Electronic Communications of the EASST
No \volume defined!

No \volumetitle defined!

How to reach a usable DSL? Moving toward a Systematic Evaluation

Ankica Barišić , Vasco Amaral, Miguel Goulão and Bruno Barroca

13 pages

No *ed(s) defined!
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

How to reach a usable DSL? Moving toward a Systematic Evaluation

Ankica Barišić [1], Vasco Amaral[2], Miguel Goulão [3] and Bruno Barroca[4]

[1] barisic.ankica@gmail.com
[2] vasco.amaral@di.fct.unl.pt
[3] miguel.goulao@di.fct.unl.pt
[4] bruno.barroca@di.fct.unl.pt
CITI, Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Campus de Caparica, 2829-516 Caparica, Portugal

Abstract: Domain Specific Languages (DSLs) are claimed to increase productivity, while reducing the required maintenance and programming expertise. In this context, DSL usability by domain experts is a key factor for its successful adoption.

Evidence that support those improvement claims is mostly anecdotal. Our systematic literature review showed that a usability evaluation was often skipped, relaxed, or at least omitted from papers reporting the development of DSLs. The few exceptions mostly take place at the end of the development process where fixing problems identified is too expensive.

We argue that a systematic approach based on User Interface experimental validation techniques should be used to assess the impact of the new DSLs. The rationale is that assessing important and specially tailored usability attributes for DSLs early in language construction will ultimately foster a higher productivity of the DSL users. This paper, besides discussing the quality criteria, proposes a development and evaluation process that can be used to achieve usable DSLs in a better way.

Keywords: Domain Specific Languages Evaluation, Quality in Use, Software Languages Engineering

1 Introduction

Domain Specific Languages (DSLs) and Models (DSMs) are used to raise the level of abstraction, while at the same time narrowing down the design space [GRT04]. This shift of developers' focus to using abstractions that are part of the real domain world, rather than general purpose abstractions closer to the computation domain world, is said to bring important productivity gains when compared to software development using general purpose languages (GPLs) [KT00]. As developers no longer need to make error-prone mappings from domain concepts to computation concepts, they can understand, validate, and modify the produced software, by adapting the domain-specific specifications [DK98]. This approach relies on the existence of appropriate DSLs, which have to be built for each particular domain. Building such languages is usually a key challenge for software language engineers.

Software Languages Engineering (SLE) is becoming a mature and systematic activity, building upon the collective experience of a growing community, and the increasing availability of supporting tools [Kle09]. A typical SLE process starts with the Domain Engineering phase, in order to elicit the domain concepts. The following step is to design the language, capturing the referred concepts and their relationships. Then, the language is implemented, typically using workbench tools, and documented. A development process goes on to the testing, deployment, evolution, recovery, and retirement of languages. However streamlined the process is becoming, it still presents a serious gap in what should be a crucial phase: evaluation, which includes acceptance testing.

If DSLs are meant to close that gap, between the Domain Experts and the Solution computation-platforms, then, from this perspective, they can be regarded as similar to **Human/Computer** (H/C) **Interaction**. *The interaction should favor an increase in efficiency of people performing their duties without this having to cause extra organizational costs, inconveniences, dangers and dissatisfaction for the user, undesirable impacts on the context of use and/or the environment, long periods of learning, assistance and maintenance* [Cat00]. Following this line of thought, most of the requirements concerning evaluation of User Interface (UI) are actually associated with a qualitative software characteristic called *Usability*; which is defined by quality standards in terms of achieving the **Quality in Use** [ISO04].

A good DSL is hard to build because, as noted by Mernik *et al.* [MHS05], it requires both domain knowledge and language development expertise, and few people have both. We should assert claims like that *'the newly designed language brings efficiency to the process'*, or that *'it is usable and effective'*, with an unbiased evaluation process. The closer we get to fill the gap between domain experts and the solution platforms, the closer we are to increase the user's productivity.

