

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Master Course in Computing Engineering

Diego de Lara e Albuquerque

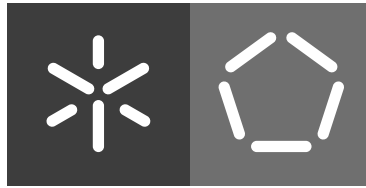
Automatic detection of architectural violations in evolutionary systems

Master dissertation

Supervised by: António Nestor Ribeiro

Alessandro Fabricio Garcia

Braga, May 22, 2014



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Master Course in Computing Engineering

Diego de Lara e Albuquerque

Automatic detection of architectural violations in evolutionary systems

Master dissertation

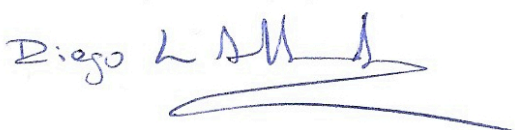
Supervised by: António Nestor Ribeiro

Alessandro Fabricio Garcia

Braga, May 22, 2014

É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE.

Universidade do Minho, 11/07/2014

Assinatura: 

Acknowledgements

My gratitude goes to Professor António Ribeiro for his guidance, patience and trust; to Professor Alessandro Garcia for the excellent support, for all the opportunities he gave me during my MSc and to trust in me. This project was only possible because of you.

I want to thank all the members of OPUS Research group, for the countless hours of both silly and serious discussions, and the great work environment they create. It was a pleasure to work alongside all of you.

Thanks to my lovely girlfriend, for all her love, caring and patience to listen to my complains throughout this year.

Thanks to my family and friends, for their love, caring and support, throughout my entire life.

Resumo

Aplicações de software evoluem ao longo dos anos a um custo: a sua arquitetura modular tende a ficar degradada. Isso acontece principalmente porque a manutenção da aplicação de software, muitas vezes leva a degradação arquitetural. Neste contexto, os arquitetos de software precisam elaborar estratégias para a detecção de sintomas de degradação arquitetural e assim manter a qualidade da arquitetura do software. A elaboração dessas estratégias muitas vezes passa pelo uso de ferramentas linguagens específicas de domínio (DSLs), que ajudam a especificar regras arquiteturais. Essas ferramentas também impõem a adesão destas regras no programa em evolução. No entanto, a sua adoção no desenvolvimento de software tradicional é em grande parte dependente da usabilidade da linguagem. Infelizmente, também é muitas vezes difícil de identificar cedo os seus pontos fortes e fracos de usabilidade, já que não há orientação sobre como revelá-los objetivamente. Usabilidade é uma característica de qualidade multifacetada, que é um desafio para quantificar antes de uma DSL ser usada pelos seus stakeholders. Há ainda menos apoio e experiência sobre como avaliar quantitativamente a usabilidade de DSLs utilizados em tarefas de manutenção de software. Assim, esta dissertação apresenta um framework de medição de usabilidade que foi desenvolvido com base nas dimensões cognitivas de Notações (CDN). O framework foi avaliado qualitativamente e quantitativamente usando duas DSLs textuais para especificação de regras arquiteturais no contexto de dois sistemas orientados a objetos em evolução. Os resultados sugerem que as métricas propostas foram úteis para: (1) identificar precocemente as limitações de usabilidade das DSLs abordadas, (2) revelar características específicas das DSLs que favorecem nas tarefas de manutenção de software e (3) analisar com sucesso oito dimensões de usabilidade que são fundamentais em muitas DSLs. No entanto, juntamente com os resultados dessa avaliação também revelou que este tipo de ferramentas não dá apoio para a comunicação entre stakeholders, criando uma lacuna no desenvolvimento de software. Para solucionar este problema foi proposto heurísticas para ferramentas que usam DSLs para detetar sintomas de degradação arquitetural. Estas heurísticas vão permitir a troca de informações entre stakeholders, assim, aumentando também a usabilidade da ferramenta. Finalmente, nós escolhemos TamDera como ferramenta para implementar essas heurísticas em nosso domínio de estudo. Portanto, implementamos na nova versão do TamDera o suporte de comunicação para stakeholders, utilizando uma nova arquitetura e um novo ambiente para as heurísticas desenvolvidas.

Abstract

Software applications evolve over the years at a cost: their architecture modularity tends to be degraded. This happens mainly because software application maintenance often leads to architectural degradation. In this context, software architects need to elaborate strategies for detecting architectural degradation symptoms and thus maintaining the software architectural quality. The elaborations of these strategies often rely on tools with domain-specific languages (DSLs), which help them to specify software architecture rules. These tools also enforce the adherence of these rules in the evolving program. However, their adoption in mainstream software development is largely dependent on the usability of the language. Unfortunately, it is also often hard to identify their usability strengths and weaknesses early, as there is no guidance on how to objectively reveal them. Usability is a multi-faceted quality characteristic, which is challenging to quantify before a DSL is actually used by its stakeholders. There is even less support and experience on how to quantitatively evaluate the usability of DSLs used in software maintenance tasks. To this end in this dissertation, a usability measurement framework was developed based on the Cognitive Dimensions of Notations (CDN). The framework was evaluated both qualitatively and quantitatively using two textual DSLs for architecture rules in the context of two evolving object-oriented systems. The results suggested that the proposed metrics were useful: (1) to early identify the DSL usability limitations to be addressed, (2) to reveal specific features of the DSLs favoring software maintenance tasks, and (3) to successfully analyze eight usability dimensions that are critical in many DSLs. However, along with these results this evaluation also revealed that this kind of tools lack support for communication among the stakeholders, creating a gap in the software development. To solve this problem we proposed heuristics for tools that use DSLs for detecting architecture degradation symptoms. These heuristics will permit the exchange of information between the stakeholders, thereby, also increasing the tool usability. Finally, we chose TamDera as the tool to implement these heuristics in our study domain. Therefore, we implemented in the new version of TamDera the communication support for the stakeholders by using a new architecture and a new environment with the developed heuristics.

Contents

List of Figures	xi
List of Tables	xiii
List of Listings	xv
1 Introduction	15
1.1 Goals	16
1.2 Dissertation structure	16
2 State of the art	17
2.1 Domain-specific language for detecting architectural problems	18
2.2 Tools with DSLs for detecting architectural problems	19
2.2.1 DETEX	19
2.2.2 TamDera	20
2.3 Summary	21
3 A Framework for DSLs Usability Evaluation	23
3.1 CDN Framework	24
3.2 CDN Instantiation for evaluating DSL Usability	25
3.2.1 Metamodel of a DSL for detecting architecture degradation symptoms	25
3.2.2 Interpretation of the cognitive dimensions	26
3.3 Qualitative Evaluation	28
3.3.1 Qualitative evaluation goal and procedures	29
3.3.2 Data Analysis	30
3.4 Metric Definition	32
3.5 Quantitative Evaluation	36
3.5.1 Evaluation goal	36
3.5.2 Target DSLs	37
3.5.3 Target applications	37
3.5.4 Selection of the versions of the target applications	38
3.6 Data Analysis and Discussion	38
3.6.1 Usability measures	38
3.6.2 Early indicators of usability strengths and weaknesses	40
3.6.3 Usability in specific project settings	43
3.7 Study Limitations	44
3.8 Summary	45
4 TamDera++	47
4.1 Motivating Example	47

CONTENTS

4.2	Limitations of Related Work	49
4.3	Study Decisions	50
4.3.1	Goal	50
4.3.2	Decisions Steps	50
4.4	Implementation	51
4.4.1	Proposed TamDera++ Environment	51
4.4.2	TamDera++ proposed Architecture	52
4.5	Summary	54
5	Conclusion	55
5.1	Future Work	55
	Bibliography	57
	Supplemental Material	63
.1		
	Appendix A	63
.2	Appendix B	64
	Acronyms	65

List of Figures

2.1	MobileMedia (left) and HealthWatcher (right) architectures taken from the dissertation "Blending and Reusing Rules for Architectural Degradation Prevention" [1] . . .	18
2.2	Simplified design of TamDera's tool retrieve from [4]	21
3.1	A metamodel of a DSL for detecting architecture degradation symptoms	26
3.2	Cognitive dimensions of Expressiveness	34
3.3	Cognitive dimensions of Conciseness	34
4.1	Architecture of the MobileMedia (version 1)	48
4.2	Architecture of the MobileMedia (version 4)	49
4.3	TamDera++ proposed environment	52
4.4	TamDera++ proposed architecture	53
4.5	TamDera++ User Interface	53

List of Tables

1.1	Publications directly related to this dissertation	16
3.1	Cognitive Dimensions Originally defined by CDN [23, p.116-8]	24
3.2	Basic terminology of the metamodel	27
3.3	Information required in the survey	30
3.4	Questions used in the interview	30
3.5	Instantiation agreement	32
3.6	GQM instantiation	33
3.7	Expressiveness metrics (for question 1)	35
3.8	Conciseness metrics (for question 2)	36
3.9	Results of the Expressiveness metrics	39
3.10	Results of the Conciseness metrics	40

List of Listings

2.1	Example of a specification using DETEX DSL [9]	19
2.2	Example of a specification using TamDera DSL	20
3.1	Rules for Business Facade in TamDera	41
3.2	Rules for Business Facade in DETEX	41

Chapter 1

Introduction

Nowadays, several studies point out that maintainability is one of the main cost factors in software development projects [34, 35, 36]. This factor made software architects, programmers and code reviewers be concerned with architectural degradation and the problems it would bring to software maintainability. Many studies [2, 5, 7] confirmed how software architecture would eventually degrade with undisciplined changes throughout software maintenance and evolution. In this context, architects need to elaborate strategies for detecting architectural degradation symptoms and thus maintaining the software architectural quality. A common strategy relies on tools (e.g. [4, 9]) that use a unified [domain-specific language \(DSL\)](#), which help them to specify software architecture rules. In particular, [DSLs](#) for this domain are used by software architects, programmers and code reviewers to specify and check the adherence of the source code with respect to architecture rules. However, it is particularly challenging to design a usable [DSL](#) in this domain for several reasons [2, 5, 33], including: (1) it needs to offer a concise set of abstractions in order to enable architects to express the high-level design rules, (2) it needs to be concise and expressive enough in order to support programmers and code reviewers in understanding which program elements are affected by the architecture rules, and (3) it needs to be expressive enough to allow users to tailor the architecture rules as they implement, maintain and evolve modules of a program.

Unfortunately, it is also often hard to identify their usability strengths and weaknesses early, as there is no guidance on how to objectively reveal them. Usability is a multi-faceted quality characteristic, which is challenging to quantify before a [DSL](#) is actually used by its stakeholders. There is even less support and experience on how to quantitatively evaluate the usability of [DSLs](#) used in software maintenance tasks. To this end in this dissertation, a usability measurement framework was developed based on the [Cognitive Dimensions of Notations \(CDN\)](#).

However, reports from this framework revealed that one of the usability prerequisites is the lack of communication among the stakeholders, creating a gap in the software development. We believe the resolution of this lack of communication would allow a better specification of the architecture rules by the software architects and faster learning of the application architecture by the programmers. Therefore, we proposed three heuristics for tools that use [DSLs](#) for detecting architecture degradation symptoms. These heuristics permit the exchange of information between software architects and programmers, thereby, also increasing the tool usability. Finally, in our study domain we chose TamDera as the tool to implement these heuristics. We selected TamDera due to the fact that it was already designed with different categories of stakeholders in mind. Therefore, we created a new version of TamDera with communication support for the stakeholders by implementing a new architecture and a new environment with the developed heuristics.

1.1 Goals

The focus of this dissertation is to improve the usability of existing tools with [DSLs](#) for detecting architectural degradation symptoms. Their adoption in mainstream software development is largely dependent on the usability of the language. However, there is little support and experience on how to quantitatively evaluate the usability of [DSLs](#) used in software maintenance tasks. Therefore, a usability measurement framework was developed based on the [CDN](#) to evaluate [DSLs](#) for this domain as one of the goals for this dissertation. Thereafter we proposed heuristics for enhancing the tools usability that use [DSLs](#) for detecting architecture degradation symptoms. These heuristics were based from the evaluation of the usability measurement framework. In order to implement this heuristics we chose TamDera, a tool that leverages a [DSL](#) for supporting detection of architectural anomaly symptoms in the source code. We intend to improve TamDera with enhanced communication support for the stakeholders by using a new architecture and a new environment with the proposed heuristics of this study. With that, it will permit the exchange of information between the stakeholders to be more consistent, thereby, also increasing the tool usability.

The contributions of this dissertation were reported in papers, which have been published or are under submission. However, those studies were instrumental to reveal the research problem being addressed in this dissertation. These papers are listed in table 1.1.

Table 1.1: Publications directly related to this dissertation

Albuquerque et al. Quantifying Usability of Domain-Specific Languages: An Empirical Study on Software Maintenance. Submitted to Journal of Systems and Software, 2014.
Albuquerque et al. Promoting Cooperative Detection of Architectural Anomalies in Evolving Systems. Ready to submit to SBCARS, 2014.

1.2 Dissertation structure

The next chapter contains background content about the architectural degradation, [DSLs](#) and information of tools for detecting architectural degradation symptoms. Chapters 3 provides details about the measurement framework development: the eight usability dimensions; qualitatively and quantitatively evaluation using two textual [DSLs](#); revelation of specific features of the [DSLs](#) favoring software maintenance tasks. The development of the heuristics in TamDera, including design decisions, and implementation details, are covered in chapter 4. The last chapter concludes this dissertation and suggests future work directions to continue research on this topic.

Chapter 2

State of the art

Several studies have shown that maintainability is one of the main cost factors in software development projects [34, 35, 36]. This contributed to software architects, programmers and code reviewers be concerned with architectural degradation and the problems it would bring to software maintainability. Many studies [2, 5, 7] confirmed how software architecture would eventually degrade with undisciplined changes throughout software maintenance and evolution. These studies have been conducted to investigate the relationship between the architectural degradation, and the so-called architectural anomalies (drift and erosion anomalies).

Architectural erosion is defined as “the process of introducing a decision into a system that violates dependency rules of elements defined in the system’s intended architecture” [7]; a simple example is an unintended dependency established in a program between code elements realizing two architectural components. In other words, the dependencies in the *implemented* architecture diverge from the dependencies defined in the *intended* architecture. This is possible to see in Figure 2.1 where is shown a architecture pattern Model-View-Controller (MVC) from Mobile-Media system. For example, the class BaseController defined in the Controller component invokes the Data services provided by AlbumData. This causes the BaseController handling exceptions thrown by AlbumData, which should have been handled by the Model component. Therefore, it introduces unwanted dependencies between code modules and thus diverging from the *intended* architecture of the MobileMedia. **Architectural drift** is “the introduction of design decisions into a system that were not included in the intended architecture, albeit they do not violate any of the prescribed dependency rules” [7]. Typical examples of drift anomalies are related to architecture rules realizing design principles, such as narrow component interface, low-coupled components or single responsibility assigned to each component [28]. Once again lets take as an example the class BaseController in Figure 2.1. This class has many methods, therefore, becoming the source of a code anomaly, called Large Class [5]. This anomaly contributes to the manifestaion of an architectural drift symptom, called Ambiguous Interface [5]. This architectural drift symptom happens because this class provides an over-generalized interface for handling all commands. In other words, the BaseController module in the source code is aggregating several responsibilities from different service requests that should not have.

