Albuquerque, D.; Cafeo, B.; Garcia, A.; Barbosa, S.; Abrahao Gonzales, SM.; Ribeiro, A. (2015). Quantifying usability of domain-specific languages: An empirical study on software maintenance. Journal of Systems and Software. 101:245-259. doi:10.1016/j.jss.2014.11.051.

Quantifying Usability of Domain-Specific Languages: An Empirical Study on Software Maintenance

Diego Albuquerque[1,2] **Bruno B. P. Cafeo**[2] **Alessandro Garcia**[2]

Simone Barbosa[2] **Silvia Abrahão**[3] **António Ribeiro**[1]

[1]*University of Minho*
Campus de Gualtar – 4710-057
Braga – Portugal
Phone/FAX: +351 253604430
e-mail:pg19789@alunos.uminho.pt, anr@di.uminho.pt

[2]*Pontifical Catholic University of Rio de Janeiro – PUC-Rio*
Rua Marquês de São Vicente, 225 – 22453-900
Rio de Janeiro – Brazil
Phone/FAX: +55 21 3527-1500

e-mail:{bcafeo, afgarcia, simone }@inf.puc-rio.br

[3]*Universitat Politècnica de València*
Camino de Vera – 46022
Valencia – Spain
Phone/FAX: +34 (96) 3877000

e-mail: sabrahao@dsic.upv.es

Abstract

A domain-specific language (DSL) aims to support software development by offering abstractions to a particular domain. It is expected that DSLs improve the maintainability of artifacts otherwise produced with general-purpose languages. However, the maintainability of the DSL artifacts and, hence, their adoption in mainstream development, is largely dependent on the usability of the language itself. Unfortunately, it is 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 beforehand by DSL stakeholders. There is even less support on how to quantitatively evaluate the usability of DSLs used in maintenance tasks. In this context, this paper reports a study to compare the usability of textual DSLs under the perspective of software maintenance. A usability measurement framework was developed based on the Cognitive Dimensions of Notations. The framework was evaluated both qualitatively and quantitatively using two DSLs in the context of two evolving object-oriented systems. The results suggested that the proposed metrics were useful: (1) to early identify DSL usability limitations, (2) to reveal specific DSL features favoring maintenance tasks, and (3) to successfully analyze eight critical DSL usability dimensions.
Keywords: DSL; architectural degradation; code anomalies; usability; CDN framework; metrics

1. INTRODUCTION

A domain-specific language (DSL) aims to facilitate construction of software artifacts through specialized abstractions and notations [18]. 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, 25]. Several studies [10, 18, 19, 25, 39] 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 [38]. 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 [18, 38].

Recently, we observed some studies concerned with analyzing the usability of DSLs from several point of views [10, 12, 19]. 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 [38, 40, 41], therefore complementing the qualitative analysis approaches found in the literature [22, 30, 31, 32]. 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, 37]. In particular, we defined a usability metrics suite that was developed based on the Cognitive Dimensions of Notations (CDN) framework [20]. 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

[1] From hereafter, we use the term "DSLs" to refer only to textual DSLs.

code. In addition, both DSLs were designed for different categories of stakeholders, including software architects, programmers and code reviewers.

The remainder of this paper is organized as follows: Section 2 gives some background information needed to better understand the scope of this paper. Section 3 describes the steps required to create the metrics. Section 4 describes the design of a qualitative study aimed at assessing the proposed instantiation of the cognitive dimensions. Section 5 describes the metrics suite developed to analyze the usability of DSLs. Section 6 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 7. Section 8 describes the threats to the validity of our study. Section 9 discusses related work. Finally, Section 10 concludes the work and suggests future developments.

2. BACKGROUND

A Domain-Specific Language (DSL) is a type of programming language or specification language in software development dedicated to a particular problem or solution domain [10, 12, 13]. A DSL facilitates software development through appropriate abstractions and notations. Several studies [10, 11, 12, 15] 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.

In our study, we selected the domain of architectural rules. In this domain, DSLs are used by software architects, programmers and code reviewers to specify and check the adherence of the source code with respect to architectural rules. It is particularly challenging to design a usable DSL in this domain for several reasons [2, 5, 26], 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 architectural 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.