This paper is organized as follows. *Section* 2 provides an evaluation approach and background definitions. *Section* 3 presents usability evaluation in general. *Section* 4 gives us an overview on how usability evaluation of programming languages is done. *Section* 5 discusses what DSL evaluation process should look like. Finally, *Section* 6 concludes.

2 Background

We will now include some essential definitions that we will use through the remainder of the paper, namely on the descriptions of existing methodologies for usability evaluation.

2.1 Domain Specific Language Definition

Intuitively, a language is a means for communication between peers. For instance, two persons can communicate with each other by exchanging sentences. These sentences are composed by signs in a particular order. According to the context of a conversation, these sentences can have different interpretations. If the context is not clear, we call these different interpretations *ambiguous*.

In our particular research we are interested essentially in the communication between humans and computers. Hence, we will only consider languages that are used as communication in-

terfaces between humans and machines — i.e. UIs. Therefore human-human languages (e.g. natural languages) and machine-machine languages (e.g. communication protocols) are not relevant for the purposes of the work described in this paper. Examples of user interfaces range from compilers to command-shell and graphical applications. In each of those examples we can deduce the (H/C) language that is being used to perform that communication: in compilers we may have a programming language; in a graphical application we may have an application specific language, and so on. Moreover, we argue that any user interface is a realization of a language. A language is a theoretical object (a.k.a. model) that describes the allowed terms and how to compose them into the sentences involved in a particular human-machine communication.

The **Contexts of Use** i.e. *'the users, tasks, equipment (hardware, software and materials), and the physical and social environments in which a product is used'* [ISO04] is one of the characteristics that we can use to evaluate its usability. In fact, we can use this characteristic to pragmatically distinguish between different products: in our case different languages may have different *Contexts of Use*. Moreover, if they have different *Contexts of Use*, then we can infer that the users of those languages (the humans) most likely will have different *knowledge sets*, each one with a minimum amount of *ontological concepts* [AK03] required in order to actually be able to use each language.

If we say that *Context of Use* has some ontological purpose, then we can see it as *a problem to be solved* in the language user's mind. One example of this is the set of *general purpose languages* (GPL) where each user has to know about *programming* concepts *(variables, cycles, clauses, component, events)*, plus the domain concepts from a given *Context of Use*. Moreover, languages that reduce the use of *computation domain concepts* and focus on the *domain concepts* of the *contexts of use's* problem, are called **domain specific languages**.

If we perform an analysis of the names of reusable components (in reusable infrastructures), and the reusable data structures and methods from existing APIs, and figure out all the possible ways of how they can be composed in a meaningful way, then we can infer an emerging language from those reusable infrastructures. Again, if we restrict the set *contexts of use* to a restricted regular-bounded set (according to the required fidelity of the language: usually precision leads to quality), and we can express meaningful programs only by composing the reusable existing components (or by calling the existing API's methods), then we can infer a **bottom-up** *domain specific language* from that reusable infrastructure. This **bottom-up** method of building languages by reusing existing reusable infrastructures may however generate languages that lack generality in the capability of solving any class of problems of a given domain, or the domain of the problem is not yet fully bounded (categorized) — i.e. there may be irregular composition patterns that can be non-sense w.r.t. the problem.

A **top-down** method would be to complete the domain analysis phase that is behind the existing reusable infrastructure, by discarding any existing implementation and focusing only on the complete description and categorization of the class of problems from which its users will use our new *domain specific language* to describe their solutions while using the identified *problem concepts* — w.r.t. its *contexts of use*. If we find a mapping between all the possible expressible solutions which might be very difficult in some cases in our new *domain specific language* and the existing *concepts* of a reusable infrastructure, then we have assembled a **top-down** *domain specific language*. DSLs that are built in a top down fashion are mostly called horizontal DSLs ([Kle09]).

In practice, it is more common for a DSL design for H/C communication to be built using a combination of bottom-up and top-down approaches.