It is prohibitive to check all anti-erosion and anti-drift rules in an ad hoc fashion as systems are developed and maintained. In this context, software architects need to specify strategies for detecting both types of architectural anomalies in order to support the software architecture maintenance. Programmers and code reviewers also need to be informed when their implementation changes violate one or more anti-drift and anti-erosion rules. A common strategy relies on tools (e.g. [4, 9]) that use a unified DSL, which helps them specify software design rules. Anti-erosion rules in these DSLs define dependency constraints between code elements realizing architectural elements. Anti-drift rules define constraints related to attributes of code elements realizing architectural elements. They are based on the use of metrics and thresholds to identify the violation

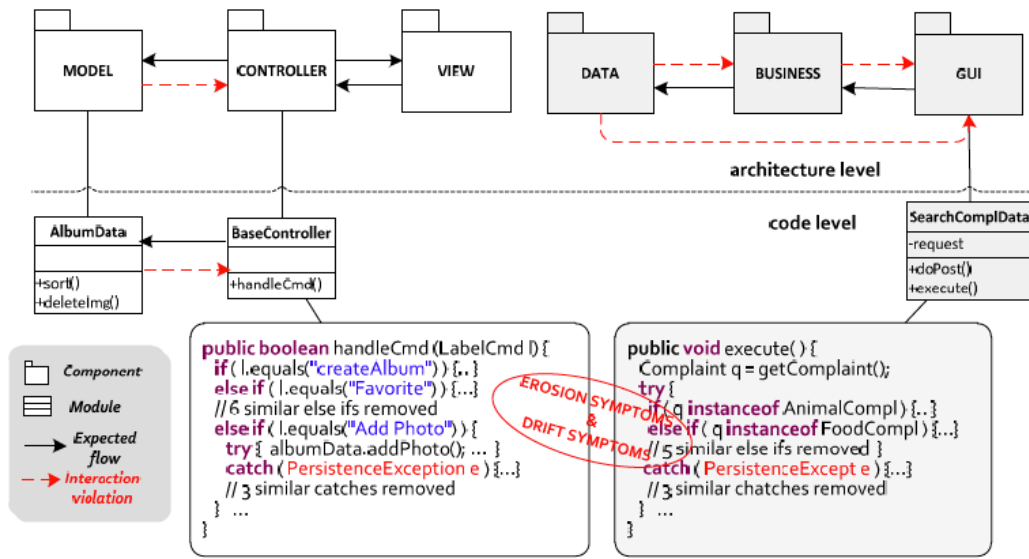


Figure 2.1: MobileMedia (left) and HealthWatcher (right) architectures taken from the dissertation "Blending and Reusing Rules for Architectural Degradation Prevention" [1]

of design attributes. For instance, they rely on coupling metrics to identify the coupling of classes realizing a particular architectural element.

The remainder of this Chapter is organized as follows: Section 2.1 will further discuss the key characteristics of a DSL intended to support the specification and checking of architectural anomalies in a program. And Section 2.2 will present some of the those DSLs for detecting architectural problems and their respective tools in which they are embedded.

2.1 Domain-specific language for detecting architectural problems

A DSL is a type of programming language or specification language in software development dedicated to a particular problem or solution domain [10, 12]. A DSL facilitates software development through appropriate abstractions and notations. Several studies [10, 11, 12] identify various benefits of using DSLs in the area of software engineering, including the provision of an idiom at the level of abstraction of the problem domain. These studies also show how the expressive power of DSLs is significant when they are properly designed for one specific domain.

Nowadays there are currently hundreds of DSLs, in a wide range of domains in the context of software systems, engineering, and telecommunications, among others [10]. In particular, there are several DSLs in software engineering particularly intended to support developers in specifying design rules at different levels of abstraction (e.g. [4, 8, 9, 41, 42, 43]). For instance, some DSLs are intended to support programmers in defining low-level design rules that are relevant at the implementation level (e.g. [8, 43]). Hence, we chose to apply our study to the domain of architecture-level design rules. In addition, several studies have reported that existing languages for defining design rules are not expressive and concise enough, in particular, when rule changes need to be made through software maintenance and evolution [8, 12, 44]. Moreover, DSLs for

defining architecture rules have been recently proposed [4, 9, 14, 44]. Nevertheless, these DSLs have not yet been explicitly assessed with regard to usability. We have not found many DSLs that provide support for a wide range of architecture design rules, in particular for the detection of architectural anomalies. DSLs, such as TamDera or DETEX, were created to fill this gap [4, 14]. Moreover, according to Humm and Engelschall [12], most of the existing DSLs for detecting architectural anomalies have low conciseness in general, because they follow Java-like or SQL-like syntax [12]. Examples like F#, Ruby, Groovy, and Scala fall in this category [12]. In this context, we are interested in evaluating the usability of DSLs in this domain, from the point of view of software architects, programmers and code reviewers when using DSL specifications. The selected DSLs for our study are described in the following section.

2.2 Tools with DSLs for detecting architectural problems

There are several solutions for detecting architectural problems [4, 14, 52, 61, 64]. As stated previously, a common tactic is to use DSL in order to detect these anomalies. Thus in the following sections we will present DSLs for detecting architectural problems and their respective tools in which they are embedded.

2.2.1 DETEX

The first DSL chosen is called DETEX, an instantiation of the method DECOR, supported by the Ptidej tool [9, 14]. In more detail, DECOR is a method that states all the necessary steps to define a detection technique. However, DECOR represents a generic method for the detection of code anomalies, therefore, DETEX was created. DETEX is an instantiation or a concrete implementation of DECOR in the form of a detection technique. Therefore, the steps of DETEX are: (1) Domain analysis that consists of performing an analysis of the domain related to code anomalies in order to identify key concepts, (2) Specification using a DSL in the form of rule cards using the previous vocabulary. The DETEX DSL is defined with a Backus Normal Form (BNF) grammar (appendix .1). The DETEX DSL is constituted by a **rule card** (as shown in line 1 of Listing 2.1) as a set of rules and a **rule** (lines 2-8) describes the properties that a class must have in order to be considered a code anomaly.

Listing 2.1: Example of a specification using DETEX DSL [9]

```

1 RULE_CARD: SpaghettiCode{
2 RULE: SpaghettiCode {INTER LongMethod NoParameter NoInheritance NoPolymorphism
   ProceduralName UseGlobalVariable};
3 RULE: LongMethod {METRIC LOC_METHOD VERY_HIGH 10.0};
4 RULE:NoParameter LongMethod {METRIC NMNOPARAM VERY_HIGH 5.0};
5 RULE: NoInheritance {METRIC DIT 1 0.0};
6 RULE: NoPolymorphism {STRUCT NO_POLYMORPHISM};
7 RULE: ProceduralName { LEXIC CLASS_NAME (Make, Create, Execute)};
8 RULE: UseGlobalVariable {STRUCT USE_GLOBAL_VARIABLE};
9 };

```

The DETEX DSL allows the definition of properties for the detection of code anomalies, specifying the structural relationships among these properties and characterizing properties according to their internal attributes using metrics (line 5), structure (line 6), and their lexicon (line 7), (3) Algorithm generation made from models of the rule cards. These models are created by reifying the

rules using a metamodel and a parser, and (4) the Detection algorithms are applied automatically on models of systems obtained from original designs.

2.2.2 TamDera

To avoid symptoms of architectural degradation, there are various tools and techniques that allow the detection and prevention of architectural anomalies [4]. However, there is no tool that can detect or prevent the two known types of architectural anomalies. TamDera [4] is the only tool that we have knowledge that can perform this hybrid strategy for architectural degradation [4]. Its language allows not only to create rules using anti-erosion and anti-drift but also allows the reuse of the same rules in multiple contexts. To achieve this objective, TamDera DSL provides two abstractions as defined with a BNF grammar (appendix 2): (1) **architectural concept**, also denoted by the keyword *concept*, represents a relevant concern to the mind-set of software architects. These concerns can be components, interfaces, or any other decision expressed in an architecture document (as shown in line 1 of Listing 2.2), and (2) **concept mapping** where each module elements comprise each architectural concept. The concept mapping is created through regular expressions that identify properties shared by module elements realizing the concept. Hence, common names (suffixes, prefixes, and package names) are denoted by the keyword *name* and common parent (super class or interface) of code elements denoted by the keyword *parent* (line 2 of Listing 2.2).

Listing 2.2: Example of a specification using TamDera DSL

```
1 concept GUIHW{
2 parent:"Command"
3 LOC < 100
4 CC < 5
5 }
6 GUIHW must-derive AbstractCommand
```

Moreover, TamDera DSL allows the specification of anti-degradation rules in terms of architectural concepts and their interactions. The anti-erosion rules refer the concepts names in specific declarative statements. In other words, they describe interactions between the elements comprising the concepts (line 6 of Listing 2.2). The anti-drift rules are defined to a particular architectural concept, i.e., they establish boundaries (or thresholds) on the structural properties of the implementation elements composing the respective concept (line 3-4 of Listing 2.2).

TamDera tool design is composed of four components: Controller, Concept Mapper, Consistency Checker and Rule Translator (Figure 2.2). Each one of them has a particular responsibility with respect to the detection architectural anomalies. However, the tool is dependent on two basic artifacts: the system source code and the TamDera architecture models. Moreover, TamDera tool also uses external tools as **Bytecode Analysis Toolkit (BAT)** to retrieve a Prolog-based representation of the system (step 2), Together [66] obtains several measurements for module properties (such as size and coupling used for describing anti-drift rules), and (step 3) a Prolog engine [65] to statically check the conformance of anti-degradation rules (step 8).

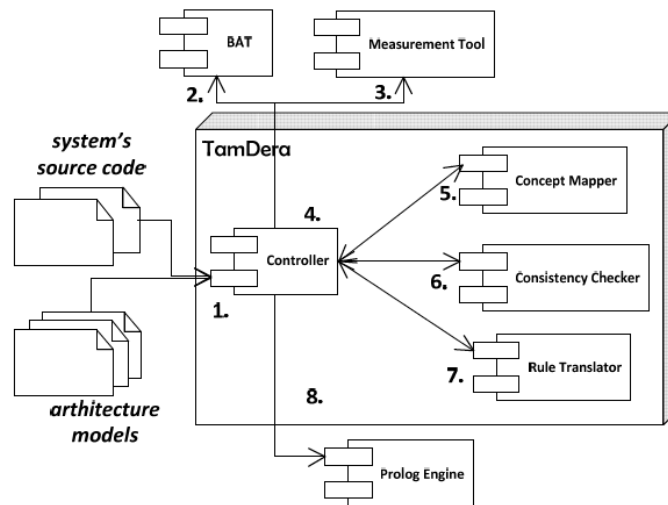


Figure 2.2: Simplified design of TamDera's tool retrieve from [4]

2.3 Summary

In this chapter, it was introduced the general background and outlines related work on architectural degradation prevention. Therefore we presented the relationship between the architectural degradation, and the so-called architectural anomalies throughout software maintenance and evolution. Moreover, we presented a common strategy that relies on tools that use a unified **DSL**, which helps specify software design rules. From this common strategy we mention our focus on **DSLs** for detecting architectural anomalies on the expressiveness and conciseness attributes of the **DSLs** as criteria that define a usable **DSL** for detecting architectural anomalies. Finally we provided a more detailed information about some of the **DSLs** for detecting architectural problems and their respective tools.

Chapter 3

A Framework for DSLs Usability Evaluation

A **DSL** aims to facilitate construction of software artifacts through specialized abstractions and notations [20]. **DSLs** are increasingly being used in many software engineering activities, including designing and checking architectural rules (e.g. [4, 9]). Nevertheless, the difficulties of using **DSLs** have become more apparent when exposed to software maintenance circumstances [10, 32]. Several studies [10, 20, 22, 32, 37] concluded that these difficulties might adversely lead to higher maintenance effort. An important factor that contributes to increased maintenance effort is the low usability of such **DSLs** [45]. The usability of a **DSL** artifact (e.g., a specification built using the **DSL**) is the quality that makes it easy for users to understand, learn, and interact with it [20, 45].

Recently, we observed some studies concerned with analyzing the usability of **DSLs** from several point of views [10, 12, 22]. There is, however, a lack of studies which rely on quantitative analysis to complement the qualitative analysis of the **DSLs** usability. The creation of a metric suite to support the quantitative analysis of **DSLs** would allow an objective comparison between **DSLs** [45, 46, 51], therefore complementing the qualitative analysis approaches found in the literature [37, 38, 39]. The results would be more reliable and provide extra information at early design stages of a **DSL** than approaches without any quantitative analysis. Moreover, such a metric suite would support the early evaluation of **DSL** usability in order to help choose the most appropriate **DSL** given the nature of the software maintenance tasks.

Concerned with the aforementioned issues, we report a study conducted to compare the usability of textual **DSLs**¹ for detecting architectural problems [4, 9, 14, 44]. In particular, we defined a usability metrics suite that was developed based on the **CDN** framework [23]. We instantiated these cognitive dimensions for evaluating **DSLs** and assessed them by a qualitative process. These instantiations of the **CDN** capture usability aspects of **DSL** artifacts relevant to software maintenance tasks. Data were collected from two **DSLs** [4, 9] for detecting architectural problems. The two chosen **DSLs** explicitly embed constructs to define architectural design rules so that they can be checked in the source code. In addition, both **DSLs** were designed for different categories of stakeholders, including software architects, programmers and code reviewers.

The remainder of this Chapter is organized as follows: Section 3.1 gives information about **CDN** framework used in this study. Section 3.2 describes the steps required to create the metrics. Section 3.3 describes the design of a qualitative study aimed at assessing the proposed instantiation of the cognitive dimensions. Section 3.4 describes the metrics suite developed to analyze the usability of **DSLs**. Section 3.5 describes the design of an exploratory study aimed at comparing the two textual **DSLs** and assessing the usefulness of the proposed metrics. The results of the study are analyzed and discussed in Section 3.6. Section 3.7 describes the threats to the validity of our study. Finally, Section 3.8 concludes the work and suggests future developments.

¹From hereafter, we use the term “**DSLs**” to refer only to textual **DSLs**.

3.1 CDN Framework

The **CDN** framework is “a set of discussion tools for use by designers and people evaluating designs” [23]. We chose this framework because we found that it is a widely used technique to support usability evaluation in the literature [6, 25, 31]. This framework provides **cognitive dimensions (CDs)** of general use in different domains, as shown in Table 3.1. These **CDs** are conceptual tools defined to help the designer or evaluator to reason about the system or language being assessed [6, 23]. In addition, these **CDs** allow “to improve the exchange of experience, opinions, criticism and suggestions” [6]. This framework was originally proposed to evaluate notational systems for designing artifacts, aiming “to improve the quality of discussion” [23, p.107]. These **CDs** cover a wide range of issues and, consequently, their definitions may lead to different interpretations. Previous work has employed this framework to qualitatively evaluate the design of **DSLs** in different contexts [6, 25].

However, to the best of our knowledge, no previous study has defined a **CDN**-based metrics suite to support a quantitative evaluation of **DSLs**. We selected a subset of the **CDs** to support the evaluation of **DSLs** in evolving systems. According to the literature, **DSLs** comprise four important aspects: expressiveness, conciseness, integration, and performance [12]. However, only the first two characteristics are considered in this study, since they are important in terms of the language itself. In other words, we aim to evaluate the specifications that the user-developer needs to understand and/or produce and not the interaction of the language with some tool. These two characteristics are defined as: (1) **DSL Expressiveness**, which refers to the extent a domain-specific language allows to directly represent the elements of a domain, and (2) **DSL Conciseness**, which refers to the economy of terms without harming the artifact comprehension.