Thus, the next subsection briefly describes the framework used for developing the usability metrics suite. This framework characterizes important usability properties to be assessed in the design of languages, such as DSLs (Section 2.1).

2.1. CDN Framework

The Cognitive Dimensions of Notations (CDN) framework is "a set of discussion tools for use by designers and people evaluating designs" [20]. We chose this framework because we found that it is a widely used technique to support usability evaluation in the literature [6, 21, 24]. This framework provides cognitive dimensions[2] of general use in different domains, as shown in Table I. These CDs are conceptual tools defined to help the designer or evaluator to reason about the system or language being assessed [6, 20]. 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" [20, 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, 21].

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

[2] From hereafter, we use the term "CDs" to refer cognitive dimensions

evaluation of DSLs in evolving systems (Section 3). 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 paper, 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 I. Cognitive Dimensions Originally Defined by CDN [20, 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

2.2. DSL for detecting architectural problems

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, 34, 35, 36]). 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, 36]). As mentioned in Section 2, 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, 37]. Moreover, DSLs for defining architectural rules have been recently proposed [4, 9, 14, 37]. 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 architectural 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. We focus on the expressiveness and conciseness attributes of the DSLs as criteria that define a usable

DSL for detecting architectural anomalies. The selected DSLs for our study are described in Section 6.2.

Nowadays, several studies point out that maintainability is one of the main cost factors in software development projects [27, 28, 29]. 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 several 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. 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 [23].

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 of design attributes. For instance, they rely on coupling metrics to identify the coupling of classes realizing a particular architectural element. Section 3 will further discuss the key characteristics of a DSL intended to support the specification and checking of architectural anomalies in a program.

3. CDN INSTANTIATION FOR EVALUATING DSL USABILITY

As mentioned in Section 2.1, 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 3.1). The metamodel supports the definition and interpretation of the CDs for our study domain. This metamodel was created because it formalizes the domain language [18], 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 (Section 3.1) and the interpretation of the CDs for DSLs (Section 3.2).

3.1. Metamodel of a DSL for detecting architecture degradation symptoms

As aforementioned, we argue that is important to define a metamodel (Figure 1) that represents all DSLs properties found in our domain [9, 18]. 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 II, we describe the metamodel and its interrelationships.

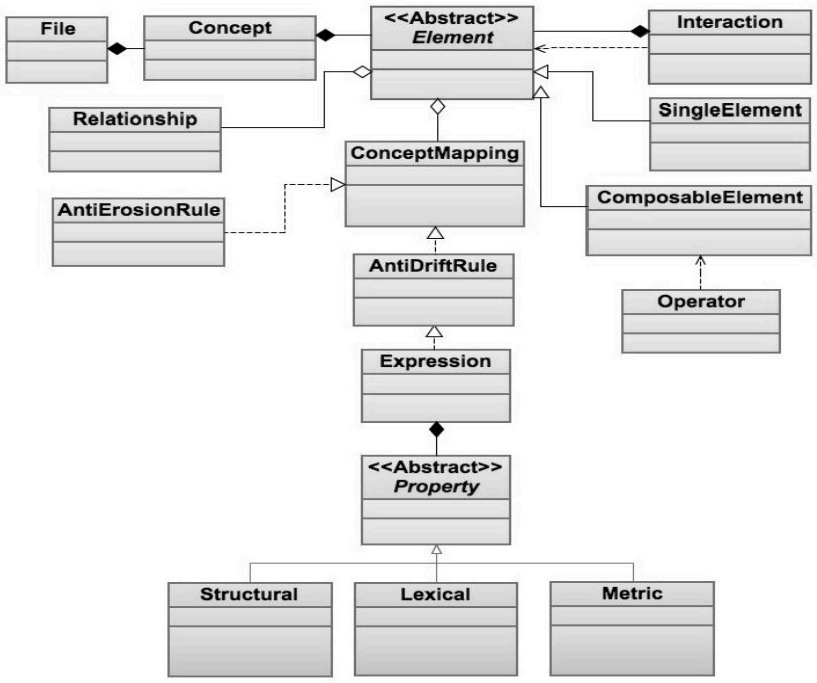

Figure 1. A metamodel of a DSL for detecting architecture degradation symptoms.