2.2 Usability definition

As previously mentioned, *Usability* is a key characteristic for evaluating the Quality of UIs, and, since we defined H/C languages as UIs, in our perspective, we should also use it for evaluating the Quality of this kind of languages. The difference between usability and the other software qualities is that to achieve it, one has to concentrate not on system features but specifically on user-system interaction characteristics. There are several interpretations of usability e.g. [Bev09], [PB09], [Bev95]. However, if we look at the standards, such as ISO 9241-11 **Usability** is defined as: *'the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use'* [Bev09]. Moreover ISO 9126 estimated this definition with the notion of *'Goal Quality'*, that has to be evaluated through the already mentioned **Quality in Use** that is perceived by the user during actual utilization of a product in real **Context of Use** [ISO04].

Similarly to the other software qualities, usability evaluation cannot be simply added at the end of the development process. Instead, it has to be included in the development process from the beginning by taking into consideration internal and external quality attributes.

Internal quality is the *'totality of characteristics of the software product from an internal view'* that provides usability metrics that are used for predicting the extent to which the software in question can be understood, learned, operated, attractive and compliant with usability regulations and guidelines. Internal metrics can be applied to a non-executable software product during designing and coding. Internal metrics provide users, evaluators, testers, and developers with the benefit that they are able to evaluate software product quality and address quality issues early before the software product becomes executable [ISO04].

External quality is the *'totality of characteristics of the software product from an external view'* that provide us with metrics that use measures of a software product derived from measures of the behavior of the system of which it is a part, by testing, operating and observing the executable software or system. Before acquiring or using a software product it should be evaluated using metrics based on business objectives related to the use, exploitation and management of the product in a real Context of Use. External metrics provide users, evaluators, testers, and developers with the benefit that they are able to evaluate software product quality during testing or operation [ISO04].

Evaluating **Quality in Use** validates software quality in specific user-task scenarios. *Quality in Use* is the user's view of the quality of a system containing software, and is measured in terms of the result of using the software, rather than properties of the software itself. Achieving *Quality in Use* is dependent on achieving the necessary *External quality*, which in turn is dependent on achieving the necessary *Internal quality*. Measures are normally required at all three levels, as meeting criteria for internal measures is not usually sufficient to ensure achievement of criteria for external measures, and meeting criteria for external measures is not usually sufficient to ensure achieving criteria for *Quality in Use*. The most complete Quality model for achieving *Usability* was given by ISO IEC CD 25010.3 [PB09] in the terms of achieving *Quality in Use* as presented in *Fig.* 1.