Table 3.1: Cognitive Dimensions Originally defined by CDN [23, p.116-8]

Cognitive Dimension	Description
Viscosity	Resistance to change
Visibility	Ability to view entities easily
Premature Commitment	Constraints on the order of doing things
Hidden Dependencies	Relevant relations between entities are not visible
Role-Expressiveness	The purpose of an entity is readily inferred
Error-Proneness	The notation invites mistakes and the system gives little protection
Abstraction	Types and availability of abstraction mechanisms
Secondary Notation	Extra information in means other than formal syntax
Closeness of Mapping	Closeness of representation to domain
Consistency	Similar semantics are expressed in similar syntactic forms
Diffuseness	Verbosity of language
Hard Mental Operations	High demand on cognitive resources
Provisionality	Degree of commitment to actions or marks
Progressive Evaluation	Work-to-date can be checked at any time

3.2 CDN Instantiation for evaluating DSL Usability

As mentioned in Section 3.1, the CDN framework provides general definitions for its CDs and, therefore, they need to be refined to particular contexts. The interpretation of the CDs might lead to ambiguous and overlapping definitions [6]. For example, suppose a situation where the user wants to write an artifact with the DSL in a constant and similar manner. This type of situation can be interpreted as a case of either Consistency or Viscosity. This issue is important to keep in mind because it can bring additional challenges for those who instantiate usability evaluation frameworks, such as the one proposed here. Therefore, we noticed that we should clearly define the instantiation of the CDs, in our case to capture usability aspects of DSL artifacts relevant to software maintenance tasks. In addition, someone can reuse (or discard) our definitions if the instantiation satisfies (or not) the expectations about each CD in their context.

To support this goal, we noticed the importance of first creating a metamodel that clearly defines all the characteristics of the DSLs for architectural degradation detection (Section 2.1). The metamodel supports the definition and interpretation of the CDs for our study domain. This metamodel was created because it formalizes the domain language [20], providing the DSL evaluator with a notation to identify key characteristics of the DSLs. This also provides them with the basis on to interpret each CD in terms of specific DSLs.

The next subsections describe our targeted interpretation in terms of DSLs (subsection 3.2.1) and the interpretation of the CDs for DSLs (subsection 3.2.2).

3.2.1 Metamodel of a DSL for detecting architecture degradation symptoms

As aforementioned, we argue that is important to define a metamodel (Figure 3.1) that represents all DSLs properties found in our domain [9, 20]. In this way, it is possible to verify whether the properties of a particular DSL for detection of degradation are encompassed in this metamodel. Having this confirmation then it is possible to use the CDs to evaluate DSL expressiveness and conciseness. In Table 3.2, we describe the metamodel and its interrelationships. In this metamodel the two most important entities are: Element and Concept. These two entities form all the rules necessary to define symptoms of architecture erosion and drift (Section 2). Element is an aggregation of one or more rules that assemble a Concept (Table 3.2). Each entity connected to Element represents a characteristic of a restriction, also known as a rule, for detecting architectural degradation. A Concept represents an architectural module of a system, such as a component. Each Concept is assembled of Elements, and different Concepts are included into one or more files (File).

An Element can contain only one rule (SingleElement) or several rules (ComposableElement); the composition of rules is based on Operators of intersection, union, difference, inclusion, and negation. Element composition can contain other Element by extension or inheritance (Interaction) or with relationships (Relationship) such as association or aggregation. Elements can have two rule formats in terms of architectural degradation that undergo by a mapping (ConceptMapping): drift restrictions and erosion restrictions. Drift restriction (AntiDriftRule) is an expression (Expression) that can have three properties (Property): (1) Structural, which is composed of properties that represent structural components (e.g., the detection of a global variable), (2) Lexical, which is composed of properties that represent the vocabulary used to realize a component in the implementation, such as classes, interfaces, methods or fields, and (3) Metric, which selects a measure and a threshold to enable the identification of a drift architecture anomaly (e.g., the maximum coupling that will be restricted to one or more architectural components of a system). Another

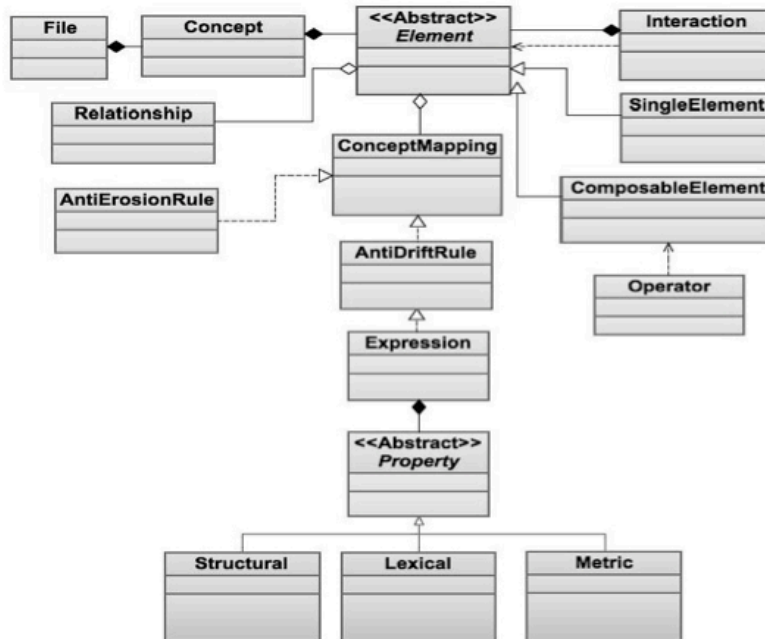


Figure 3.1: A metamodel of a DSL for detecting architecture degradation symptoms

type of constraint is the Erosion constraint (AntiErosionRule) where the user creates dependency restrictions to the Elements of a system. For example, prohibiting access of code Elements from the **Graphical User Interface (GUI)** layer to services provided by Data layer Elements.

3.2.2 Interpretation of the cognitive dimensions

To be able to adequately interpret all the **CDs** in Table 3.1, it is required to instantiate them for the purpose of our quantitative analysis. Quantification of a **CD** is only possible if there is a well-defined character of it in the target domain of study. Therefore, we used the definitions of the **CDN** framework [23] and tailored them for **DSLs** of the studied domain, as shown below. Moreover, the instantiation of the **CDs** underwent a refining process through a qualitative evaluation study (Section 3.3). The qualitative evaluation was the process required to verify if our **CDs** interpretation was (or was not) acceptable to different users and experts in our context of study.

Viscosity, the amount of required changes in the **DSL** specifications to adapt their Concepts for a different use. This characteristic can be quantified by computing the number of Elements changed in the **DSL** specification when those Elements are used in different contexts in each new version of the target application.

Visibility, how easy it is to visualize related portions of the **DSL** specifications. We consider how the Concepts of a **DSL** are distributed; in other words, in how many files the **DSL** specification is distributed. For example, for non-complex languages it is easier for the user to see all the **DSL** specifications without extensions or inheritances. In that way, the user can add and see all the Elements of the **DSL** specifications without changing files.

Diffuseness, how many Elements are necessary to define the **DSL** specification. If a Concept needs additional elements defined in a **DSL**, we say that this Concept is more diffuse. For example, if a Concept can be written with only four Elements in a **DSL** and requires three Elements in another

Table 3.2: Basic terminology of the metamodel

Word	Description
AntiDriftRule	Constraints governing the characteristics of architectural Concepts
AntiErosionRule	Constraints governing the Concepts of architectural interaction
Concept	Each Concept is a relevant concern to the software architect; it is realized by a set of module Elements in the architecture implementation. Each Concept has a set of rules, described in a DSL specification, exploiting multiple properties of module Elements to detect individual code anomalies or anomaly patterns
Concept Mapping	It is an expression that describes how a Concept is realized in the source code by exploiting properties shared by module Elements realizing the Concept
Element	Aggregation of one or more rules that assemble a Concept
Expression	Aggregation of properties
File	File where the Concepts with their possible dependencies are located
Interaction	Concepts that are imported, extended and inserted into another Concept
Lexical	Rules that represent the vocabulary used to name a class, interface, method, field, or a parameter
Metric	It is related to rules that are composed of a software metric, a mathematical operator and a value. These rules can also use threshold and fuzziness
Module Elements	Elements in the source code of a module; A range of classes and interfaces to inner members of modules, such as methods
Operator	Elements can be combined using multiple set operators including intersection, union, difference, inclusion, and negation
Structural	Rules that represent structure of a constituent (class, interface, method, field, parameter, etc)
Property	One of the three types of rule in an AntiDriftRule: Structural, Lexical or Metric
Threshold	List of threshold variables that can be defined. It can be numerical values or ordinal values used to define all the Concepts under analysis
Value	Cardinal value
Relationship	Relationships with other Concepts, such associations, aggregation and composition
Related Portion	Constructions with Concept inheritance

DSL, then we say that the DSL with four Elements is more diffuse.

Premature Commitment, the early steps required to create a given DSL specification. In other words, are all the steps necessary to realize before defining a Concept. This cognitive dimension is not applicable to our investigation to evaluate expressiveness and conciseness, because DSLs for architectural degradation do not have this kind of characteristic.

Hidden Dependencies, unexpressed architectural dependencies between different Concepts defined in the DSL specifications. They represent existing architectural dependencies that cannot be explicitly described in the DSL specification.

Error-Proneness, the amount of possible errors that cannot be detected in an early stage of the

DSL specification process. Detection of Elements that can only be detected during the actual DSL specification. This occurs when the Concept created can inherit Elements of another Concept. This cognitive dimension was discarded because, in our interpretation of DSLs, we believe it has already been covered by another dimension, more precisely, Closeness of Mapping.

Progressive Evaluation, the ability to test part of the DSL specifications during development. This cognitive dimension is not applicable to our investigation to evaluate expressiveness and conciseness. That happens because we are studying the language specification itself and not the interaction and execution of the DSL [12], which do not comprise the aspects of DSLs we are analyzing in this study.

Role-Expressiveness, determines how many representations can be used to express the purpose of a Concept in a DSL specification. For this cognitive dimension, we identify the possible representations used to define the role of each Concept. For example, if in the DSL realization a Concept can be associated with a Class and a Method, then there are two possible representations for characterizing the purpose of a Concept in the DSL.

Abstraction, the number of abstractions the developer must use or create to define a Concept. We consider that the abstractions in a DSL are the creation and use of Elements in the defined Concepts. In other words, it is the total number of Elements that constitute each Concept.

Closeness of Mapping, how close the DSL specification is to the architectural conceptual domain. The DSL specification may be distributed across different files and not just one. Because of this, the number of Elements per Concept can be misleading. This happens because, if an Element inherits Elements from another Concept, they do not need to be set again, thus artificially reducing the number of Elements defined per Concept. Therefore, we need to check whether the main file of the DSL specification (without extension or inheritances) is identical to the real DSL specifications (with extensions or inheritances).

Consistency, how similar the DSL notations and abstractions in the DSL specification are. This cognitive dimension is not applicable to our study to evaluate expressiveness and conciseness, because DSLs for architectural degradation do not have this kind of characteristic.

Hard Mental Operations, operations that require the developer to think about many DSL notations and abstractions at the same time. Some Elements that the developer seeks are scattered across different Concepts in the DSL specifications. In this way, the user needs to remember more information when implementing a new Concept.

Provisionality, the ability to change/adapt parts of the DSL specification in the future. We see change/adaptations in DSLs as Elements that are needed to be modified in each Concept. In this way this CD shows the necessary Elements to be implemented per Concept. This CD was discarded because, in our interpretation of DSLs, it has already been covered by another dimension (i.e., Abstraction).

Secondary Notation, the support for additional DSL notations and abstractions information without formal syntax. It involves the extra information of Element distribution in a DSL specification. In other words, if the DSL supports comments in the DSL specification. This CD was discarded, because DSLs for defining architectural rules generally lack properties that make this type of CD reveal important characteristics.

3.3 Qualitative Evaluation

As mentioned in subsection 3.2.2, the CDs of the CDN framework need to be refined to particular contexts. The CDs instantiation is required given the general and overlapping definitions

of the **CDs** [6]. However, the interpretation of the **CDs** also tends to be subjective and multiple interpretations may be framed by different experts working in the same field. Thus, an assessment was required to verify if our interpretation of the **CDs** was (or was not) acceptable and valuable to different experts in our context of study. With this requirement in mind, we decided to conduct a qualitative evaluation to assess our proposed cognitive dimension. This qualitative evaluation gave us insights on the **CDs** interpretation by practitioners directly involved in the use or development of **DSLs**. Their experience was also useful to give us additional insights on **DSL** usability evaluation we had not thought of beforehand. As a consequence, we could check to what extent our interpretation of the **CDs** could accommodate different viewpoints, and, if required, perform refinements in the **CDs** instantiation.

Another step of the qualitative evaluation was to use our **CDs** instantiation in order to evaluate **DSLs** usability. This step was performed to analyze whether the participants would be able to evaluate correctly all the instantiation of the **CDs**. Therefore, this gave us information to compare with the data obtained in the quantitative evaluation (Section 3.5). As a result, we could check to what extent our metrics were effective to evaluate **DSLs** usability. In the following, we first describe the goals of the qualitative evaluation as well as the data collection procedure (subsection 3.3.1), and then we analyze and discuss the results obtained (subsection 3.3.2).

3.3.1 Qualitative evaluation goal and procedures

Our research goal was to assess with experts of the field our instantiation of the **CDN** framework (Section 3.1) and use those proposed **CDs** to evaluate the usability of two **DSLs**. The assessment focused on analyzing whether the proposed **CDs**: (1) were properly framed to our particular context, and (2) were useful for analyzing **DSLs** usability.

The procedures involved the recruitment of four participants with experience in either using or developing **DSLs**. Two of them had developed **DSLs** focused on supporting software development tasks; the other two have practical experience in using **DSLs** specifically aimed at detecting architectural degradation symptoms in large-scale software projects. All the participants had extensive theoretical knowledge about architectural anomalies. Moreover, the participants have diverse experience in software development projects (from two to seven years). This heterogeneity helped to gather a wider perspective on the assessment of our propositions related to the **CDs**.

During the two steps of the qualitative evaluation we provided to the participants, as support material: (i) the metamodel, (ii) two files with the **DSL** specification adapted to the two **DSLs** of the study, and (iii) their respective BNFs. We designed a survey to the participants in order to find out what was their rate of agreement in relation to the **CDs** instantiation. Therefore, in the survey we asked to the participants to develop their own interpretation of the **CDs** from the original definition of the **CDN** framework. This individual interpretation is important for the participants in order to have a basis for answering the following survey questions. During the survey the participants were encouraged to speak freely while answering the questions. After that, we conducted a semi-structured interview with the participants who had the lower rates of agreement in the survey, to get more information about the rationale about their disagreement. The interviews enabled us to identify improvements for the **CDs** instantiation from those practitioners. Finally, we also had to eliminate survey misunderstandings and confirm whether our interpretation of their answers was correct.

The following steps were carried out to accomplish our qualitative evaluation:

- Defining goals and define the process of this assessment;

- Selecting practitioners who have some experience in DSLs and architectural anomalies;
- Conducting surveys with the practitioners using structured questions. All the surveys were recorded;
- Conducting semi-structured interviews with practitioners. All interviews were recorded;
- Conducting a qualitative evaluation of two DSLs with the instantiations of the CDs of CDN framework;
- After each interview, we transcribed the interviews' recorded content;
- Data interpretation – analysis of each factor of influence.

We used semi-structured and open questions in the interviews to allow a detailed investigation about the context in which the interviewees were immersed. This procedure allowed us to make explicit the interviewees' tacit knowledge. Table 3.3 and Table 3.4 show the questions asked in the survey and in the interview, respectively. The interviewees' answers to such questions would give us information to guide the next steps of the research evaluation.