Table II. 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

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 2.2). Element is an aggregation of one or more rules that assemble a Concept (Table II). Each entity connected to Element represents a characteristic of a restriction, also known as a rule (Table III), 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 (Table III). 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 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.

Figures 2 and 3 show two different DSLs rules aiming to detect the same architectural anomaly in the same system. In particular, the Concept Controller has drift restriction rules to constrain the number of lines in a method (LOC), and the Lack of Cohesion in Methods (LOCM) of its code elements. Figures 2 and 3 also show how the specification of DSLs can be different even when implementing the same rules to the same system. In this way, it is important to create a high level abstraction to allow us to evaluate different DSLs in a common ground. In other words, to analyze different DSLs, one may create a metamodel to represent the high level elements of DSLs. For example, the entity Concept is defined in Line 1 of Figure 2 and Figure 3. However, this Concept is implemented in different ways in each DSL. Moreover, each Concept contains Elements that are represented in the following lines including the extension ControllerComponents (line 1, 3 and 4 in Figure 2, and lines 2-5 in Figure 3). In addition, the extension ControllerComponents represents the identity Interaction in the metamodel when in the other DSL is not possible to represent. All the described Elements described above are used to create the same anti-drift rule in both DSLs. However, one can notice that TamDera used three Elements, already including the entity Interaction. On the other hand, in the Detex implementation, four elements including the entity Expression are necessary to implement the rule. Section 6.2 will further discuss the characteristics of the DSLs represented in Figures 2 and 3.

```
1 Concept Controller extends ControllerComponents
2 {
3   name:"lancs.mobilemedia.core.ui.controller"
4   LOCM < 30
5 }
```

Figure 2. A DSL rule for a Controller component implemented in TamDera.

```
1 RULE_CARD : Controller {
2 RULE : Layer {INTER LOCGuiLayer CLASSregExpr};
3 RULE : LOCGuiLayer {(METRIC: LOC_CLASS, INF, 270,0) };
4 RULE :  LOCMController {(METRIC: LCOM, 1, INF, 30,0)};
5 RULE : CLASSregExpr {(LEXIC: CLASSE_NAME,
     {lancs.mobilemedia.core.ui.controller})};
6 };
```

Figure 3. A DSL rule for a Controller component implemented in Detex.

3.2. Interpretation of the cognitive dimensions

To be able to adequately interpret all the CDs in Table I, 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 [20] and we tailored them for DSLs of the domain analyzed in our study by using the metamodel presented on Section 3.1. The metamodel helped the instantiation of the CDs by showing important DSLs properties found in our domain. In this way, for instance, it is possible to know which properties/entities from the metamodel can be used to evaluate the DSLs using the instantiated CDs. Moreover, the instantiation of the CDs underwent a refining process through a qualitative evaluation study (Section 4). 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 to different purposes. This characteristic can be quantified by computing the number of Elements changed in the DSL specification from the previous DSL specification in each new version of the target applications. For example, every time some architectural component is added, changed or deleted in an evolution, the DSL specification may need to be changed.

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 including theirs interconnections 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 by using four Elements in a DSL, and it requires three Elements in other DSL, then we say that the DSL where it was used four Elements is more diffuse than the DSL where three Elements where used.

Premature Commitment, the early steps required to create a given DSL specification. In other words, this dimension addresses all the necessary steps before defining a Concept, such as: applications analysis, definition of architectural concepts, etc. 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. This CD represents existing architectural dependencies that cannot be explicitly described in the DSL specification. For example, Concepts that use Elements from other Concepts which in turn extend other Elements. In this case, there are implicit dependencies that are not explicitly represented 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. In our experience in analyzing DSLs, we noticed that same rules can be implemented by different DSLs with a different combination of elements. In this way, the number of elements used in each DSL may vary. For example, if one have to write an architectural rule for a package (coarse granularity), in one DSL, it would be necessary at least three Elements to encompass this package and only one Element in other DSL. On the other hand, if one write an architectural rule for a method (fine granularity), it is necessary just one Element in both languages. Therefore, we argue that more Elements may hamper the understanding (i.e.: the abstraction) of Concepts in the DSL specification.

