. Knodel and D. Popescu. A comparison of static architecture compliance checking approaches. In *Software Architecture, 2007. WICSA '07. The Working IEEE/IFIP Conference on*, page 12, Jan. 2007. doi: 10.1109/WICSA.2007.1.
- [64] J. Knodel, D. Muthig, and D. Rost. Constructive architecture compliance checking – an experiment on support by live feedback. In *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM 2008)*, pages 287–296, 2008.
- [65] J. Kramer. Whither software architecture? (keynote). In *ICSE*, page 963, 2012.
- [66] P. Kruchten, P. Lago, and H. van Vliet. Building up and reasoning about architectural knowledge. In C. Hofmeister, I. Crnkovic, and R. Reussner, editors, *Quality of Software Architectures*, volume 4214 of *Lecture Notes in Computer Science*, pages 43–58. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-48819-4. doi: 10.1007/11921998_8. URL http://dx.doi.org/10.1007/11921998_8.
- [67] P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, Nov. 1995. ISSN 0740-7459. doi: 10.1109/52.469759.
- [68] J. Lakos. *Large Scale C++ Software Design*. Addison Wesley, 1996. ISBN 0-201-63362-0.
- [69] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues. Ocarina : An environment for aadl models analysis and automatic code generation for high integrity applications. In F. Kordon and Y. Kermarrec, editors, *Reliable Software Technologies – Ada-Europe 2009*, volume 5570 of *Lecture Notes in Computer Science*, pages 237–250. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-01923-4. doi: 10.1007/978-3-642-01924-1_17. URL http://dx.doi.org/10.1007/978-3-642-01924-1_17.

- [70] J. Laval, S. Denier, and S. Ducasse. Identifying cycle causes with cycletable. In *FAMOOSr 2009: 3rd Workshop on FAMIX and MOOSE in Software Reengineering*, Brest, France, 2009.
- [71] J. Laval, S. Denier, S. Ducasse, and J.-R. Falleri. Supporting simultaneous versions for software evolution assessment. *Journal of Science of Computer Programming (SCP)*, May 2010.
- [72] J. Laval, S. Ducasse, and N. Anquetil. Ozone: Package layered structure identification in presence of cycles. In *9th Belgian-Netherlands software eVOLution seminar (BENEVOL 2010)*, Lille, France, 2010.
- [73] M. M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, Berlin, 1996. Springer.
- [74] A. Lozano, K. Mens, and A. Kellens. Usage contracts: Offering immediate feedback on violations of structural source-code regularities. *Science of Computer Programming*, 105:73 – 91, 2015. ISSN 0167-6423. doi: 10.1016/j.scico.2015.01.004. URL <http://www.sciencedirect.com/science/article/pii/S016764231500012X>.
- [75] D. C. Luckham, J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.
- [76] J. Magee and J. Kramer. Dynamic structure in software architectures. In *SIGSOFT'96: Proceedings of the 4th Symposium on Foundations of software engineering*, pages 3–14, San Francisco, CA, USA, 1996. ACM. doi: 10.1145/239098.239104.
- [77] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *ESEC'95: Proceedings of the 5th European Software Engineering Conference*, volume 989 of LNCS, pages 137–153, Sitges, Spain, Sept. 1995. Springer-Verlag. doi: 10.1007/3-540-60406-5_12.
- [78] M. Mair and S. Herold. Towards extensive software architecture erosion repairs. In K. Drira, editor, *Software Architecture*, volume 7957 of *Lecture Notes in Computer Science*, pages 299–306. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-39030-2. doi: 10.1007/978-3-642-39031-9_25. URL http://dx.doi.org/10.1007/978-3-642-39031-9_25.
- [79] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang. What industry needs from architectural languages: A survey. *Software Engineering, IEEE Transactions on*, 39(6):869–891, 2013. ISSN 0098-5589. doi: 10.1109/TSE.2012.74.
- [80] R. Marinescu and G. Ganea. inCode.Rules: An agile approach for defining and checking architectural constraints. In *Intelligent Computer Communication and Processing (ICCP), 2010 IEEE International Conference on*, pages 305–312, Aug. 2010. doi: 10.1109/ICCP.2010.5606420.