Table 3.3: Information required in the survey

Write your instantiation from the definition above
Is the above Cognitive Dimension useful? (yes, maybe or no)
What is the level of agreement of the instantiation? (1 = strongly disagree to 5 = strongly agree)

Table 3.4: Questions used in the interview

What could be improved on the cognitive dimension with which you disagreed?
What did you disagree on? Something in the instantiation or in its explanation?
Do you think it is possible to evaluate this cognitive dimension for DSLs for detecting architectural anomalies?

Each survey and interview were fully transcribed. By using the transcriptions, we were able to thoroughly analyze the interviewees' knowledge and opinions. We must point out that we have anonymized the transcriptions in order to preserve the participants' identities.

3.3.2 Data Analysis

Table 3.5 presents the overall results of the instantiation agreement in the survey. Each line represents a cognitive dimension. The columns represent the answers of each participant regarding our instantiation of the CDs of the CDN. The values range from one to five, where one represents a strong disagreement (lightest cell color), and five means a strong agreement (darkest cell color). It is also important to notice that the answer three in the Table 3.5 means that participants neither agree nor disagree with the instantiation of a specific CD. This type of answer might happen due to the following reasons: (1) the participant was not able to interpret the instantiation of the CD within our context, or (2) the participant did not understand our proposed CD instantiation.

In our analysis, we found a difference of values in our data related to the level of agreement in Table 3.5. This happened because we had 86% of agreement in the **CDN** instantiation by two participants that were previously classified as **DSLs** users (B and C). Moreover, we had a lower agreement by the participants that already have developed **DSLs** (A and D). These values indicated that further reflection or refinement for some **CDs** instantiation should be considered, as they imply the interpretation was different from **DSL** developers to users. Therefore, we observed that the heterogeneity of people's views reflect in our user analysis. This understanding is an important one to consider while evaluating any particular **DSL**. In addition, from the survey and the interviews we found that it was not easy to reach a consensus in the refinement of some **CDs**. For example, the instantiation agreement for Role-Expressiveness in Table 3.5 shows that two participants disagreed on some point whereas the two others strongly agreed. Therefore, such information indicated that in our **CDs** instantiation there might still exist disagreement in the interpretation by some user or developer in the future. And once again a refinement is needed to close that gap as aforementioned.

The **CDs** instantiation with the highest agreement (all answers were 4 or 5) for all the participants were: **Viscosity**, **Hidden Dependencies**, **Abstraction**, **Diffuseness**, **Hard Mental Operations**, and **Progressive Evaluation**. This indicated that our interpretations were strongly consistent with the participants' point of view. Hence, this result indicated that these **CDs** instantiation: (1) were closer to the **DSLs** developers' and users' interpretation, and (2) might be less sensitive to interpretation in our research domain. This understanding is very valuable to improve a given **CDs** instantiation, that is, to make it more useful and/or easier to use and, hence, more viable for **DSL** evaluation. **Visibility** and **Provisionality** were the **CDs** instantiation with mostly strong agreement, except for a medium agreement (answer 3). This indicated that the **CDs** instantiation was consistent with the point of view of the **DSLs** developers and users. However, the answers also showed that the **CDs** instantiation, despite being on the right path, needed some refinement to be used.

The ones with divergent agreement were: **Role-Expressiveness**, **Error-Proneness**, **Secondary Notation**, **Closeness of Mapping**, and **Consistency**. This indicated that these **CDs** were more sensitive to interpretation by **DSL** developers and users. Such result becomes a problem when the **CDs** instantiation was made to provide one step to overcome this obstacle. However, even though there were divergent answers, all these instantiations had at least one strong agreement answer (answer 5). Therefore, for such cases, we decided to follow our interpretation and just do the refinement. We think that this is how we will reach a consensus. The only weak agreement (majority of the answers were below 3) we had on the **CDs** instantiation was regarding **Premature Commitment**. Although this **CD** did not obtain a higher score than a medium agreement (answer 3), all participants said that this cognitive dimension cannot be used to evaluate **DSLs** for our research domain. This indicated that this **CD** is unfit to evaluate **DSLs** for our research domain, regardless of its interpretation. Therefore, we changed it taking into consideration the participants' answers, but we still considered it unfit to be used. Finally, we performed an open interview with two participants to analyze two **DSLs** with our instantiation of the **CDs**. We confirmed during the analysis that in some cases they were not sure about their answers. For example, in the case of the Viscosity and Abstraction **CDs**, it was hard for the participants to identify which **DSL** was better. This issue is further discussed in Section 3.6.

All this information indicated that a qualitative evaluation is highly dependent on the participants' experience, knowledge, and point of view. We believe that an ideal solution for the developers or new users of **DSLs** is to have a less subjective means to support an initial analysis of these **CDs**. If, for example, the developers could rely on a metrics suite, they would derive precise

information, and thereby help the early assessment of DSLs. This would in turn help to support their qualitative evaluation, so that DSL developers and users can identify why a particular DSL may be unacceptable. In this way, DSL developers and users are able to pursue appropriate maintainability.

Table 3.5: Instantiation agreement

	A	B	C	D
Viscosity	5	5	5	5
Visibility	5	5	4	3
Premature Commitment	2	3	3	2
Hidden Dependencies	4	5	5	4
Role-Expressiveness	2	2	5	5
Error-Proneness	3	5	3	2
Abstraction	4	5	5	5
Secondary Notation	1	5	5	5
Closeness of Mapping	3	3	5	2
Consistency	1	5	3	2
Diffuseness	4	5	5	5
Hard Mental Operations	4	5	5	4
Provisionality	3	3	5	4
Progressive Evaluation	5	5	5	5

3.4 Metric Definition

Once we refined the CDs (subsection 3.2.2), we decided to use the Goal, Question, Metric (GQM) methodology [40] to help us support and create a metrics suite. We followed the GQM methodology by implementing the three steps: (1) we defined our goal, (2) we created questions to define our goal as completely as possible in DSL usability, and (3) we created metrics for each question with the CDs instantiation (subsection 3.2.2). Following the described steps above, we identified our goal and its characteristics as shown in Table 3.6.

During the GQM methodology, we noticed the need to specify the two questions we wanted to answer (questions Q1 and Q2 in Table 3.6). This happened because these questions use general definitions that might lead to ambiguous or different interpretations. These definitions are: (1) **DSL Expressiveness**, which refers to the extent a domain-specific language allows to directly represent the elements of a domain, and (2) **DSL Conciseness**, which refers to the economy of terms without harming the artifact comprehension. Therefore, we subdivided each question in two sub questions that we think capture the properties of those definitions. For the sub questions Q1.1 and Q1.2 in Table 3.6, we have drawn on the definition of DSL Expressiveness. We identified that (1) the Expressiveness of a DSL must allow to express all the necessary logic for a given domain problem [12], which in other words would be the number of possible representations to be expressed (question Q1.1) and (2) the representations must have a sufficient level of abstraction to be able to solve domain problems [46] (question Q1.2). For the sub questions Q2.1 and Q2.2 in Table 3.6, we have drawn on the definition of DSL Conciseness. We identified that conciseness of a DSL should express all the domain statements adequately. In other words, the DSL rep-

Table 3.6: GQM instantiation

Goal	Purpose Issue Object Viewpoint	Identify the DSL usability limitations from the user's viewpoint	Measured Cognitive Dimension
Question	Q1	What is the DSL expressiveness?	
	Q1.1	How many representations can be expressed?	Hidden Dependencies
			Role-Expressiveness
	Q1.2	What is the level of abstraction that can be represented?	Abstraction
			Closeness of Mapping
Question	Q2	What is the DSL conciseness?	
	Q2.1	What is the number of Elements and Concepts to create/change the DSL specification?	Viscosity
			Diffuseness
	Q2.2	How fragmented is the DSL specification?	Visibility
			Hard Mental Operations

representations must be concise as possible without causing the user to misunderstand them [12]. Thus, following the representations of our DSLs defined in the metamodel (subsection 3.2.1) as Elements and Concepts, the DSL Conciseness becomes the number of Elements and Concepts necessary to create or modify a DSL specification (Question Q2.1). In addition, we identified that the DSL Conciseness can be influenced by the fragmentation of DSL specification [12, 46], thus influencing the user's understanding (Question Q2.2). Finally, we selected the instantiation of the CDs and divided them into two groups: CDs related to DSL Expressiveness and those related to DSL Conciseness (Figures 3.2 and 3.3). It is important to point out that we did not use certain CDs to analyze expressiveness and conciseness. We made this decision due to the following reasons: (1) some of the discarded CDs were already addressed by other dimensions that we are considering (Error-Proneness and Provisionality in Figure 3.2), and (2) some CDs are not applicable to

our investigation, to evaluate expressiveness and conciseness, as aforementioned.

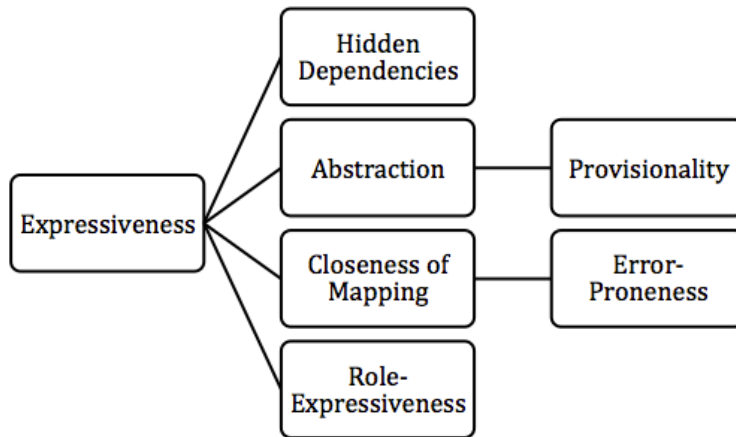


Figure 3.2: Cognitive dimensions of Expressiveness

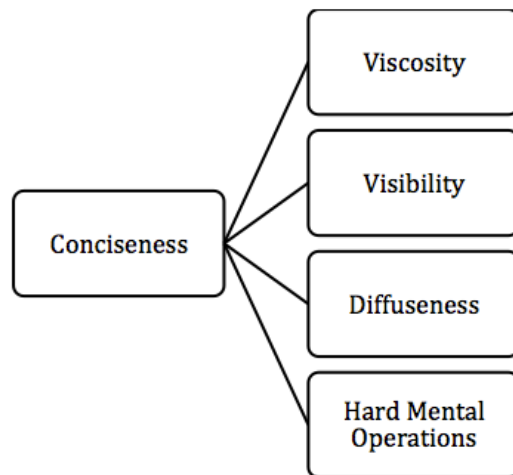


Figure 3.3: Cognitive dimensions of Conciseness

Tables 3.7 and 3.8 present the **CDs** considered in our study, with their respective descriptions. Table 3.7 shows the **CDs** related to **DSL Expressiveness**, while Table 3.8 shows the **CDs** related to **DSL Conciseness**. These tables also show the metrics we propose to use for each cognitive dimension. To help understand the proposed metrics, we first explain what qualities we expected from the **DSL metrics**. The following criteria for **DSL metrics** were:

- To be able to represent the characteristics from the **CDs**.
- To have as few parameters as possible to make the evaluations straightforward and the results comparable.

- To be clear, easily understandable by the DSL developers or users.
- To be general enough to allow comparison of most DSLs of our domain.
- To be few in number and yet expressive, so they may be used in large evaluations of DSLs.

As aforementioned, each metric was based on the interpretation of the CDs (subsection 3.2.2) for usability in DSLs, the characteristics represented in the metamodel (subsection 3.2.1) and the qualities expected from the DSL metrics. For each CD we created a unique and direct metric. One reason for that was the need to create a metric that was directly linked to the interpretation of the CD and that used the metamodel entities such as Element or Concept, in order to improve the comprehension of future data analysis. For the metric created for Abstraction, we considered that any abstraction would be our Element (abstract entity in the metamodel) by the number of Concepts. Therefore, the DSL developers and users can know how many abstractions exist and how these abstractions are scattered in the DSL specification.

For the metric for **Closeness of Mapping**, we considered that the conceptual domain is the relationship between the number of Concepts that DSL developers and users can observe and the actual number of Concepts actually used. So the DSL developers and users can visualize quantitatively the Concepts he is working on (conceptual domain). Finally, the metric for **Hard Mental Operations** was considered in such a way that all cognitive operations that the DSL developers and users had to do during the DSL specification were the number of Elements belonging to each Concept. Therefore, it may be possible to understand the mental effort realized by the DSL developers and users during the DSL specification.

Table 3.7: Expressiveness metrics (for question 1)

Cognitive Dimension	Description	Metric
Hidden Dependencies	Unexpressed dependencies between different parts of the DSL specifications	The number of unexpressed dependencies (ideal measure=0))
Role-Expressiveness	How many representations can be used to express the purpose of a Concept in a DSL specification	Number of used representations for the Concepts against number of possible representations for the Concepts
Abstraction	The number of abstractions the developer must use or create to define a Concept	The total number of Elements that are required to be described against the number of Concepts
Closeness of Mapping	How close the DSL specifications are to the architectural conceptual domain	The number of Concepts that are required to be described against the real number of Concepts that need to be described and understood

Table 3.8: Conciseness metrics (for question 2)

Cognitive Dimension	Description	Metric
Viscosity	The amount of necessary changes in the DSL specifications to adapt it for a different use	The number of Elements of the specification that must be modified in each Concept and its dependencies (ideal measure =1)
Visibility	How easy it is to visualize related portions of the DSL specifications	The number of files and how many Concepts are in the specification
Diffuseness	How many Elements are necessary to define the DSL specification	The necessary number of Elements in the specification to do the Concepts and its dependencies
Hard Mental Operations	Operations that require the developer to think about many DSL notations and abstractions at the same time	Average number of Elements in each Concept

3.5 Quantitative Evaluation

After we created the metrics suite necessary to conduct an assessment of its usefulness, we selected two DSLs to perform the quantitative study using two applications. The purpose of this evaluation was to complement the qualitative study performed earlier. The qualitative evaluation presented in Section 3.3 was mainly targeted at assessing the adequacy of the CDs instantiation. However, the participants were also invited to judge the adequability of the metrics definition. In the following, we describe the goal (subsection 3.5.1), the selected DSLs to be evaluated (subsection 3.5.2), the selected applications with architecture rules represented using those DSLs (subsection 3.5.3) and their versions (subsection 3.5.4).

3.5.1 Evaluation goal

The study goal was to assess the usefulness of the proposed quantitative framework (metrics suite) for the usability evaluation of DSLs in software maintenance tasks (Section 3.4). Software maintenance tasks were applied to produce the applications' versions, and usability metrics were applied to these versions. The assessment focused on analyzing to what extent the proposed metrics were useful: (1) to perform an early identification of DSL usability limitations that should be addressed (subsection 3.5.2), and (2) to perform a usability comparison of DSLs designed to address the same software engineering problem. We analyzed whether the metrics helped to reveal when each of the DSLs should be employed, according to particular project settings (subsection 3.5.3). We also checked whether the usability metrics and evaluations results were useful to support an indepth analysis of expressiveness and conciseness (subsection 3.5.4).

The study was conducted in the context of two DSLs recently designed to support the detection of architecture degradation symptoms (subsection 3.5.2). The comparative assessment of these DSLs was based on their use in the versions of two systems (subsections 3.5.3 and 3.5.4). As mentioned above, artifacts were produced for each version as we were interested to assess their

usability in the context of software maintenance. This procedure was required to enable us to better quantify and understand the impact of these DSLs in terms of their expressiveness and conciseness. This impact was observed as the architectures and implementations were changed along the version history of each system. Some of the CDs, by definition, require exposing the DSL artifacts to change. This is the case, for instance, of viscosity and hard mental operations (subsection 3.2.2).