Closeness of Mapping, how close the DSL specification is to the architectural conceptual domain. A DSL specification may be distributed across different files. In this way, 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 compare two variables: (i) number of Concepts in the file with the DSL specification, and (ii) number of all concepts in the DSL specification. Moreover, it is important to consider that the number of concepts in the file with the DSL specification is the one which includes all 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. In other words, we considered hard as the number of Elements needed to reason about during the DSL specification. For example before defining a Concept "View", in a system with MVC architecture, we need to reason about if the Elements in this Concept do not interfere with other Concepts in our DSL specification. Otherwise can have Elements being implemented in the Concept "View" (for a example an extension) without the developer knowing or wanting.

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.

4. QUALITATIVE EVALUATION

As mentioned in Section 3, 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 7). 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 (Section 4.1), and then we analyze and discuss the results obtained (Section 4.2).

4.1. Qualitative evaluation goal and procedures

Our research goal was to assess with experts of the field our instantiation of the CDN framework (Section 3.2) 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, shown in Table IV. All values in Table IV range from: none, little, moderate and expert. Two of them had developed DSLs focused on supporting software development tasks (B and C); the other two (A and D) 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 gather a wider perspective on the assessment of our propositions related to the CDs.

Table IV – Description of the subjects.

Participant	Software Evolution	Architectural Anomalies	DSL	Experience in Software Development Companies (Years)
A	Moderate	Moderate	Moderate	1-2
B	Moderate	Moderate	Little	3
C	Moderate	Moderate	Little	7
D	Little	Moderate	Moderate	2

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 [42, 43, 44]. All the surveys were recorded;
- Conducting semi-structured interviews with practitioners [45, 46]. 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 V and Table VI 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 V. 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 VI. 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.

4.2. Data Analysis

Table VII 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 VII 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 VII. 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 VII 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 7.

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 VII. 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

5. METRIC DEFINITION

Once we refined the CDs (Section 4), we decided to use the GQM methodology [33] 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 (Section 3.2). Following the described steps above, we identified our goal and its characteristics as shown below:

- **Goal**: Identify the DSL usability limitations from the user's viewpoint.
- **Question Q1**: What is the DSL expressiveness?
  - Q1.1: How many representations can be expressed?
    - Measured Cognitive Dimensions: Hidden Dependencies and Role-Expressiveness.
  - Q1.2: What is the level of abstraction that can be represented?
    - Measured Cognitive Dimensions: Abstraction and 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?
    - Measured Cognitive Dimensions: Viscosity and Diffuseness.
  - Q2.2: How fragmented is the DSL specification?
    - Measured Cognitive Dimensions: Visibility and Hard Mental Operations.

During the GQM methodology, we noticed the need to specify the two questions we wanted to answer (questions Q1 and Q2). 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, 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 [40] (question Q1.2). For the sub questions Q2.1 and Q2.2, 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 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 (Section 3.1) 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, 40], 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 4 and 5). 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 4), and (2) some CDs are not applicable to our investigation, to evaluate expressiveness and conciseness, as aforementioned.

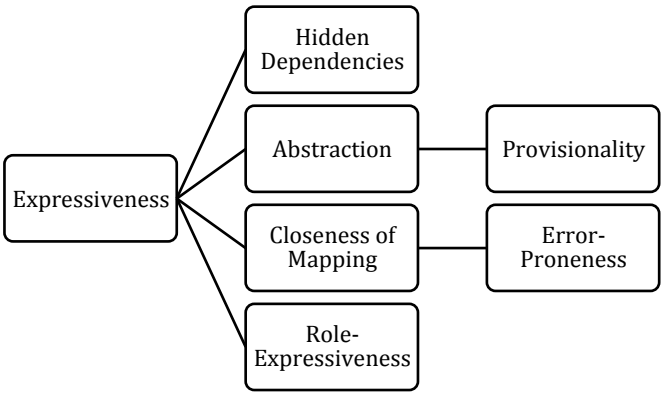

Figure 4. Cognitive dimensions of Expressiveness.