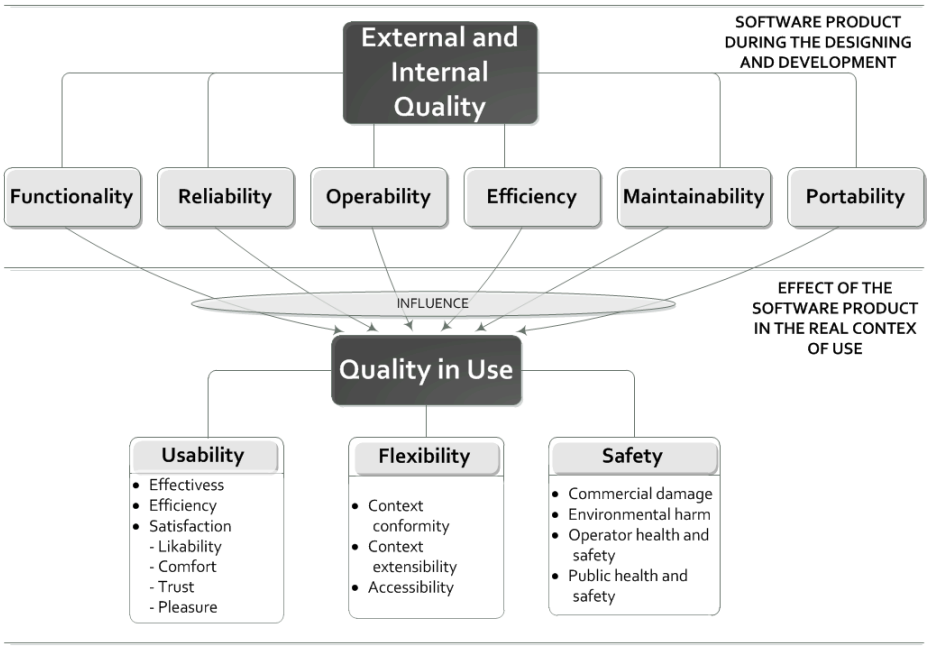

Figure 1: Quality model for achieving Quality in Use based on ISO IEC CD 25010.3

Achieving *Quality in Use* for different users means achieving different goals; *The end user* for whom quality in use is mainly a result of Functionality, Reliability, Operability and Efficiency; *The person maintaining* the software for whom Quality in Use is a result of Maintainability; *The person porting* the software for whom Quality in Use is a result of Portability. The new model for achieving *Quality in Use* provides a framework for a more comprehensive approach to specifying usability requirements and measuring usability with taking in account the stakeholder perspective.

To evaluate achieved Quality in Use of DSLs in real context of use we find it most relevant to evaluate its attributes of Usability and Flexibility (*Fig.* 1):

- *Effectiveness* that should determine the accuracy and completion of the implementation of the sentences,
- *Efficiency* which tell us what level of effectiveness is achieved at the expense of various resources, such as mental and physical effort, time or financial cost, commonly measured in the sense of time spent to complete a sentence,
- *Satisfaction* that captures freedom from inconveniences and positive attitude towards the use of the language and
- *Accessibility* with accent on learnability and memorability of the language terms.

In that order, we need to find suitable quantitative and qualitative measures that will be reliable to capture achieved goals. These attributes should contribute, in most cases, to claimed productivity improvements of DSLs and can be evaluated as in [BAGB11a].

3 Usability Evaluation

3.1 General approach to Usability evaluation

Nielsen and Molich proposed evaluating usability in four ways [NM90];

Formally by some analysis techniques. *Evaluations using models and simulations* can predict measures such as time to complete a task or the difficulty of learning to use a product. Some models have the potential advantage that they can be used without the need for any prototype to be developed.

Automatically by a computerized procedure. This can be done by *Automated checking of conformance to guidelines and standards* or by *Evaluation of data collected during system usage*. This kind of evaluation is possible when initial prototypes or initial versions of full implementation are available.

Empirically by experiments with test users. *Evaluation with users* is recommended at all stages of development if possible, or at least in final stage of development. We can use: *Formative* methods that focus on understanding the user's behavior, intentions and expectations in order to understand any problems encountered, and typically employ a 'think-aloud' protocol or *Summative* methods that measure the product usability, and can be used to establish and test user requirements. Testing may be based on the principles of standards and measure a range of usability components. Each type of measure is usually regarded as a separate factor with a relative importance that depends on the Context of Use. Iterative testing with small numbers of participants is preferable, starting early in design and development process.

Heuristically by simply looking at the product and passing judgment according to an own opinion. It is usually considered as *Evaluation conducted by expert* and it can be used when initial prototypes are available. Expert methods that do not use task scenarios are referred to as reviews or inspections, while task-based evaluations are referred to as walkthroughs. Conducting expert evaluation is recommended to identify as many usability issues as possible in order to eliminate them before conducting user-based evaluations.

Empiric and Heuristic approaches have been used to assess the usability of DSLs [MPGB00], [KMB+96], [HPV09], [BAGB11a], [KMC11]. The lessons learned from those works can be useful for establishing guidelines and devising tool support for DSL evaluation. These should then be combined with formal and automatic approaches, in a proposal for a cost-effective approach to usability assessment.