3.5.2 Target DSLs

We defined some criteria in order to select the DSLs in the domain of architecture rules checking. The chosen language should support the handling of architecture design rules by at least three categories of stakeholders: (1) architects, who use it to define and maintain the high-level software architecture rules; (2) programmers, who use it to specialize the rules in terms of source code elements realizing them; and (3) reviewers, who read the rules' specification to check whether the source code developed by the programmer violates any architectural rule. It is hard to design a language that is usable by all these different types of stakeholders, while satisfying all the usability dimensions. Unified DSLs supporting all forms of anti-degradation rules are just emerging [4, 9, 14]; i.e. the designs of these languages are still in progress (subsection 2.1). This fact implies that it is unlikely that a single DSL addresses well all the usability dimensions. Therefore, they would benefit from early usability indicators before being either adopted in mainstream projects or assessed in controlled experiments.

We chose two DSLs explicitly designed with all the aforementioned categories of stakeholders in mind. The majority of the other DSLs in the field are mostly dedicated to programmers as they rely on syntaxes of programming languages (subsection 2.1). In addition, the chosen DSLs explicitly embed constructs to define both anti-drift and anti-erosion rules. The first DSL chosen is called DETEX, an instantiation of the method DECOR, supported by the Ptidej tool [9, 14]. DETEX allows the specification and detection of code anomalies, which are relevant to high-level designs. This DSL is well documented and it was already evaluated with respect to their usefulness to detect architecture anomalies in real software projects [18, 19]. The second DSL is called TamDera [4, 44], which is also fully documented [4], and relies on a robust backend infrastructure, called Vespucci [33]. This DSL was also evaluated to detect architectural anomalies in existing software projects [4, 44].

3.5.3 Target applications

We selected two applications for which it was possible to explore all (or almost all) of the constructs and mechanisms of the DSLs. We therefore looked for applications with a wide range of well-known architecture degradation symptoms. They should be from different domains, realize different architecture styles, and be designed by different developers. We also chose systems whose full set of architecture design rules were accessible. These rules should preferably be available to the community so that other researchers could replicate and extend our usability quantitative study in the future.

Based on these criteria, we selected two systems: MobileMedia [1] and Health Watcher [3]. Health Watcher is a web system for registering complaints about health issues in public institutions [4]. MobileMedia is a product line that manages different types of media on mobile devices [4]. The former realizes the N-tier architecture style, while the latter implements the MVC style. These

projects were already used in other architectural degradation studies, and their drift and erosion anomalies have been reported elsewhere [4, 5].

3.5.4 Selection of the versions of the target applications

Both DSLs are intended to improve the maintainability of design rules. Therefore, we evaluated their use through several versions of the two target systems (subsection 3.5.3). The exposition of design rules to changes would enable us to assess to what extent they satisfy the usability dimensions in the presence of several changes. In addition, the evaluation of dimensions – such as viscosity, abstraction and hard mental operations – can be assessed with higher confidence when observing them upon actual changes, rather than estimating them based on single-version specifications.

We considered all the versions of Health Watcher and Mobile Media. After their analysis, we selected a subset of them to present their results here. We focused on presenting three versions of each system: (1) versions 1, 4 and 8 of Health Watcher, and (2) versions 1, 4 and 7 of MobileMedia. We named these versions as HW1v1, HWv4, HWv8, MMv1, MMv4 and MMv7, respectively. These versions are those that suffered from the most widely scoped changes in both implementation and architecture artifacts along the system’s evolution [5]. The other versions entailed minor or none architectural-level changes.

We relied on the architecture documentation of both systems (available for the chosen versions) in order to produce the rule specifications with TamDera and DETEX. In addition, from the work of Macia et al. [5] we obtained a list of architectural anomalies that were reported by the developers for each version of each system. Based on the list of reported anomalies, we wrote additional rules of architectural anomaly detection with each DSL considered in our study. The entire specification with the DSLs was carefully written to use proper constructs of both DSLs analyzed.

3.6 Data Analysis and Discussion

Aiming to compare the DSLs and to assess the usefulness of the proposed quantitative framework, we applied the usability metrics to the TamDera and DETEX specifications. The metrics were computed for each version of the Health Watcher (HW) and MobileMedia (MM) specifications with both DSLs. The results are presented in subsection 3.6.1. The discussion about the usefulness of the metrics and other findings with respect to our research goals (subsection 3.5.1) are discussed in the following subsections. During the discussion of a particular CD, we highlight the CD name in **boldface** in order to facilitate the identification of points for discussion in the results and other broader discussions. The conceptual elements of the DSLs are also capitalized to facilitate reading. Note that, during the following data analysis, we report findings of the qualitative evaluation (Section 3.3) where we found relevant to do so. The combination of quantitative and qualitative evidence provides a more convincing result on the usefulness of the DSL usability metrics.

3.6.1 Usability measures

The results shown in Tables 3.9 and 3.10 represent the obtained measures related to expressiveness and conciseness, respectively. For every cognitive dimension, parentheses and slashes are used for direct representation of the metric results, in the format indicated by the first column.

Their use also facilitates the understanding of the **CDs** values in a proportional manner. For example, the measure for the **CD Abstraction** is presented in contrast to the number of Elements to the number of Concepts.

The metric for each **CD** was applied to the specifications based on both **DSLs**, i.e. TamDera and DETEX (in the first and second sub-line in the second column, respectively). The columns 3-8 represent the results for each usability metric through the six versions of our case studies: three versions of HealthWatcher (versions v1, v4, v8), and three versions of Mobile Media (versions v1, v4, v7). All values of the versions of HW and MM represent absolute measures with respect to the size of a specification. The measure for **Viscosity** in version 1 of both systems (Table 3.10) has no data (-) because this metric is obtained from the analysis of Elements changed from one version to the next one. The metric **Hard Mental Operations** (Table 3.10) is the average of Elements per Concepts by specifying the Concept of architectural detection rules.

Table 3.9: Results of the Expressiveness metrics

Cognitive Dimension	DSL	HWv1	HWv4	HWv8	MMv1	MMv4	MMv7
Hidden Dependencies (Unexpressed dependencies)	TamDera	0	0	0	0	0	0
	DETEX	0	0	0	0	0	0
Role-Expressiveness (Concept Representations/ Total # of Concepts Representations)	TamDera	(P,C/4)	(P,C/4)	(P,C/4)	(P,C/4)	(P,C/4)	(P,C/4)
	DETEX	(C/5)	(C/5)	(C/5)	(C/5)	(C/5)	(C/5)
Abstraction Elements/ Concepts)	TamDera	(20/11)	(28/17)	(39/28)	(20/11)	(20/10)	(19/11)
	DETEX	(38/11)	(46/17)	(61/28)	(36/11)	(37/10)	(33/11)
Closeness of Mapping (Concepts in File/ Total # of Concepts)	TamDera	(11/18)	(17/26)	(28/38)	(11/19)	(10/18)	(11/19)
	DETEX	(11/11)	(17/17)	(28/28)	(11/11)	(10/10)	(11/11)

A first analysis of Tables 3.9 and 3.10 reveal that both **DSLs** share similar positive results. These results confirm that, even though both of them were recently designed, they already satisfy a wide range of usability dimensions. For instance, it was not observed a single feature of these languages leading to **Hidden Dependencies** in both systems (Table 3.9). It is of major importance for a **DSL** not to yield hidden dependencies in order to support developers with a higher degree of control of the **DSL** specifications. In the domain of architecture rules, dependencies can be

classified in two categories: (i) dependencies between high-level rules, and (ii) dependencies between a high-level rule and the counterpart programming elements in the source code that should realize them. We noticed that both DETEX and TamDera provide abstractions to explicitly define these types of dependencies. On the other hand, we observed the metrics were also useful to reveal particular usability strengths and weaknesses of each DSL. This information is discussed in the next subsection.

Table 3.10: Results of the Conciseness metrics

Cognitive Dimension	DSL	HWv1	HWv4	HWv8	MMv1	MMv4	MMv7
Viscosity (Elements)	TamDera	-	5	2	-	7	3
	DETEX	-	10	3	-	12	8
Visibility (Files/ Concepts)	TamDera	(3/ 11)	(3/ 17)	(3/ 28)	(3/ 11)	(3/ 10)	(3/ 11)
	DETEX	(1/ 11)	(1/ 17)	(1/ 28)	(1/ 11)	(1/ 10)	(1/ 11)
Diffuseness (Elements)	TamDera	20	28	39	20	20	19
	DETEX	38	46	61	36	37	33
Hard Mental Operations (Average Elements per Concept)	TamDera	1.8	1.6	1.4	1.8	2	1.7
	DETEX	3.5	2.7	2.2	3.3	3.7	3

3.6.2 Early indicators of usability strengths and weaknesses

Tables 3.9 and 3.10 also highlight some differences between the two DLSs. The results of the second cognitive dimension analyzed, **Role-Expressiveness** (Table 3.9), reveal that TamDera support four representations (packages, classes, interfaces and methods) and DETEX offers five (classes, interfaces, methods, fields, or parameters). Therefore, a first reaction would lead us to conclude that DETEX outperformed TamDera for this CD. Even though the metric was useful to highlight a usability difference between the two DSLs, a higher (or lower) value does not always indicate a better (or worse) usability. Although DETEX has more representations than TamDera, some of them seem to be rarely used for detecting architecture degradation symptoms. For instance, fields and parameters were never used in the definition of architecture rules in both case studies. On the other hand, we noticed that packages, as supported in TamDera, were often required to express architecture rules in both systems, as developers often decompose packages in terms of architectural Concepts.

These observations may lead to two interpretations: (i) DETEX specifications might be harder to be used by software architects as there are language representations rarely useful or meaningful for them, or (ii) DETEX might be interesting to be used in projects where the architecture- and implementation-level design rules are specified by the same developers. We can also conclude that, even though DETEX has more representations to express the Elements of architectural rules, TamDera offers more representations to the architects when it comes to defining anti-erosion and anti-drift rules. This conclusion was also reported during the assessment of this CD in the qualitative evaluation when one of the participants reported that creating architectural rules on the level

of packages and classes is enough. The participant said that: “I always thought that specifying rules at the class level was enough. Specify rules at class level is complex. So, specify rules at a fine level of granularity as, for example, in a method level, it is even more complex and challenging”. Therefore, this specific participant believes that creating architectural rules in lower levels of granularity (e.g. method, fields, and parameters) might be not simple or practical.

The metric for quantifying **Abstraction** indicates the ratio of Elements per Concept in TamDera is much smaller than in DETEX for both systems. When we analyze all the measures across all the versions, the superiority of TamDera was evident ranging from 90% (HW v1) to 95% (MM v4). This difference reveals that users often need to understand and use many more Elements (per architectural Concept) in DETEX to define an architecture rule. Examples shown in Listing 3.1 and in Listing 3.2 illustrate why TamDera outperforms DETEX in terms of Abstraction. When users are expressing architecture rules, they should focus on the rules of a particular context and on reusing general rules applicable to that particular context, abstracting away from other details. TamDera offers the reuse option (Listing 3.1, line 1), and even the possibility to change the reused Elements easily in the Concept. In DETEX, we need to replicate the same rules in each Concept all over again. Also, it should be noted that the metric for Abstraction led to the same conclusion as the qualitative assessment for this CD by the participants. This demonstrates that the metric is aligned with the expectations of the developers or users.

Listing 3.1: Rules for Business Facade in TamDera

```
1 Concept BusinessFacade extends BusinessLayer
2 {
3 name:"healthwatcher.bussiness.HealthWatcherFacade"
4 LOCM < 20
5 }
```

In Table 3.10, the metric for **Viscosity** reveals that the ratio of Elements per Concept (from one version to another) requiring changes by the users is lower in TamDera than in DETEX. The measures indicate that the users would need to make from 50% to 100% more modifications in HW, and reaching the range of 71% to 167% additional modifications in MM through all the versions. These observations create a problem when users want to minimize the time and effort spent when expressing architecture rules. A key reason to explain this fact is the TamDera’s possibility to create possible rule extensions for a Concept.

Listing 3.2: Rules for Business Facade in DETEX

```
1 RULE_CARD: BusinessFacade{
2 RULE: UnionregExpr {INTER CLASSExpr FacadeConstraints NoDeeperInheritanceTree
   LOCLayer};
3 RULE: CLASSExpr {(LEXIC: CLASSE_NAME{ healthwatcher.business.HealthWatcherFacade})};
4 RULE: LOCMConstraint {(METRIC: LOCM, INF, 20.0)};
5 RULE: NoDeeperInheritanceTree {(METRIC: DIT, INF_EQ, 5.0)};
6 RULE: LOCLayer {(METRIC: LOC_CLASS, INF, 600.0)};
7 };
```

For example, in Listing 3.1 (line 1) represents a TamDera extension of a Concept named BusinessLayer. With this extension mechanism available in TamDera, the user reduces the effort of changing some Elements within each reused Concept. This effort reduction happens because, in TamDera, whenever the user modifies something in the extended Concept, such change(s) will be implicitly inherited by the different Concepts that extend the former. The same specifications of

this extension in DETEX are represented in Listing 3.2 (lines 5-6). When realizing the same behaviour in DETEX the user would need to change all the Elements where the Concept BusinessLayer is used. Therefore, the extensibility of TamDera rules for a Concept via inheritance enables the users change fewer Elements in every edition when expressing architecture rules. In addition, both programmers and code reviewers would need to read more terms when understanding the architectural rules in each Concept. Or at least, this information should be notified to them in some way. This metric has also shown its added value if we consider the qualitative evaluation of the DSLs usability using this CD. This happened because the two participants, who took part in the qualitative evaluation of the DSLs, reported different answers for the Viscosity CD. One participant said that both DSLs were similar and the other said that the Viscosity in DETEX was higher. They also confirmed that it is hard for the participants to analyze each DSL in terms of this CD. Finally, they confirmed that it would be beneficial to have a metric to support that analysis. Therefore, they could rely on some concrete information with this metric to perform this CD analysis.

The **Diffuseness** measurements indicate that DETEX requires more Elements than TamDera in order to specify the same set of architectural rules. The number of Elements the user needs to create in order to define architecture rules are from 56% to 95% higher in DETEX than in the TamDera specifications. In particular cases – i.e. v4 to v8 of MM – there is an increase of 95% of Elements in DETEX. According to this point of view, this difference represents considerable effort spent by the user when changing the set of architecture rules. Once again, this difference can be explained by the rule extensions supported by TamDera mechanisms. For example, an analysis of Listing 3.1 shows that TamDera requires fewer Elements to represent the Concept BusinessFacade. TamDera only needs two Elements (Listing 3.1, lines 3-4) and one extension, in comparison to five Elements required by DETEX (Listing 3.2, lines 2-6). This metric has also shown its added value if we consider the qualitative evaluation of the DSLs usability using this CD. Two participants, who took part of the qualitative evaluation of the DSLs, reported different answers for the Diffuseness CD. One participant reported that both DSLs were similar and the other said that the Diffuseness in DETEX was higher than in TamDera. They confirm that it is hard for the participants to analyze each DSL in terms of this CD, and confirmed that they would benefit from a metric to support that analysis. Based on this metric, they could rely on some concrete information to perform this CD analysis.