- [81] R. C. Martin. *Granularity*, 1996. URL <http://www.objectmentor.com/resources/articles/granularity.pdf>. www.objectmentor.com.
- [82] R. C. Martin. *Design principles and design patterns*, 2000. URL http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf. www.objectmentor.com.
- [83] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003. ISBN 0135974445.
- [84] A. Mattsson, B. Fitzgerald, B. Lundell, and B. Lings. An approach for modeling architectural design rules in uml and its application to embedded software. *ACM Trans. Softw. Eng. Methodol.*, 21(2):10:1–10:29, Mar. 2012. ISSN 1049-331X. doi: 10.1145/2089116.2089120. URL <http://doi.acm.org/10.1145/2089116.2089120>.
- [85] N. Medvidovic, E. M. Dashofy, and R. N. Taylor. Moving architectural description from under the technology lamppost. *Information and Software Technology*, 49(1):12–31, 2007. doi: 10.1016/j.infsof.2006.08.006.
- [86] S. J. Mellor and M. J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley Professional, May 2002. ISBN 0201748045. URL <http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0201748045>.
- [87] H. Melton and E. Tempero. Identifying refactoring opportunities by identifying dependency cycles. In *Proceedings of the 29th Australasian Computer Science Conference - Volume 48, ACSC '06*, pages 35–41, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc. ISBN 1-920682-30-9.
- [88] H. Melton and E. Tempero. The crss metric for package design quality. In *ACSC '07: Proceedings of the Australian Computer Science Conference, 2007*.
- [89] H. Melton and E. Tempero. An empirical study of cycles among classes in java. *Empirical Software Engineering*, 12(4):389–415, 2007. ISSN 1382-3256. doi: 10.1007/s10664-006-9033-1.
- [90] H. Melton and E. Tempero. Jooj: Real-time support for avoiding cyclic dependencies. In G. Dobbie, editor, *Thirtieth Australasian Computer Science Conference (ACSC2007)*, volume 62 of *CRPIT*, pages 87–95, Ballarat Australia, 2007. ACS.
- [91] K. Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 2000. URL ftp://prog.vub.ac.be/tech_report/2000/vub-prog-phd-00-02.pdf.
- [92] K. Mens and A. Kellens. IntensiVE, a toolsuite for documenting and checking structural source-code regularities. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 10 pp. –248, mar 2006. doi: 10.1109/CSMR.2006.29. URL <http://soft.vub.ac.be/Publications/2006/vub-prog-tr-06-01.pdf>.

- [93] K. Mens, R. Wuyts, and T. D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS-Europe 99*, pages 33–45, June 1999. URL <http://scg.unibe.ch/archive/papers/Mens99a.pdf>.
- [94] M. B. Miles and M. Huberman. *Qualitative Data Analysis: An Expanded Sourcebook(2nd Edition)*. Sage Publications, Inc, 2nd edition, 1994. ISBN 0803955405.
- [95] M. Moriconi and R. A. Riemenschneider. Introduction to SADL 1.0: A language for specifying software architecture hierarchies. Sri-csl-97-01, SRI International, 1997.
- [96] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT '95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.
- [97] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering, SIGSOFT '95*, pages 18–28, New York, NY, USA, 1995. ACM. ISBN 0-89791-716-2. doi: 10.1145/222124.222136. URL <http://doi.acm.org/10.1145/222124.222136>.
- [98] M. G. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, and P. Balachandran. Making defect-finding tools work for you. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE '10*, pages 99–108, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1810295.1810310. URL <http://doi.acm.org/10.1145/1810295.1810310>.
- [99] O. Nierstrasz, S. Ducasse, and T. Girba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York, NY, USA, Sept. 2005. ACM Press. ISBN 1-59593-014-0. doi: 10.1145/1095430.1081707. URL <http://scg.unibe.ch/archive/papers/Nier05cStoryOfMoose.pdf>. Invited paper.
- [100] OCL. Object constraint language specification, version 2.0, 2006. URL <http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf>. <http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf>.
- [101] T. Oyetoyan, D. Cruzes, and R. Conradi. Can refactoring cyclic dependent components reduce defect-proneness? In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 420–423, Sept. 2013. doi: 10.1109/ICSM.2013.62.
- [102] T. D. Oyetoyan, D. S. Cruzes, and C. Thurmman-Nielsen. A decision support system to refactor class cycles. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 231–240. IEEE, 2015.