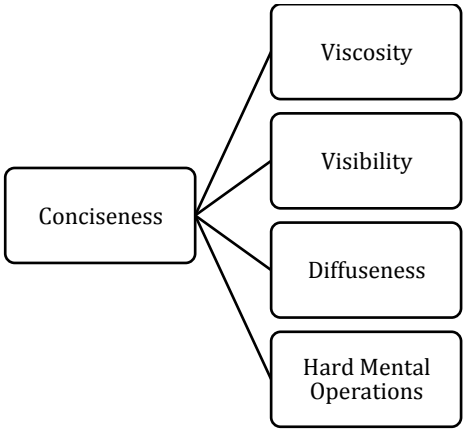

Figure 5. Cognitive dimensions of Conciseness.

Tables VIII and IX present the CDs considered in our study, with their respective descriptions. Table VIII shows the CDs related to DSL Expressiveness, while Table IX 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 (Section 3.2) for usability in DSLs, the characteristics represented in the metamodel (Section 3.1) 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 VIII. 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 IX. 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

6. 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 4 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 (Section 6.1), the selected DSLs to be evaluated (Section 6.2), the selected applications with architecture rules represented using those DSLs (Section 6.3) and their versions (Section 6.4).

6.1. 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 5). 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 (Section 6.2), 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 (Section 6.3). We also checked whether the usability metrics and evaluations results were useful to support an in-depth analysis of expressiveness and conciseness (Section 6.4).

The study was conducted in the context of two DSLs recently designed to support the detection of architecture degradation symptoms (Section 6.2). The comparative assessment of these DSLs was based on their use in the versions of two systems (Sections 6.3 and 6.4). 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 (Section 3.2).

6.2. 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 (Section 2.2). This fact implies that 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 (Section 2.2). 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 [16, 17]. The second DSL is called TamDera [4, 44], which is also fully documented [4], and relies on a robust backend infrastructure, called Vespucci [26]. This DSL was also evaluated to detect architectural anomalies in existing software projects [4, 37].

6.3. 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] (Table X). 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]. For example, Macia et al. [5] has identified that the HealthWatcher (HW) system presented a significant number of architectural violations that increased over time in this system [5]. The authors also revealed that the MobileMedia (MM) system presented a significant number of architectural drift symptoms that emerged along the system evolution. In particular, they were caused by the non-modular realization of new concerns progressively included in the latest system versions.

Table X – Characteristics of the systems used in our study.

	MobileMedia	**HealthWatcher**
Application Type	Software Product Line	Web framework
Programming Language	Java	Java
# of Versions	3	3
Avg. # of Packages	3	8
Avg. # of Classes	30	105
Avg. # of Methods	158	676
Avg. NCSS	1277	5221

6.4. 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 (Section 6.3). 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 HWv1, 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.

7. 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 Section 7.1. The discussion about the usefulness of the metrics and other findings with respect to our research goals (Section 6.1) 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 4) 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.

7.1. Usability measures

The results shown in Tables XI and XII 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 XII) 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 XII) is the average of Elements per Concepts by specifying the Concept of architectural detection rules.

Table XI. Results of the Expressiveness metrics.

Cognitive Dimension	**DSL**	**HW v1**	**HW v4**	**HW v8**	**MM v1**	**MM v4**	**MM v7**
Hidden Dependencies (*Unexpressed dependencies*)	TamDera	0	0	0	0	0	0
	Detex	0	0	0	0	0	0
Role-Expressiveness *(Concept Representations[3]/ 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)

[3] P - Packages, C- Classes, I - Interfaces, M - Methods, F-Fields, Pa -Parameters

A first analysis of Tables XI and XII reveal that both DSLs share similar 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 XI). 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 subsections.

Table XII. 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

7.2. Early indicators of usability strengths and weaknesses

Tables XI and XII also highlight some differences between the two DLSs. The results of the second cognitive dimension analyzed, **Role-Expressiveness** (Table XI), 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 Figure 6 and Figure 7 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 (Figure 6, 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.