The metric for **Hard Mental Operations** shows that TamDera, on average, requires fewer Elements per Concept than DETEX. The measures indicate the user needs to reason, on average, about 1.2 (HW) to 1.5 (MM) additional Elements per Concept. This value means the user needs to reason up from 57% to 94% more Elements for each mental operation based on a DETEX specification. This difference makes it in turn much easier to perform upfront specification or maintenance operations in TamDera. This difference is already high if only the Concept is considered. However, if you consider the total number of Elements needed to express architecture rules, the difference can become even more significant. In particular, we have observed that the maintenance tasks often required the change of more than one Concept in both HW and MM. Once again, we can observe this in Listing 3.1 (lines 3-4), where the user just needs to think about two Elements in contrast to five Elements in the counterpart DETEX specification (Listing 3.2, lines 2-6). This metric has also shown its added value if we consider the qualitative evaluation of the DSLs usability using this CD. We observed, for instance, that one of the participants really struggled to analyze this CD and he was not sure what to answer. He confirmed that this CD is not easy to identify and that he would benefit from a metric to support the analysis of this CD. Based on this metric, he could rely on some concrete information to perform this CD analysis; an information to take as a starting point.

In Table 3.9, the metric for quantifying **Closeness of Mapping** indicates that the number of Concepts for expressing architecture rules is the same as the total number of Concepts actually being used in DETEX. It is different in TamDera because its feature for reuse can use Concepts of other files to express the architecture rules. The measures reveal that the TamDera user only expresses from 61% to 74% in HW and from 56% to 58% in MM of all possible Concepts. This fact can become a relevant problem for the user when reusing Concepts, because he does not have total control of all the Concepts when expressing architecture rules. For example, during the qualitative evaluation of the DSLs usability with this CD, a participant reported the results of this CD has revealed he was potentially wrong. He noticed that he had actually misinterpreted the meaning of the CD. He agreed the metric captured a useful quantifiable property associated with Closeness of Mapping. Therefore, this metric can provide a hint to the user on the understanding of the level of control of the Concepts he has during the DSL specification.

The results of the metric employed for quantifying **Visibility** are presented in Table 3.10. This results reveal that it is easier to visualize or change the DSL specifications in DETEX, because all the Concepts are in the same file. The measures show that the TamDera user has expressed the architecture rules in three files, against one file in DETEX. This represents several problems for the user's cognitive performance because the user is forced to change to different files in order to visualize some Concept. Once again, it is possible to notice in Listing 3.1 (line 1) that there is an extension of a Concept from another file in TamDera specification, whereas no additional files are required in Listing 3.2 for the counterpart DETEX specification. Also, it should be noted that the metric for Visibility led to the same conclusion that the qualitative assessment for this CD by the participants. This demonstrates that the metric is aligned with the expectations of the developers or users. However, it is important to remember that the interpretation of the metric may vary. The reason for that is because of the user or the developer goal. For example, if one user defines that the visibility for some DSL is having fewer files because their perception is better this would mean that the lower the metric value, the better the visibility of a DSL.

3.6.3 Usability in specific project settings

This subsection discusses our lessons learned on the use of our evaluation framework (Section 3.4) to support the expressiveness and conciseness analyzes.

Expressiveness analysis

An evaluation taking into consideration of the DSL expressiveness requires a joint analysis of the four CDs related to this attribute in order to lead us to broader and fair conclusions. The CDs Hidden Dependencies, Role-Expressiveness, Abstraction, and Concept Mapping together identify the level of expressive power that a language offers to represent the different rules of a DSL. It is not possible to analyze the expressiveness of a DSL without thinking of the four CDs together, because one dimension complements the others. More importantly, expressiveness trade-offs are revealed when all these attributes are analyzed.

For example, the CDs Hidden Dependencies and Role-Expressiveness together show the level of control that a user has when implementing architectural rules. Someone could infer that TamDera has superior expressiveness, given the higher number of supported representations to express the architecture-level rules. This is somehow confirmed by the Abstraction metric. However, when the metric for Closeness of Mapping is considered, it becomes clear that the superior Abstraction and Role-Expressiveness of TamDera comes at a cost: more files need to be created,

understood and maintained by architects, programmers and code reviewers. Moreover, we believe that this indicates that the communication between them needs to be improved in order to obtain a better analysis.

Conciseness analysis

The **CDs** Viscosity, Visibility, Diffuseness, and Hard Mental Operations together indicate the level of conciseness that a language offers to represent the different rules of a **DSL**. The joint analysis of all conciseness **CDs** led us to infer other interesting findings. For example, the TamDera reuse mechanisms are consistently the key factors to support more concise specifications of architectural rules. The conciseness benefits in TamDera tend to increase for all the metrics as new versions are generated. We also have noticed that Diffuseness plays a central role in conciseness evaluation. By reducing the number of Elements required to write a Concept, all other conciseness **CDs** would be influenced. For example, if one reduces the Diffuseness degree in DETEX or TamDera, the number of Hard Mental Operations would also be reduced.

Expressiveness vs. Conciseness

To improve the expressiveness or conciseness of a **DSL**, both **CDs** groups must be considered. In our analysis, we have noticed that the improvement of a group influences the other. For example, if in DETEX we improve the Closeness of Mapping by sharing different Concepts for some files, this will worsen the cognitive dimension Visibility but, in contrast, the cognitive dimension Hard Mental Operations will improve. This leads us to the conclusion that it is not possible to improve the expressiveness and conciseness of a **DSL** without considering the other **CDs** group. So, it is necessary to define the objectives of the **DSLs** or to set a degree of balance between the two groups before starting to develop new characteristics of a **DSL**.

Suitability of the **DSLs** to different project settings

The use of different versions of HW and MM allowed us to infer circumstances in which it is better to use each of these two **DSLs**. For projects with a few versions dominated by stable, non-reusable rules or with a small set of architectural rules, the use of DETEX seems more advisable than TamDera. The former allows the specification of the architecture rules in a single file, which is easier to learn by different stakeholders. Nevertheless, TamDera seems to be more advisable for large systems with many architectural rules or in projects with many planned versions. TamDera has also shown superior usability in cases where the **DSL** specification involves similar Concepts, where the differences can be expressed by inheritance and compositional reuse [4]. TamDera enables a better organization of rule specifications per Concept in different files. The language also allows stakeholders of different projects to work without needing to redefine rules or Concepts from the scratch or modify existing files from one project to another.

3.7 Study Limitations

In our study, a first limitation is related to the operational definition of the metrics to analyze the usability of the **DSLs**. To reduce the influence of this limitation, we proposed a metamodel that identifies the most important entities of a textual **DSL** in order to propose metrics based on this

metamodel. It is worth to notice that, despite our metamodel not being instantiated for several DSLs used for detecting architectural anomalies, most of the entities of the metamodel are found in DSLs that support a subset of either anti-drift or anti-erosion rules [4]. The metamodel can also be extended in the future in case DSLs in this domain evolve, thereby allowing a better evolution of the usability metrics' definitions. In addition, our study protocol can be reused to assess DSLs of several architectural rule-checking subdomains.

Another limitation relies on the procedures for quantifying the values of the metrics. Since most of them needed to be extracted manually, they can be directly associated with the decisions made while extracting them. In order to ameliorate this issue, the quantification of metrics in each application was widely discussed among experienced developers before data analysis. We also consulted the designers and developers of DETEX and TamDera to address certain doubts about the language features. For instance, we needed to confirm with them the full set of representations actually supported at the moment in both DSLs.

Another threat resides on the choice of what must be analyzed in both chosen DSLs, since they have different capabilities regarding the detection of architectural degradation (subsection 3.5.2). To reduce this threat, we performed a detailed analysis of the DSLs properties in order to reduce the difference between them and chose only common characteristics of both DSLs. Threats to external validity are conditions that allow the generalization of results. To address this kind of threat, we selected applications from different domains and developed by different research groups. These applications are representative of architectural degradation and maintenance tasks, allowing us to use several constructions of the DSLs analyzed. Moreover, the applications have a significant size (subsection 3.5.3) and they embrace different types of architecture-level changes.

3.8 Summary

In this Chapter, we presented a study to compare the usability of textual DSLs under the perspective of software maintenance. We developed a usability metrics suite based on the CDN framework. We compared two textual DSLs to detect architectural problems through several versions of two evolving systems.

The main results suggested that the proposed metrics were useful to early identify the DSL usability limitations, to reveal specific features of the DSLs favoring software maintenance tasks, and to successfully analyze eight usability dimensions that are critical in many DSLs. In this context, the results obtained are evidence that the metric suite created for quantitatively analyzing the usability of DSLs supports an objective comparison between DSLs, and therefore might help to improve them and to promote their acceptance. The proposed approach can also complement qualitative analyzes approaches found in the literature. Moreover, the results also provided extra information of the tools that used those DSLs. The results of the metrics indicate that a constant communication between the stakeholders is fundamental. Therefore, tools for detection architectural anomalies need to enable constant communication between the stakeholders while developing an application. We believe this would allow a better specification of the architectural rules and faster learning of the application architecture by the programmers and code reviewers.

Chapter 4

TamDera++

Software evolution is a natural process to adapt and correct all requirements imposed by the society [53]. Parnas [57] identified that software changes are often made by people who do not understand the original design. These changes often lead the software architecture to degrade over time [2, 5, 7]. An architectural degradation usually causes in turn, a lot of effort on the realization of further code changes [62]. In worse cases, architectural degradation can lead to system discontinuity [60, 62]. In this context, research over the last years has led to the proposition of strategies to detect architectural degradation symptoms [5, 62]. One of these strategies is relying on tools that use **DSL**, which help developers and software architects specify software design rules [4, 14].

However, our study revealed that the lack of communication among the stakeholders creates a gap in the software development as well as in the usability of these tools (Chapter 3). Hence, this communication gap can compromise the architecture during the evolution process. Unfortunately current tools do not explicitly support a cycle of communication between stakeholders.

The main contribution of this Chapter is a set of proposed heuristics for enhancing tooling support for improving collaborative detection of architecture degradation symptoms. The purpose of these heuristics is also addressing the usability weaknesses on the use of existing **DSLs** (Chapter 3). The heuristics were based on the aforementioned tool, TamDera (subsection 2.2.2), a tool that leverages a **DSL** for supporting detection of architectural anomaly symptoms in the source code. We selected TamDera due to the fact that it was already designed with different categories of stakeholders in mind. Moreover, to implement the proposed heuristics on TamDera a new architecture and a new environment were created.

The remainder of this Chapter is organized in the following way: Section 4.1 describes in more detail the problem we are aiming to solve. Section 4.2 gives some background information in order to ascertain a better understanding of the scope of this Chapter. Section 4.3 describes the Chapter goal and the proposed heuristics. Section 4.4 describes the new tool environment and architecture to support the proposed heuristics. Finally, Section 4.5 concludes the work and suggests future developments.

4.1 Motivating Example

In order to get a better understanding of the scope of the work done in this Chapter and the issues that we intended to solve, we considered a scenario extracted from MobileMedia, a mobile application responsible for the managing of different types of media in mobile devices. Moreover, MobileMedia implements the Model-View-Controller pattern and was already used in other architectural degradation studies [4, 5]. Figures 4.1 and 4.2 represent the architecture defined for versions 1 and 4, respectively. For every component added or modified from version 1 to version 4 of the architecture, symbols were created for better readability, as can be seen in Figure

4.2. Their analyzes reveal that several architectural elements have been added such as **AlbumController** or even modified as in the case of **BaseController**, from version 1 to version 4. As a consequence, the counterpart program elements have been also modified to incorporate these architectural changes. In particular it is possible to see that that the component **BaseController** has almost direct communication with every component in the architecture of version 1. Moreover the **BaseController** component in version 1 requires the **NewAlbumScreen** component, however, in version 4 the same component does not even exist anymore. This shows how architectural changes cause different software design rules, described in a DSL specification, to be edited or be refined during the software evolution.

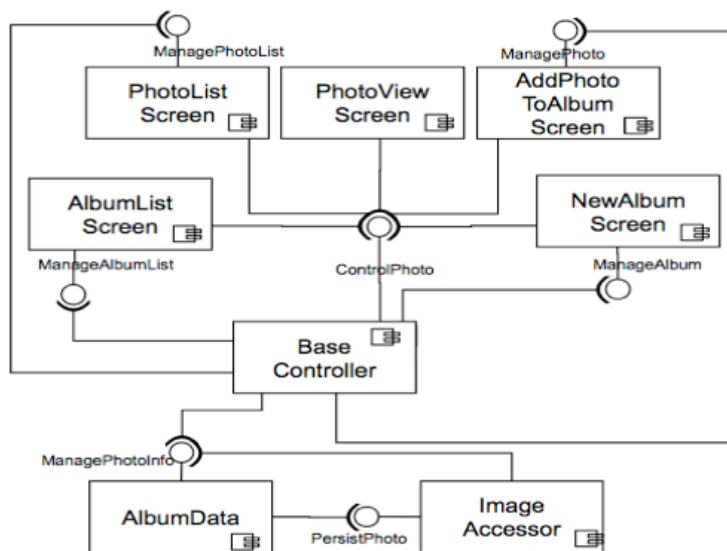


Figure 4.1: Architecture of the MobileMedia (version 1)

The study in Chapter 3 supports the notion that software evolution from one version to another influences various characteristics DSL-based specification of the aforementioned. In other words, the changes needed to be done in a DSL-based specification can be complex and time consuming as can be seen by the modified components and added in the architecture of the MobileMedia version 1 to version 4 (Figures 4.1 and 4.2). Therefore, is advised to have a control of the DSL specification during the evolution of the application. This will cause the DSL specification to be always updated in order to detect architectural degradation symptoms. Otherwise, we may be creating false positives or negatives during the detection as well as creating unnecessary effort to refactor the new code. Therefore, to have a DSL specification always updated there must be constant communication between the stakeholders during the evolution of the application. This communication is necessary because there may be some DSL specifications that are not possible to be accomplished in the application source code. This communication can serve for stakeholders to adapt the new specification with the new code more quickly but also to detect problems that may exist in the DSL specification. Subsection 4.3.1 will further discuss this subject.