- [103] L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. Mendonca. Static architecture-conformance checking: An illustrative overview. *Software, IEEE*, 27(5):82–89, Sept. 2010. ISSN 0740-7459. doi: 10.1109/MS.2009.117.
- [104] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, Oct. 1992. URL <http://www.bell-labs.com/user/dep/work/papers/swa-sen.ps>.
- [105] E. Poort, N. Martens, I. Weerd, and H. Vliet. How architects see non-functional requirements: Beware of modifiability. In B. Regnell and D. Damian, editors, *Requirements Engineering: Foundation for Software Quality*, volume 7195 of *Lecture Notes in Computer Science*, pages 37–51. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-28713-8. doi: 10.1007/978-3-642-28714-5_4. URL http://dx.doi.org/10.1007/978-3-642-28714-5_4.
- [106] L. Pruijt, C. Koppe, and S. Brinkkemper. Architecture compliance checking of semantically rich modular architectures: A comparative study of tool support. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 220–229, Sept. 2013. doi: 10.1109/ICSM.2013.33.
- [107] J. Rosik, A. Le Gear, J. Buckley, and M. Ali Babar. An industrial case study of architecture conformance. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '08*, pages 80–89, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-971-5. doi: 10.1145/1414004.1414019. URL <http://doi.acm.org/10.1145/1414004.1414019>.
- [108] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15*, pages 598–608, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL <http://dl.acm.org/citation.cfm?id=2818754.2818828>.
- [109] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proceedings of OOPSLA'05*, pages 167–176, 2005.
- [110] S. Shah, J. Dietrich, and C. McCartin. Making smart moves to untangle programs. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 359–364, Mar. 2012. doi: 10.1109/CSMR.2012.44.
- [111] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, Apr. 1995. doi: 10.1109/32.385970.
- [112] R. Svensson, T. Gorschek, B. Regnell, R. Torkar, A. Shahrokni, and R. Feldt. Quality requirements in industrial practice — an extended interview study at eleven companies. *Software Engineering, IEEE Transactions on*, 38(4):923–935, 2012. ISSN 0098-5589. doi: 10.1109/TSE.2011.47.

- [113] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, pages 146–160, 1972. doi: 10.1137/0201010.
- [114] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, New York, NY, USA, Jan. 2009. ISBN 978-0470167748.
- [115] R. Terra and M. T. Valente. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 39(12): 1073–1094, Aug. 2009. ISSN 0038-0644. doi: 10.1002/spe.v39:12. URL <http://dx.doi.org/10.1002/spe.v39:12>.
- [116] C. Tibermacine, R. Fleurquin, and S. Sadou. A family of languages for architecture constraint specification. *Journal of Systems and Software*, 83(5):815 – 831, 2010. ISSN 0164-1212. doi: 10.1016/j.jss.2009.11.736. URL <http://www.sciencedirect.com/science/article/pii/S016412120900315X>.
- [117] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *Software Engineering, IEEE Transactions on*, 35(3):347–367, May 2009. ISSN 0098-5589. doi: 10.1109/TSE.2009.1.
- [118] N. Ubayashi, J. Nomura, and T. Tamai. Archface: A contract place where architectural design and code meet together. In *ICSE'10: Proceedings of the 32nd International Conference on Software Engineering*, pages 75–84, Cape Town, South Africa, 2010. ACM. doi: 10.1145/1806799.1806815.
- [119] M. Voelter. Architecture as language: A story. *InfoQ*, Feb. 2008.
- [120] R. Weinreich and G. Buchgeher. Automatic reference architecture conformance checking for soa-based software systems. In *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*, pages 95–104, Apr. 2014. doi: 10.1109/WICSA.2014.22.
- [121] E. Woods and R. Hilliard. Architecture description languages in practice session report. In *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*, pages 243–246, 2005. doi: 10.1109/WICSA.2005.15.