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

Figure 6. Rules for Business Façade in TamDera.

In Table XII, 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 rule extensions for a Concept.

```
  1   RULE_CARD: BusinessFacade {
  2   RULE: UnionregExpr {INTER CLASSExper
      FacadeConstraints NoDeeperInheritanceTree
LOCBLayer};
  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    };
```

Figure 7. Rules for Business Façade in Detex.

For example, Figure 6 (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 Figure 7 (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. 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 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 Figure 6 shows that TamDera requires fewer Elements to represent the Concept BusinessFacade. TamDera only needs two Elements (Figure 6, lines 3-4) and one extension, in comparison to five Elements required by Detex (Figure 7, 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 Figure 6 (lines 3-4), where the user just needs to think about two Elements in contrast to five Elements in the counterpart Detex specification (Figure 7, 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 XI, 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 XII. 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 Figure 6 (line 1) that there is an extension of a Concept from another file in TamDera specification, whereas no additional files are required in Figure 7 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.

7.3. Usability in specific project settings

This Section discusses our lessons learned on the use of our evaluation framework (Section 5) to support the expressiveness and conciseness analyses.

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.

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.

8. 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 (Section 6.2). 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 (Section 6.3) and they embrace different types of architecture-level changes.

9. RELATED WORK

In addition to the way we instantiated the CDN framework to evaluate the usability of DSLs for architectural rules checking, there are other instantiations in the community with different purposes. In [6], Maia et al. present the creation, description, and adaptation of the usability CDs to compare two middleware systems with regard to their flexibility. However, Maia describes a qualitative CDN-based method to analyze the cognitive effort made by programmers while adapting middleware implementations, more precisely OiL3 and Mico. They also show how both platforms designed for flexibility have been compared, and report the observations reported from two experts in middleware implementation and the programming languages used in these systems.

Another instantiation [21] presents a study on the Nested Context Language (NCL), the standard declarative language of the Brazilian terrestrial digital TV system. This study shows the usability of its design and conceptual model in supporting reuse at a declarative level. The reuse can happen in static or running code, inside and between applications and reuse of code spans. For that, this study used CDN framework to analyze aspects of the NCL usability also in a qualitative manner.

10. CONCLUSIONS AND FUTURE WORK

In this work, 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 analyses 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.

To the best of our knowledge, this work is a first attempt to define an evaluation methodology for quantitatively analyzing the usability of DSLs. It needs further improvement and validation, although we believe it supports our argument that the use of quantitative analyses can be a valuable approach to understand the limitations of DSLs. So, 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 (e.g.: behavior control and coordination, software architectures, databases, etc) based on our instantiation process, (iii) add a neutral application and neutral elements, like Architecture Description Languages developed by others in order to obtain supplementary results (iv) extend the 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. We believe that the combination of a qualitative analysis with quantitative analysis is possible because the study itself has successfully integrated qualitative

with quantitative evaluation. However, we see two basic ways to integrate quantitative and qualitative analyses in the future: i) conduct two analyses in parallel, independently, and compare their results (this approach can even bring interesting results regarding the refinement of the quantitative analysis that was proposed), and ii) make an initial quantitative analysis, which would give inputs for a deeper qualitative analysis of points shown by quantitative analysis as being critical.

ACKNOWLEDGMENTS. The authors are grateful to Elder Cirilo, Isela Macia, and Eiji Adachi for valuable contributions to this work. We thank Naouel Moha and Yann-Gaël Guéhéneuc for their material and information of the implementation about Detex. We thank Alessandro Gurgel for his material and information of the implementation about TamDera.

This work was funded by: B. Cafeo CAPES PhD Scholarship, and CNPq scholarship grant number 141688/2013-0; A. Garcia Faperj – distinguished scientist grant (number E-26/102.211/2009), CNPq – productivity grants (number 305526/2009-0 and 308490/2012-6), Universal project grants (number 483882/2009-7 and 485348/2011-0), and PUC-Rio (productivity grant).