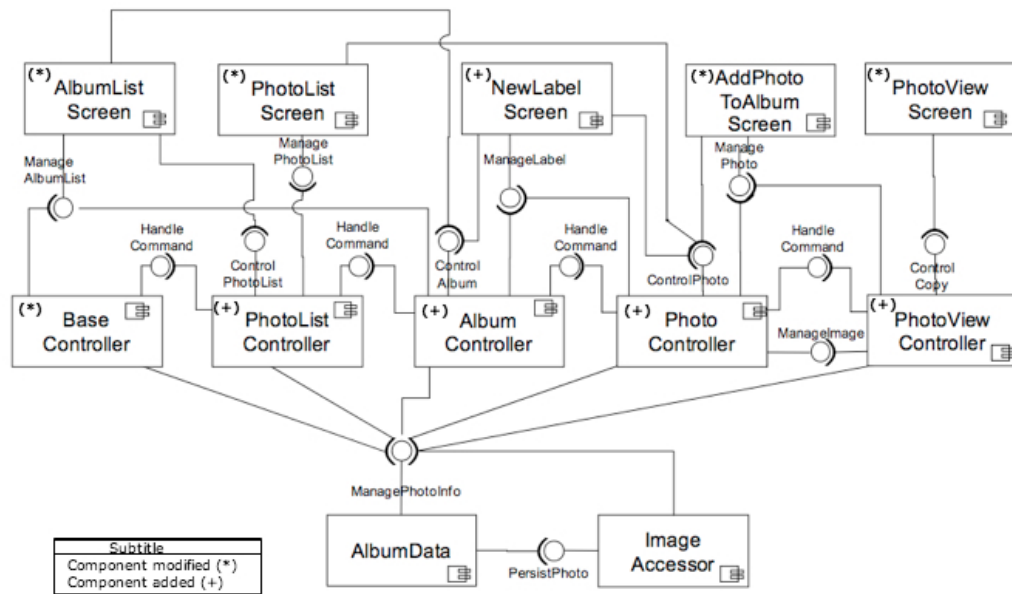


Figure 4.2: Architecture of the MobileMedia (version 4)

4.2 Limitations of Related Work

Studies [2, 5, 7], show how software architecture degradation is caused by several unintended changes throughout software evolution, known as versions. These studies bring forward the relationship between the architectural degradation and the architectural anomalies. Architectural anomalies are decisions in the source code that violate the intended architecture. Therefore, due to the necessity for strategies to detect architectural degradation symptoms, a number of tools were created. Among these tools there are ones that rely on domain specific languages (DSLs), which help developers and software architects specify software design rules [4, 14]. For example, Ptidej [14] is a tool that uses a DSL called DETEX, an instantiation of the method DECOR, which allows stakeholders to specify rules at a high level of abstraction to detect architectural anomalies. SCOOP [61] is a tool that embeds a domain-specific language to be used by the stakeholders and exploits relationships between architectural anomalies to detect the relevant ones. And finally TamDera is a tool [4] that uses a domain specific language, which allows the specification of design rules whilst promoting the hierarchical and compositional reuse of those design rules across multiple contexts. However, most of these tools do not have a cycle of communication between the stakeholders as mentioned previously. This happens because the majority of the DSLs in the field are dedicated to programmers as they rely on syntaxes of programming languages (Chapter 3). Therefore, we selected TamDera in order to extend and promote this communication cycle during each commit of a new code due to the fact that it was already designed with different categories of stakeholders in mind. Moreover, this tool has a wide capability regarding the detection of architectural degradation symptoms using its DSL as shown in subsection 2.2.2. To achieve this, TamDera DSL provides two abstractions: (1) **architectural concept**, also denoted by the keyword *Concept*, represents a relevant concern to the mind-set of software architects. These concerns can be components, interfaces, or any other decision expressed in an architecture document, and (2) **concept mapping** where each module element comprises each architectural Concept.

4.3 Study Decisions

This section describes the study goal (subsection 4.3.1) and the proposed heuristics created to improve the communication between different stakeholders (subsection 4.3.2).

4.3.1 Goal

The Chapter goal was to propose solutions for usability problems found in previous studies (Chapter 3) of tools that use DSLs for detecting architecture degradation symptoms. One problem that these tools have is that they do not support cycles of communication between stakeholders, thus creating a gap in the software development as well as in the usability of these tools. Hence, this lack of communication may cause DSLs specifications to be outdated and create false positives or negatives during the detection and creating unnecessary effort to refactor the new code. To solve this problem we proposed heuristics for tools that use DSLs for detecting architecture degradation. These heuristics will permit the exchange of information between the stakeholders, thereby, also increasing the tool usability. Finally, we chose TamDera as the tool to implement these heuristics in our study domain. This decision was made because TamDera was already designed with different categories of stakeholders in mind, as mentioned previously in subsection 2.2.2. Therefore, making it easier to implement the heuristics and everything related to it, as will be shown in Section 4.4.

4.3.2 Decisions Steps

This subsection aims to express the heuristics implemented in TamDera proposed new version¹. The explanations provide the basis on which the TamDera++ will be created. Is important to understand that the heuristics were based on information from the study made in Chapter 3 as will be shown.

Objective 1: *Inform the software architects the most violated Concepts after X number of commits of new code.*

Rationale: If we see towards the MobileMedia architecture from 1 it is possible to see that Component BaseController has different connections with other components. Thereafter in version 4 the architecture has undergone several changes to correct this situation. This indicates that the DSL specification of TamDera needs to follow the software evolution. Moreover if we take a look at the study from Chapter 3, the CD Viscosity from Table 3.10 we see that TamDera own specification changes with each new version. Therefore, the information in Objective 1 can indicate that something is wrong with one of the Concepts. In other words, the Concept being violated can indicate that a major architectural problem is occurring or that the Concept is not suitable for the architecture being developed.

Procedure: For this objective to be accomplished TamDera++ will save information of the violated Concept. Therefore, whenever a programmer commits a new code, TamDera++ will analyze whether a Concept was violated or not. Subsequently, TamDera++ will save all the information of the Concepts that were violated. After X commits TamDera++ will analyze if any of these Concepts were violated more than once during the software development. Consequently, TamDera++ provides the software architect with all the information regarding the most violated Concepts.

¹From hereafter, we use the term "TamDera++" to refer TamDera proposed new version

Objective 2: *Inform the software architects if different programmers violate the same Concept, after an X number of commits of a new code.*

Rationale: Once again in Table 3.10, the CD Viscosity indicate that TamDera own specification changes with each new version of MobileMedia. Therefore, the Concept that is being violated can indicate that the Concept is not clear enough for the programmers or the Concept is not suitable for the architecture being developed. For example there may be a Concept, which does not have the appropriate thresholds for the software architecture developed.

Procedure: In this objective TamDera++ will save the metadata of the violated Concepts and information such as the commit date and the programmer who committed it. After X commits, TamDera++ will analyze whether or not any of those Concepts were violated by more than one programmer during the software development. If this is true, TamDera++ will inform the software architect and provide information about these violated Concepts. Later, TamDera++ will give two options to the software architect: (i) it will inform all programmers who violated a Concept to conduct code refactoring or (ii) analyze the Concepts that were violated for a possible refinement.

Objective 3: *Inform the software architect if the same violated Concept come from the same programmer, after an X number of commits of new code.*

Rationale: This information may indicate that, for a particular reason, the violated Concepts are not suitable for a particular software development. Once again if we take as example the evolution of the Component BaseController in MobileMedia architecture, it is possible to see that the TamDera specification needs to be adapted. Therefore, different from the previous objectives, this particular objective prioritize and recommends the dialogue between the architect and the programmer so they may reach a solution.

Procedure: In this objective, once more TamDera++ will save the metadata of the violated Concepts and information such as the commit date and the programmer who committed it. After X commits, TamDera++ will analyze if any of these Concepts were violated more than once by the same programmer. Hence, if this is confirmed, TamDera++ will inform the software architect with information about these violated Concepts and the programmer responsible.

4.4 Implementation

This section describes the proposed environment (subsection 4.4.1) and the new TamDera architecture (subsection 4.4.2) necessary to support the proposed heuristics in subsection 4.3.2.

4.4.1 Proposed TamDera++ Environment

We created a representation (Figure 4.3) of the TamDera++ environment, to better understand the new functionalities of TamDera++ with its stakeholders.

This representation of the TamDera++ environment shows how the communication of different stakeholders is made, either indirectly through the use of different TamDera specifications over the development or by a direct communication with the use of an instant messaging service. Furthermore, the TamDera++ environment should support the handling of architecture design rules by at least two categories of stakeholders: programmers and software architects.

The **programmer** checks the source code from the [Apache Subversion \(SVN\)](#) server and introduces a new code or changes it, for then be executed by TamDera++. That way TamDera++ first uploads the TamDera specifications of the [SVN](#) server and then analyzes the source code for architectural anomalies. That information will give the developer the possibility to: (i) refactor

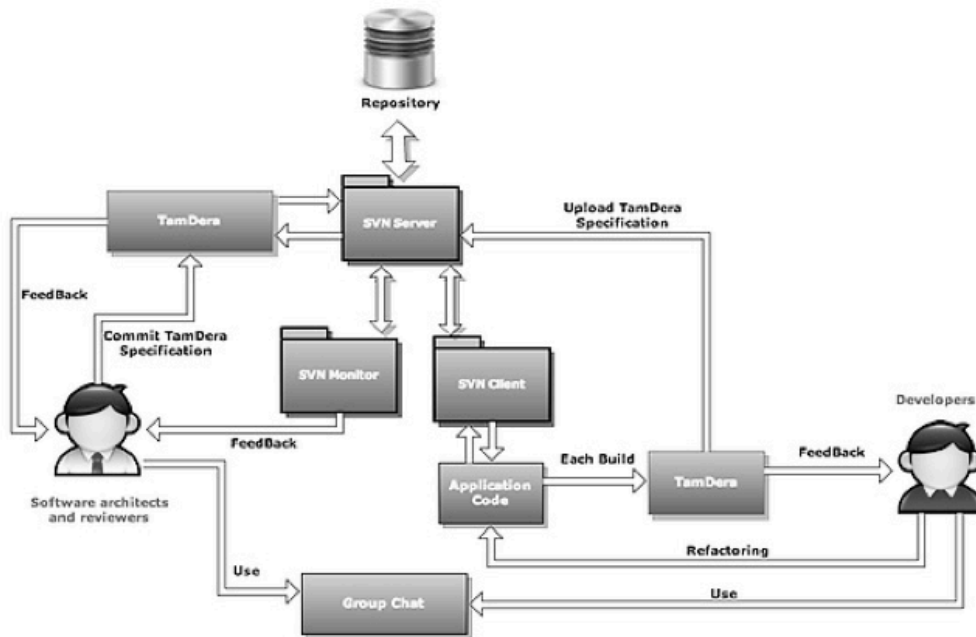


Figure 4.3: TamDera++ proposed environment

the source code, (ii) commit the source code without refactoring or (iii) with the use of an instant messaging service that provides both text and voice communication that informs, software architects of any problem or disagreement.

The **software architect** in each commit from the programmer receives notifications regarding the matter. The software architect has the option to: (i) notify the programmer responsible, (ii) modify it himself or (iii) edit the TamDera++ specification if the software architect finds that a threshold is not correct or if there is missing some specification for a component in the source code.

4.4.2 TamDera++ proposed Architecture

In this proposed new version of TamDera (Figure 4.4) we maintained the main components of the architecture, 4-8 in Figure 2.2, but we added the external tools TOGETHER and BAT as an extension of the new version of TamDera.

This will improve the usability of the tool by permitting the user not to consider anything besides the TamDera rules specification. These external tools are connected to a new component called TamDera extension, that works without the user interaction. This component provides a new interface that differentiates the users, i.e. software architects and programmers. In the interface of TamDera++, the software architect needs to provide the TamDera specification and the [SVN Uniform Resource Locator \(URL\)](#) where it will be submitted. The programmer needs only to introduce the [SVN URL](#) in order to retrieve the source code of the application and the TamDera specification (Figure 4.5).

Whenever TamDera++ verification of a violated Concept on a single commit is complete, the results are stored in an XML-database. Thereafter, every pair from the same heuristic is compared

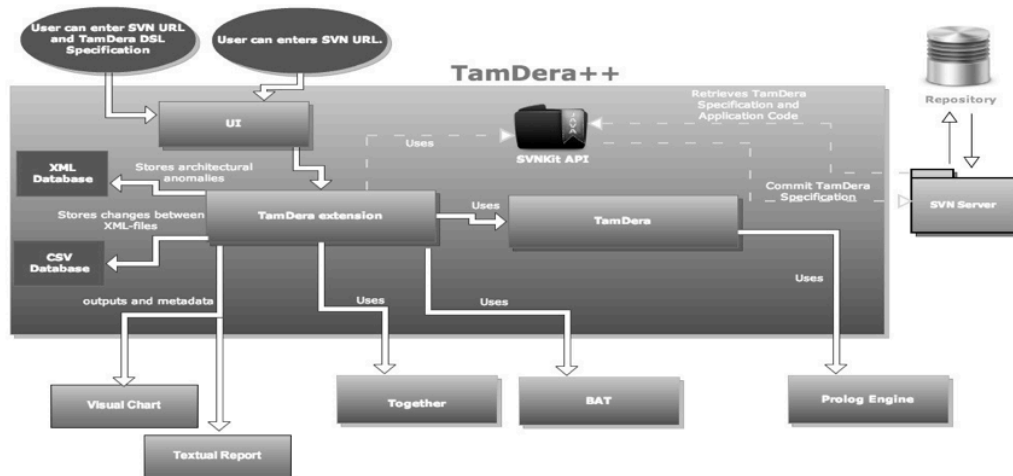


Figure 4.4: TamDera++ proposed architecture



Figure 4.5: TamDera++ User Interface

consecutively for differences in violated Concepts, thus enabling it to determine the lifespans of violated Concepts. Hence, the results of this analysis are stored in a CSV-file that will give the user the history of each commit. TamDera++ will also provide a visual chart or a textual report related to architectural Concepts in the software application. The textual report contains each architectural Concept violated and its lifespan, as well as the metadata and information such as the commit date and the programmer who committed it. Alternatively, we can generate a chart that represents the lifespans of these architectural Concepts violated.

4.5 Summary

In this chapter, we presented heuristics to improve the constant communication between the stakeholders whilst developing an application. These heuristics permit the exchange of information between the stakeholders, thereby, also increasing the tool's usability in our study domain. These heuristics were all validated due to the results of the study in chapter 3 and informal conversations with software architects. However, we believe that a future validation through an experiment should also exist. Nevertheless, these heuristics helped create a way to start thinking more about the environment and its stakeholders for software development and maintenance. Moreover, it has allowed us to know more precisely which characteristics of the TamDera tool and its environment need to be improved to support our goal in this Chapter. We also noticed that many adjustments still need to be made, which will be reflected in future work. This study has also opened doors for major developments and further research that aims to analyze code in time of the commit.

Chapter 5

Conclusion

The focus of this dissertation was to improve the usability of existing tools with [DSLs](#) for detecting architectural degradation symptoms. Therefore, as our first goal, we presented a study to compare the usability of textual [DSLs](#) under the perspective of software maintenance. We developed a usability metrics suite based on the [CDN](#) framework. We analyzed two textual [DSLs](#) to detect architectural problems through several versions of two evolving systems, HealthWatcher and Mobile Media. The main results suggested that the proposed metrics were useful in the identification of the [DSL](#) usability limitations, to reveal specific features of the [DSLs](#) favoring software maintenance tasks, and to successfully analyze eight usability dimensions that are critical in many [DSLs](#). In this context, the results obtained were evidence that the metric suite created for quantitatively analyzing the usability of [DSLs](#) supports an objective comparison between [DSLs](#). The proposed approach can also complement qualitative analyzes approaches found in the literature. Moreover, the results also provided extra information of the tools that used those [DSLs](#). These results were the base for developing the second goal of this dissertation.

Thereafter for our second goal we proposed heuristics for enhancing the tools usability that use [DSLs](#) for detecting architecture degradation symptoms. These heuristics were based of the results obtained of the evaluation from the usability metrics suite. The results of the metrics indicate that a constant communication between the stakeholders is fundamental. Therefore, tools for detecting architectural anomalies need to enable constant communication between the stakeholders whilst developing an application. We believe this would allow a better specification of the architecture rules and faster learning of the application architecture by the programmers. In order to implement these heuristics, we implemented in the new version of TamDera the communication support for the stakeholders by using a new architecture and a new environment with the developed heuristics. We believe that this will permit the exchange of information between the stakeholders to be more consistent, thereby, also increasing the tool usability. Moreover, we believe that this new version has opened doors for major developments and further research that aims to analyze code in time of commit. However, we also know that maybe some adjustments still need to be made, which will be reflected in the future work.

Finally we are happy with all the results obtained manly because we believe that we achieved all the proposed goals. We believe that this dissertation can serve as a basis for future work to be done regarding the evaluation of [DSLs](#) or in the developing and improving tools from the same domain.