3.2 Software Engineering with Usability concerns

By allowing significant changes to correct deficiencies along the development process instead of just evaluating at the end of it (when it might be too late), *User Centered Design* can reduce development and support costs, increase sales, and reduce staff cost for employers. The essential activities required to implement *User Centered Design* are described in ISO 13407 [Bev05]:

- Plan and manage the human centered design process
- Understand and specify the context of use
- Specify the stakeholder and organizational requirements

- Produce design solutions
- Evaluate designs against requirements

Usability has two complementary roles in design: as an *attribute that must be designed into the product*, and as the *highest level quality objective* which should be the overall objective of design. In the first phase it is important to study existing style guidelines, or standards for a particular type of system; interviewing current or potential users about their current system or tools they are using to help them in accomplishing their tasks, its strengths and weaknesses, and their expectations for a new or re-designed system; conducting Context of Use study of a particular situation. All these contribute to an initial understanding what the system should do for the users and how it should be designed. Initial design ideas can then be explored, considering alternative designs and how they meet users' needs. After developing potential designs it is time to build the prototypes that should be obviously simple and unfinished, as that allows people involved in evaluations to realize that it is acceptable to criticize them. In contrast, a prototype very close to the final product is likely to inhibit evaluators from openly criticizing it, which might lead to a loss of valuable feedback from those evaluators. It is important to explore particular design problems before considerable effort is put into full implementation and integration of components of a system. A number of iterations of evaluation, designing and prototyping may be required before acceptable levels of Usability are reached. Once the design of various components of system has reached acceptable levels, integration of components and final implementation of the interactive system may be required. Finally, once the system is released to users, an evaluation of its use in real contexts may be highly beneficial [PB09]. This kind of iterative evaluation approach should be merged with DSL development cycle, so we can avoid unnecessary coasts of developing inadequate DSLs for its end users.

4 Evaluation of Programming Languages

Usability of GPLs is mostly indirectly measured by the size of the community that uses GPL. The rationale is that, if so many people are using this GPL, that says something about its usability. But there are also other sorts of evaluations on GPLs, namely benchmarks, feature-based comparisons and heuristic-based evaluations [Pre00]. Comparisons are done on different versions of the same language or on the different languages focusing on some characteristic that indicate suitability of language to its intended Context of Use. There are also Heuristic-based evaluations that provide guidelines for evaluating syntax of visual languages based on the studies of cognitive effectiveness [Moo09].

When usability problems are only identified too late a common approach to mitigate them is to build tool support that minimizes their effect on users' productivity [BJRT10], [PFHS09].

There is an increasing awareness to the usability of languages, fostered by the competition of language providers. Better usability is a competitive advantage, although evaluating it remains challenging, because it is hard to interpret existing metrics in a fair, unbiased way, which resists to external validity threats concerning the broad user groups, or internal ones - it is very easy to end up comparing apples with oranges, when evaluating competing languages.

In general, the software industry does not invest much on the evaluation of the usability of DSLs [GGA10]. It is unclear whether this results from an insufficient understanding of the SLE process which, in our opinion, must include the evaluation of the DSLs. Language engineers may perceive the investment in evaluation as an unnecessary cost and prefer to risk providing a solution which has not been validated, w.r.t. its usability, by end users. With anecdotal reports of 3-10 times productivity improvements, [KT00],[WL99],[Met07b], or "clearly boosted development speeds" [Met07a] in industrial settings, why bother with validation?

The problem, of course, is that anecdotal reports on improvements lack external validity. Other reports, such as [BJMH02], present maintainability and extensibility improvements brought by a combination of DSLs and Software Product Lines (SPL), but it is unclear which share of the merit belongs to DSLs and which should be credited to SPLs. The usage of DSLs has been favorably compared to the usage of templates in code generation, with respect to flexibility, reliability and usability [KMB+96]. Other work evaluates a visual language against a GPL for which it is a front-end [MPGB00]. Another success story can be found in [HPD09], where a survey conducted with users of a particular DSL clearly reports on noticeable improvements in terms of reliability, development costs, and time-to-market. The usability of that particular DSL and its toolset are among the most important success factors of DSL introduction in that context. But are these improvements typical, or exceptional? The honest answer can only be one: we do not know.