References

1. Figueiredo E, Garcia A, Lucena C. AJATO: An AspectJ assessment tool. *European Conference on Object-Oriented Programming (ECOOP Demo), France,* 2006.
2. Garcia J, Popescu D, Edwards G, Medvidovic N. Identifying architectural bad smells. *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on,* IEEE, 2009; 255–258.
3. Greenwood P, Bartolomei T, Figueiredo E, Dosea M, Garcia A, Cacho N, Sant'Anna C, Soares S, Borba P, Kulesza U, Rashid A. On the impact of aspectual decompositions on design stability: An empirical study. *ECOOP 2007–Object- Oriented Programming*. Springer, 2007; 176–200.
4. A. Gurgel. Blending and reusing rules for architectural degradation prevention. Master's thesis, PUC-Rio, 2012. Available at: http://www2.dbd.puc-rio.br/pergamum/tesesabertas/1012623_2012_pretextual.pdf.
5. Macia I, Garcia J, Popescu D, Garcia A, Medvidovic N, von Staa A. Are automatically-detected code anomalies relevant to architectural modularity?: an exploratory analysis of evolving systems. *Proceedings of the 11th annual international conference on Aspect-oriented Software Development,* ACM, 2012; 167–178.
6. Maia R, Cerqueira R, de Souza CS, Guisasola-Gorham T. A qualitative human-centric evaluation of flexibility in middleware implementations. *Empirical Software Engineering* 2012; 17(3):166–199.
7. Perry DE, Wolf AL. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes 1992; 17(4):40–52.
8. Silva Filho RS, Bronsard F, Hasling WM. Experiences documenting and preserving software constraints using aspects. *Proceedings of the tenth international conference on Aspect-oriented software development companion,* ACM, 2011; 7–18.
9. Moha N, Guéhéneuc YG, Duchien L, Le Meur A. DECOR: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on* 2010; 36(1):20–36.
10. Van Deursen A, Klint P, Visser J. Domain-specific languages: An annotated bibliography. *Sigplan Notices* 2000; 35(6):26–36.
11. Visser E. WebDSL: A case study in domain-specific language engineering. *Generative and Transformational Techniques in Software Engineering II*. Springer, 2008; 291–373.
12. Humm BG, Engelschall RS. Language-oriented programming via DSL stacking. *ICSOFT (2),* 2010; 279–287.
13. Consel C, Marlet R. Architecture software using: a methodology for language development. *Principles of Declarative Programming*. Springer, 1998; 170–194.
14. Moha N, Guéhéneuc YG. PTIDEJ and DECOR: identification of design patterns and design defects. *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion,* ACM, 2007; 868–869.
15. Gray J, Karsai G. An examination of DSLS for concisely representing model traversals and