5.1 Future Work

To the best of our knowledge, this is a first attempt to define an evaluation methodology for quantitatively analyzing the usability of [DSLs](#). Even though it needs further improvement and validation, we believe it supports our argument that the use of quantitative analyzes can be a valuable

approach to understand the limitations of **DSLs**. In this context, we envision several directions in which this work can evolve such as: (i) perform similar studies to evaluate the integration and performance of **DSLs**, (ii) repeat the instantiation process to evaluate **DSLs** in other domains, (iii) extend **CDN** with new dimensions to support deeper analysis of **DSL** usability, and (iv) investigate how qualitative and quantitative methods can be combined to provide a better understanding of usability in **DSLs**.

Once again to the best of our knowledge, there are no solutions within this study domain that promote constant communication between the stakeholders. In this context, we envision the following directions: (i) investigate new heuristics to promote the further growth of constant communication between the stakeholders (ii) implement the proposed heuristics in other tools of the domain, and (iii) perform experiments on TamDera++ with a software development team.

Bibliography

- [1] E. Figueiredo, A. Garcia, and C. Lucena, "Ajato: An aspectj assessment tool," in *European Conference on Object-Oriented Programming (ECOOP Demo), France, 2006*.
- [2] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pp. 255–258, IEEE, 2009.
- [3] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, *et al.*, "On the impact of aspectual decompositions on design stability: An empirical study," in *ECOOP 2007–Object-Oriented Programming*, pp. 176–200, Springer, 2007.
- [4] A. Gurgel, "Blending and reusing rules for architectural degradation prevention," Master's thesis, PUC-Rio.
- [5] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. von Staa, "Are automatically-detected code anomalies relevant to architectural modularity?: an exploratory analysis of evolving systems," in *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pp. 167–178, ACM, 2012.
- [6] R. Maia, R. Cerqueira, C. S. de Souza, and T. Guisasola-Gorham, "A qualitative human-centric evaluation of flexibility in middleware implementations," *Empirical Software Engineering*, vol. 17, no. 3, pp. 166–199, 2012.
- [7] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIG-SOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.
- [8] R. S. Silva Filho, F. Bronsard, and W. M. Hasling, "Experiences documenting and preserving software constraints using aspects," in *Proceedings of the tenth international conference on Aspect-oriented software development companion*, pp. 7–18, ACM, 2011.
- [9] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A. Le Meur, "Decor: A method for the specification and detection of code and design smells," *Software Engineering, IEEE Transactions on*, vol. 36, no. 1, pp. 20–36, 2010.
- [10] A. Van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.
- [11] E. Visser, "Webdsl: A case study in domain-specific language engineering," in *Generative and Transformational Techniques in Software Engineering II*, pp. 291–373, Springer, 2008.
- [12] B. G. Humm and R. S. Engelschall, "Language-oriented programming via dsl stacking," in *ICSOFT (2)*, pp. 279–287, 2010.

BIBLIOGRAPHY

- [13] C. Consel and R. Marlet, "Architecture software using: a methodology for language development," in *Principles of Declarative Programming*, pp. 170–194, Springer, 1998.
- [14] N. Moha and Y.-G. Guéhéneuc, "P tidej and d ecor: identification of design patterns and design defects," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pp. 868–869, ACM, 2007.
- [15] D. Lawrie, H. Feild, and D. Binkley, "Syntactic identifier conciseness and consistency," in *Source Code Analysis and Manipulation, 2006. SCAM'06. Sixth IEEE International Workshop on*, pp. 139–148, IEEE, 2006.
- [16] J. Becker, P. Bergener, D. Breuker, and M. Rackers, "Evaluating the expressiveness of domain specific modeling languages using the bunge-wand-weber ontology," in *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, pp. 1–10, IEEE, 2010.
- [17] J. Gray and G. Karsai, "An examination of dsls for concisely representing model traversals and transformations," in *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, pp. 10–pp, IEEE, 2003.
- [18] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, and F. Palomba, "Supporting extract class refactoring in eclipse: The aries project," in *Proceedings of the 2012 International Conference on Software Engineering*, pp. 1419–1422, IEEE Press, 2012.
- [19] N. Haderer, F. Khomh, and G. Antoniol, "Squaner: A framework for monitoring the quality of software systems," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pp. 1–4, IEEE, 2010.
- [20] B. Langlois, C.-E. Jitia, and E. Jouenne, "Dsl classification," in *OOPSLA 7th Workshop on Domain Specific Modeling*, 2007.
- [21] M. Anlauff, A. Bemporad, S. Chakraborty, P. Kutter, D. Mignone, M. Morari, A. Pierantonio, and L. Thiele, "From ease in programming to easy maintenance: Extending dsl usability with montages," 1999.
- [22] H. Nishino, "How can a dsl for expert end-users be designed for better usability?: a case study in computer music," in *CHI'12 Extended Abstracts on Human Factors in Computing Systems*, pp. 2673–2678, ACM, 2012.
- [23] A. Blackwell and T. Green, "Notational systems—the cognitive dimensions of notations framework," *HCI Models, Theories, and Frameworks: Toward an Interdisciplinary Science*. Morgan Kaufmann, 2003.
- [24] C. d. S. S. Neto, L. F. G. Soares, and C. S. de Souza, "The nested context language reuse features," *Journal of the Brazilian Computer Society*, vol. 16, no. 4, pp. 229–245, 2010.
- [25] C. d. S. S. Neto, L. F. G. Soares, and C. S. de Souza, "Tal—template authoring language," *Journal of the Brazilian Computer Society*, vol. 18, no. 3, pp. 185–199, 2012.
- [26] L. M. Afonso, F. d. G. R. Cerqueira, and C. S. de Souza, "Evaluating application programming interfaces as communication artefacts," *Proceedings of the Psychology of Programming Interest Group*, pp. 8–31, 2012.

- [27] A. Rainer and T. Hall, "A quantitative and qualitative analysis of factors affecting software processes," *Journal of Systems and Software*, vol. 66, no. 1, pp. 7–21, 2003.
- [28] N. Bevan, "Extending quality in use to provide a framework for usability measurement," in *Human Centered Design*, pp. 13–22, Springer, 2009.
- [29] S. Di, H. Jin, S. Li, L. Chen, and C. Wang, "Globalwatch: A distributed service grid monitoring platform with high flexibility and usability," in *Services Computing, 2006. APSCC'06. IEEE Asia-Pacific Conference on*, pp. 440–446, IEEE, 2006.
- [30] W. Drytkiewicz, S. Sroka, V. Handziski, A. Köpke, and H. Karl, "A mobility framework for omnet+," 2003.
- [31] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: a 'cognitive dimensions' framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996.
- [32] A. Van Deursen and P. Klint, "Little languages: Little maintenance?," *Journal of software maintenance*, vol. 10, no. 2, pp. 75–92, 1998.
- [33] R. Mitschke, M. Eichberg, M. Mezini, A. Garcia, and I. Macia, "Modular specification and checking of structural dependencies," in *Proceedings of the 12th annual international conference on Aspect-oriented software development*, pp. 85–96, ACM, 2013.
- [34] T. M. Pigoski, *Practical software maintenance: best practices for managing your software investment*. John Wiley & Sons, Inc., 1996.
- [35] S. S. Yau and J. S. Collofello, "Some stability measures for software maintenance," *Software Engineering, IEEE Transactions on*, no. 6, pp. 545–552, 1980.
- [36] A. April and A. Abran, *Software maintenance management: evaluation and continuous improvement*, vol. 67. John Wiley & Sons, 2012.
- [37] R. Prieto-Díaz, "Domain analysis: An introduction," *ACM SIGSOFT Software Engineering Notes*, vol. 15, no. 2, pp. 47–54, 1990.
- [38] P. Hudak, "Modular domain specific languages and tools," in *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pp. 134–142, IEEE, 1998.
- [39] V. R. Basili, G. Caldiera, and H. D. Rombach, "Experience factory," *Encyclopedia of software engineering*, 1994.
- [40] R. Terra and M. T. Valente, "A dependency constraint language to manage object-oriented software architectures," *Software: Practice and Experience*, vol. 39, no. 12, pp. 1073–1094, 2009.
- [41] N. Ubayashi, J. Nomura, and T. Tamai, "Archface: a contract place where architectural design and code meet together," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 75–84, ACM, 2010.
- [42] C. Morgan, K. De Volder, and E. Wohlstadter, "A static aspect language for checking design rules," in *Proceedings of the 6th international conference on Aspect-oriented software development*, pp. 63–72, ACM, 2007.

BIBLIOGRAPHY

- [43] 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 of the 13th international conference on Modularity*, pp. 61–72, ACM, 2014.
- [44] A. Barišić, V. Amaral, M. Goulao, and B. Barroca, "Quality in use of dsls: Current evaluation methods," *Proceedings of the 3rd INForum-Simpso de Informtica (INForum2011)*, 2011.
- [45] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.
- [46] S. Sobernig, P. Gaubatz, M. Strembeck, and U. Zdun, "Comparing complexity of api designs: An exploratory experiment on dsl-based framework integration," in *ACM SIGPLAN Notices*, vol. 47, pp. 157–166, ACM, 2011.
- [47] N. Oliveira, M. J. V. Pereira, P. R. Henriques, and D. da Cruz, "Visualization of domain-specific programs' behavior," in *Visualizing Software for Understanding and Analysis, 2009. VISSOFT 2009. 5th IEEE International Workshop on*, pp. 37–40, IEEE, 2009.
- [48] K. Bernardin, A. Elbs, and R. Stiefelhagen, "Multiple object tracking performance metrics and evaluation in a smart room environment," in *Sixth IEEE International Workshop on Visual Surveillance, in conjunction with ECCV*, vol. 90, p. 91, Citeseer, 2006.
- [49] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, no. 6, pp. 476–493, 1994.
- [50] P. Gabriel, M. Goulao, and V. Amaral, "Do software languages engineers evaluate their languages?," *arXiv preprint arXiv:1109.6794*, 2011.
- [51] A. Gurgel, "Blending and reusing rules for architectural degradation prevention.," in *ECOOP 2007—Object-Oriented Programming*, pp. 176–200, Springer, 2007.
- [52] 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*, pp. 391–400, ACM, 2008.
- [53] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pp. 411–416, IEEE, 2012.
- [54] S. Chiba and M. Nishizawa, "An easy-to-use toolkit for efficient java bytecode translators," in *Generative Programming And Component Engineering*, pp. 364–376, Springer, 2003.
- [55] M. M. Lehman and J. F. Ramil, "Towards a theory of software evolution-and its practical impact," in *Principles of Software Evolution, International Symposium on*, pp. 2–2, IEEE Computer Society, 2000.
- [56] M. M. Lehman, "Laws of software evolution revisited," in *Software process technology*, pp. 108–124, Springer, 1996.
- [57] D. L. Parnas, "Software aging," in *Proceedings of the 16th international conference on Software engineering*, pp. 279–287, IEEE Computer Society Press, 1994.

- [58] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.
- [59] W. F. Tichy, "Rcs—a system for version control," *Software: Practice and Experience*, vol. 15, no. 7, pp. 637–654, 1985.
- [60] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa, "On the relevance of code anomalies for identifying architecture degradation symptoms," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pp. 277–286, IEEE, 2012.
- [61] I. Macia, R. Arcoverde, E. Cirilo, A. Garcia, and A. von Staa, "Supporting the identification of architecturally-relevant code anomalies," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pp. 662–665, IEEE, 2012.
- [62] L. Mara, G. Honorato, F. D. Medeiros, A. Garcia, and C. Lucena, "Hist-inspect: a tool for history-sensitive detection of code smells," in *Proceedings of the tenth international conference on Aspect-oriented software development companion*, pp. 65–66, ACM, 2011.
- [63] R. Marinescu, G. Ganea, and I. Verebi, "incode: Continuous quality assessment and improvement," in *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pp. 274–275, IEEE, 2010.
- [64] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, "The structure and value of modularity in software design," *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 5, pp. 99–108, 2001.
- [65] "SWI-PROLOG." <http://www.swi-prolog.org>. [Online; accessed 2012].
- [66] "TOGETHER." <http://www.borland.com/us/products/together/>. [Online; accessed 2012].

Supplemental Material

.1

Appendix A

BNF DETEX Grammar

```
rule_card ::= RULE_CARD : rule_cardName { (rule)+ };
rule      ::= RULE : ruleName { content_rule };

content_rule ::= operator ruleName (ruleName)+ | property | relationship
operator    ::= INTER | UNION | DIFF | INCL | NEG

property    ::= METRIC id_metric value_metric fuzziness
               | LEXIC id_lexic ((lexic_value,)+)
               | STRUCT id_struct
id_metric   ::= DIT | NINTERF | NMNOPARAM | LCOM | LOC_CLAS
               | LOC_METHOD | NAD | NMD | NACC | NPRIVFIELD
               | id_metric + id_metric
               | id_metric - id_metric
value_metric ::= VERY_HIGH | HIGH | MEDIUM | LOW | VERY_LOW
               | NUMBER
id_lexic    ::= CLASS_NAME | INTERFACE_NAME | METHOD_NAME
               | FIELD_NAME | PARAMETER_NAME
id_struct   ::= USE_GLOBAL_VARIABLE | NO_POLYMORPHISM
               | IS_DATACLASS | ABSTRACT_CLASS
               | ACCESSOR_METHOD | STATIC_METHOD
               | FUNCTION_CLASS | FUNCTION_METHOD
               | PROTECTED_METHOD | OVERRIDDEN_METHOD
               | INHERITED_METHOD | INHERITED_VARIABLE

relationship ::= rel_name FROM ruleName card TO ruleName card
rel_name     ::= ASSOC | AGGREG | COMPOS
card        ::= ONE | MANY | ONE_OR_MANY | OPTIONNALLY_ONE
```


.2 Appendix B

BNF TamDera Grammar

The BNF description uses the bold font to display terminal symbols and the first characteres of non-terminal symbols are shown in upper-case format. The symbols '[A]', '(A)+' and '()A'*' respectively impose the cardinalities: optional (0 or 1); at least one; and zero or more to a the symbol *A*.

ConceptDeclaration	::=	concept ConceptId [ConceptInheritance] { [ConceptMapping] [ThresholdVariableList] (AntiDriftRule)* (AssignmentThreshold)* }
ConceptMapping	::=	name: STRING
	::=	parent: STRING
AntiDriftRule	::=	Metric Operator (Value VariableId)
	::=	ConstraintSetId
Metric	::=	LOC CBO LOCM CC DIT ...
Operator	::=	> < = ≤ ≥
AntiErosionRule	::=	only ConceptList can -DependType ConceptList
	::=	ConceptList cannot - DependType ConceptList
	::=	ConceptList must -DependType ConceptList
CptList	::=	ConceptId (, ConceptId)*
DependType	::=	derive invoke depend create declare handle
ConceptInheritance	::=	extends ConceptId
ConstraintSetDecl	::=	constraintset ConstraintSetId { (AntiDriftRule)+ }
CptList	::=	ConceptId (, ConceptId)*
ThresholdVariableList	::=	thresholds: VariableId (, VariableId)*
AssignmentThreshold	::=	assign VariableId to VALUE
ConceptId	::=	STRING
ConstraintSetId	::=	STRING
VariableId	::=	STRING
Value	::=	NUMBER

Acronyms

BAT Bytecode Analysis Toolkit [20](#)

BNF Backus Normal Form [19](#), [20](#)

CD cognitive dimension [24–26](#), [28–44](#), [50](#), [51](#)

CDN Cognitive Dimensions of Notations [15](#), [16](#), [23–26](#), [28–31](#), [45](#), [55](#), [56](#)

DSL domain-specific language [15–21](#), [23–40](#), [42–45](#), [47–50](#), [55](#), [56](#)

GQM Goal, Question, Metric [32](#)

GUI Graphical User Interface [26](#)

SVN Apache Subversion [51](#), [52](#)

URL Uniform Resource Locator [52](#)