We conducted a systematic literature review to assess the extent to which DSLs are evaluated and how they are evaluated [GGA10]. Our review analyzed 15 of the most important scientific venues covering Software Languages Engineering, from 2001 to 2008. Out of over 640 papers published in those venues, we finally selected 36 which reported either on the process developing at least one DSL (33) or on experimental validation of DSLs (3). Only 3 of the inspected papers reported on using quantitative data in their DSL evaluation, while 2 papers reported the usage of qualitative data in DSL evaluation. 21 papers did not provide any information concerning how evaluation was performed. Finally, 10 papers do not report any kind of evaluation of the produced DSLs. Only 2 of the papers claim to have used industrial level validation, against 21 papers which report on toy examples. No details are available concerning the remaining 3 papers which may have performed some sort of evaluation. Only 3 papers report having used industrial, or specialized subjects in their DSL evaluation, while this information is unknown in the remaining cases where some sort of evaluation was conducted. Only 1 paper reports the usage of specific usability techniques, borrowed from the evaluation of GPLs. 6 other papers report on ad-hoc usability evaluation.

Overall, the level of DSL evaluation found in our survey can be considered low, and the details on the few performed evaluations are clearly insufficient. For instance it is often the case where information concerning who participated in the evaluation is missing. We do not know whether the final users participated in the evaluation at all, in most situations. We were not able to find compelling evidence supporting the improvement claims on DSL usage. Although this does not necessarily mean no usability evaluation is being performed, it sends the wrong message to practitioners who should be concerned with systematically evaluating the DSLs they produce. It should be noted that this kind of evaluation, comparing the impact of different languages in the software development process has some tradition, in the context of GPLs (e.g. [Pre00]) and their impact on developer productivity. Why should this be different with DSLs?

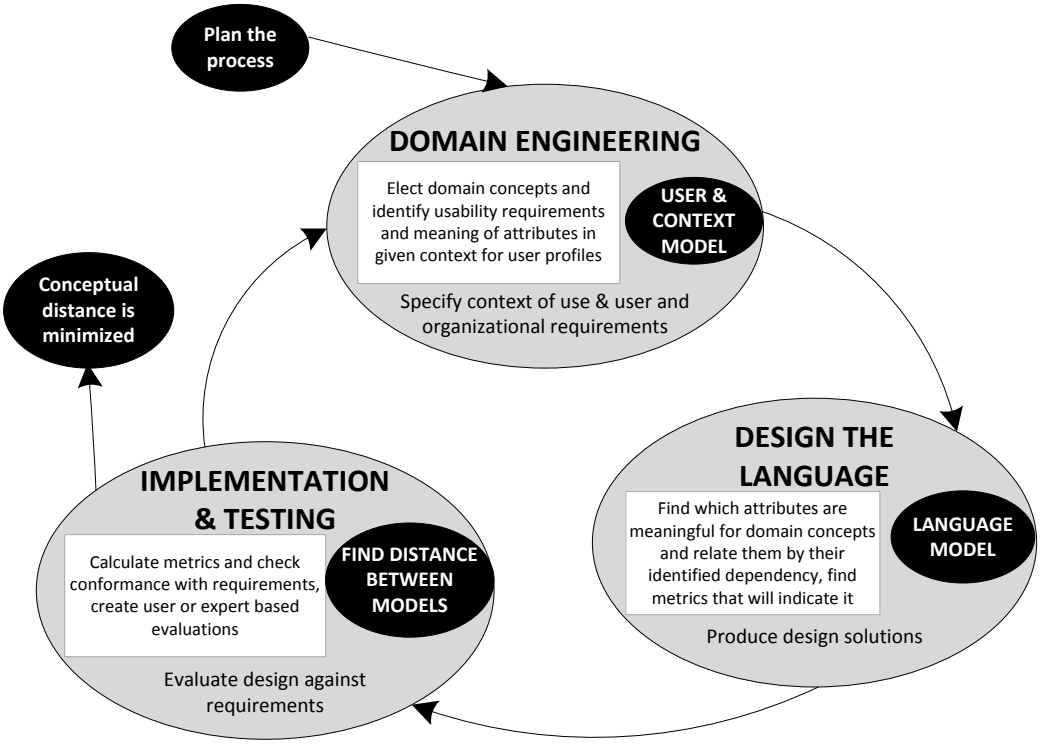

Figure 2: Evaluation Process for DSLs' Usability.