transformations. *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on,* IEEE, 2003; 10–pp.
16. Bavota G, De Lucia A, Marcus A, Oliveto R, Palomba F. Supporting extract class refactoring in Eclipse: The ARIES project. *Proceedings of the 2012 International Conference on Software Engineering,* IEEE Press, 2012; 1419–1422.
17. Haderer N, Khomh F, Antoniol G. SQUANER: A framework for monitoring the quality of software systems. Software *Maintenance (ICSM), 2010 IEEE International Conference on,* IEEE, 2010; 1–4.
18. Langlois B, Jitia CE, Jouenne E. DSL classification. *OOPSLA 7th Workshop on Domain Specific Modeling,* 2007.
19. Nishino H. How can a DSL for expert end-users be designed for better usability?: a case study in computer music. *CHI'12 Extended Abstracts on Human Factors in Computing Systems,* ACM, 2012; 2673–2678.
20. Blackwell A, Green T. Notational systems–the cognitive dimensions of notations framework. *HCI Models, Theories, and Frameworks: Toward an Interdisciplinary Science. Morgan Kaufmann* 2003.
21. Neto CdSS, Soares LFG, de Souza CS. The nested context language reuse features. *Journal of the Brazilian Computer Society* 2010; 16(4):229–245.
22. Rainer A, Hall T. A quantitative and qualitative analysis of factors affecting software processes. Journal of Systems *and Software* 2003; 66(1):7–21.
23. Bevan N. Extending quality in use to provide a framework for usability measurement. *Human Centered Design*. Springer, 2009; 13–22.
24. Green TRG, Petre M. Usability analysis of visual programming environments: a cognitive dimensions framework. *Journal of Visual Languages & Computing* 1996; 7(2):131–174.
25. Van Deursen A, Klint P. Little languages: Little maintenance? Journal of software maintenance 1998; 10(2):75–92.
26. Mitschke R, Eichberg M, Mezini M, Garcia A, Macia I. Modular specification and checking of structural dependencies. *Proceedings of the 12th annual international conference on Aspect-oriented software development,* ACM, 2013; 85–96.
27. Pigoski TM. Practical software maintenance: best practices for managing your software investment. John Wiley & Sons, Inc., 1996.
28. Yau SS, Collofello JS. Some stability measures for software maintenance. *Software Engineering, IEEE Transactions on* 1980; (6):545–552.
29. April A, Abran A. *Software maintenance management: evaluation and continuous improvement,* vol. 67. John Wiley & Sons, 2012.
30. Mernik M, Heering J, Sloane AM. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* 2005; 37(4):316–344.
31. Prieto-Díaz R. Domain analysis: an introduction. ACM SIGSOFT Software Engineering Notes 1990; 15(2):47–54.
32. Hudak P. Modular domain specific languages and tools. *Software Reuse, 1998. Proceedings. Fifth International Conference on,* IEEE, 1998; 134–142.
33. Basili VR, Caldiera G, Rombach HD. The goal question metric approach. *Encyclopedia of software engineering* 1994; 528-532.
34. Terra R, Valente MT. A dependency constraint language to manage object-oriented software architectures. Software: *Practice and Experience* 2009; 39(12):1073–1094.
35. Ubayashi N, Nomura J, Tamai T. Archface: a contract place where architectural design and code meet together. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1,* ACM, 2010; 75–84.
36. Morgan C, De Volder K, Wohlstadter E. A static aspect language for checking design rules. *Proceedings of the 6th international conference on Aspect-oriented software development,* ACM, 2007; 63–72.
37. Gurgel A, Macia I, Garcia A, von Staa A, Mezini M, Eichberg M, Mitschke R. Blending and reusing rules for architectural degradation prevention. *Proceedings of the of the 13th international conference on Modularity. ACM*, 2014. p. 61-72
38. Barisˇic A, Amaral V, Goulao M, Barroca B. Quality in use of DSLs: Current evaluation methods. Proceedings of the *3rd INForum-Simpsio de Informtica (INForum2011)* 2011.
39. Mernik M, Heering J, Sloane M. When and how to develop domain-specific languages. ACM

computing surveys (CSUR), 2005, 37.4: 316-344.
40. Sobernig S, Gaubatz P, Strembeck M, Zdun U. Comparing complexity of API designs: An exploratory experiment on DSL-based framework integration. *ACM SIGPLAN Notices,* vol. 47, ACM, 2011; 157–166.
41. Gabriel P, Goulao M, Amaral V. Do software languages engineers evaluate their languages? *arXiv preprint arXiv:1109.6794* 2011.
42. Lamersdorf A, Munch J, Rombach D. A survey on the state of the practice in distributed software development: Criteria for task allocation. *Global Software Engineering, 2009. ICGSE 2009. Fourth IEEE International Conference on, IEEE*, 2009; 41-50.
43. Laitenberger O, Dreyer M. Evaluating the usefulness and the ease of use of a web-based inspection data collection tool. *Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International. IEEE*, 1998; 122-132.
44. Wohlin C, Runeson P, Höst M, Ohlsson C, Regnell B, Wesslén A. Experimentation in software engineering. *Springer*, 2012.
45. Hove E, Anda B. Experiences from conducting semi-structured interviews in empirical software engineering research. *Software Metrics, 2005. 11th IEEE International Symposium. IEEE*, 2005; 10-23.
46. Herbsleb D, Paulish J, Bass M. Global software development at siemens: experience from nine projects. *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on. IEEE*, 2005; 524-533.