5 Discussion on DSL Usability Evaluation proposal

In order to propose a process for performing Usability evaluation on DSLs, we first must ask what are the main goals of a software language engineer when devise a new DSL. The main design objectives for building a new DSL are:

- To build a comprehensive language that captures domain expressivity.
- To achieve compliance with existing standards in a given domain.
- To overcome previously identified problems in the domain.

As in other usability evaluation methodologies, Usability evaluation should be embedded in the DSL itself, and considered from the beginning of its development together with User Centered Design activities from *Section* 3.2 as we propose in *Fig.* 2. According to our proposal, Usability requirements should be identified during *domain engineering phase* of DSL's construction i.e. while eliciting domain concepts. It should be mandatory to understand and specify the Context of Use of DSLs and which kind of user groups it should target by constructing *User* and *Context* model. In order to achieve that, the language engineer should engage interviews or questionnaires with the DSL's intended end users in order to capture information about their

working environment and the products that are already used within the domain. It is necessary to identify characteristics that the users find useful, frustrating or lacking while using existing approach to solve the problem, and group them in the usability requirements. Connecting usability requirements to conceptual dimensions we can identify what quality means in specific context of use for end user.

In the *language design phase*, the language engineer should elect according to requirements which quality attributes from *Fig.* 1 are connected to the domain concepts. For each domain concept, he should identify or predict both its frequency and relevance within the domain, that should be obtained from the precisely defined questionnaire. He can do it by assigning weights between quality attributes and domain concepts according to their influence on final Usability of the language [BAGB11b]. Aiming to produce pertinent usability metrics and tests from collected information's it is important to know which attributes will contribute to achieve Quality in Use .

During the *implementation phase*, the language engineer can benefit from the collected information by means of tools that measure Usability (w.r.t. the stated Usability requirements) directly on the DSL prototype. Finally, in the *testing phase*, the language engineer should conduct (at least) a Heuristic-based Usability evaluation to validate the list of identified Usability requirements. If it seems that the requirements are achieved, to certify that conceptual distance between domain expert in its context of work and the domain that language is able to provide, we should conduct a user-based evaluation in a real context of use. That is done by giving the users real problems to solve in order to cover the most important tasks identified in the domain. Data about satisfaction and cognitive workload should also be evaluated subjectively through questionnaires. It is especially important in this phase to measure all the learnability issues, since DSLs should be (in principle) easy to learn and remember. Of course, in order to certify that we are creating a good DSL we should conduct a comparative analysis with previous products that are already used in the domain and also were built to achieve the same goals. Examples of the user based evaluations of DSLs, that presents examples of tasks and questions that are constructed to measure achieved Quality in Use can be seen in [BAGB11a] and [KMC11].

Main idea is that we can measure the distance between the language model and the user-context model during language development through defined Internal and External quality metrics that influence Usability. The smaller the conceptual distance, the level of achieved Quality in Use will be higher.

6 Conclusions and Future Work

In this paper, we showed that the software industry, in general, does not seem to invest much on the evaluation of DSLs. However, since DSLs are built for a specific domain of use in order to close the gap between domain experts and software engineers, we find that it is essential to evaluate its usability.

It was also shown that Usability is the main quality attribute while performing UI evaluation. If we consider DSLs as a subset of UIs, then we find that evaluating DSL's Usability can bring a positive influence on their users' productivity. Moreover, unlike other software products, DSL's Usability evaluation can be an accurate activity, since precisely defined DSLs can target specific Contexts of Use, inside a particular set of user profiles.

The main contribution of this paper is the proposal of an evaluation process for DSLs' Usability that is to be applied during DSLs' life-cycle. With this evaluation process we are able to evaluate Usability of a DSL in early stages of its development in order to predict its outcome w.r.t. Usability and prevent user-interaction mistakes, hence by achieving a usable DSL by construction.

As future work, we will instantiate our proposed DSL evaluation process in the construction of new DSLs which will take into account the Usability aspect from the very beginning of their development. From this instantiation, we expect to devise languages and tools that can effectively and automatically measure the identified Usability factors early and during DSLs' development.

Bibliography

[AK03] C. Atkinson, T. Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Softw.* 20:36–41, September 2003.
doi:10.1109/MS.2003.1231149
http://portal.acm.org/citation.cfm?id=942589.942704

[BAGB11a] A. Barišić, V. Amaral, M. Goulão, B. Barroca. Quality in Use of Domain Specific Language: a Case Study. *Proceedings of the Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2011), held at Splash 2011*, Octobar 2011.
http://ecs.victoria.ac.nz/twiki/pub/Events/PLATEAU/Program/plateau2011-barisic.pdf

[BAGB11b] *Quality in Use of DSLs: Current Evaluation Methods*. University of Coimbra, Coimbra, Portugal, September 2011.

[Bev95] N. Bevan. Measuring usability as quality of use. *Software Quality Journal* 4(2):115–130, 1995.

[Bev05] N. Bevan. Cost benefits framework and case studies. *Cost-Justifying Usability: An Update for the Internet Age. Morgan Kaufmann*, 2005.

[Bev09] N. Bevan. Extending quality in use to provide a framework for usability measurement. *Human Centered Design*, pp. 13–22, 2009.

[BJMH02] D. Batory, C. Johnson, B. MacDonald, D. v. Heeder. Achieving extensibility through product-lines and domain-specific languages: a case study. *ACM Transactions on Software Engineering and Methodology* 11(2):191–214, 2002.

[BJRT10] *Using CogTool to model programming tasks*. 2010.

[Cat00] T. Catarci. What happened when database researchers met usability. *Information Systems* 25(3):177–212, 2000.

[DK98] A. V. Deursen, P. Klint. Little Languages: Little Maintenance? *Journal of Software Maintenance: Research and Practice* 10(2):75–92, 1998.

[GGA10] *Do Software Languages Engineers Evaluate their Languages?* 2010.

[GRT04] J. Gray, M. Rossi, J.-P. Tolvanen. Preface. *Journal of Visual Languages and Computing, Elsevier* 15:207–209, 2004.

[HPD09] F. Hermans, M. Pinzger, A. V. Deursen. Domain-Specific Languages in Practice: A User Study on the Success Factors. In *12th International Conference on Model Driven Engineering Languages and Systems*. Volume 5795/2009, pp. 423–437. Lecture Notes in Computer Science, Denver, Colorado, USA, 2009.

[HPV09] F. Hermans, M. Pinzger, A. Van Deursen. Domain-specific languages in practice: A user study on the success factors. *Model Driven Engineering Languages and Systems*, pp. 423–437, 2009.

[Kle09] A. Kleppe. *Software language engineering: creating domain-specific languages using metamodels*. Addison-Wesley, 2009.

[KMB+96] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, L. Walton. A Software Engineering Experiment in Software Component Generation. In *International Conference on Software Engineering (ICSE'1996)*. Pp. 542–552. IEEE Computer Society, Berlin, Germany, 1996.

[KMC11] T. Kosar, M. Mernik, J. Carver. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical Software Engineering*, pp. 1–29, 2011.

[KT00] S. Kelly, J.-P. Tolvanen. Visual domain-specific modelling: benefits and experiences of using metaCASE tools. In Bézivin and Ernst (eds.), *International Workshop on Model Engineering, at ECOOP'2000*. 2000.

[Met07a] MetaCase. EADS Case Study, http://www.metacase.com/papers/MetaEdit_in_EADS.pdf. Technical report, MetaCase, 2007.

[Met07b] MetaCase. Nokia Case Study, http://www.metacase.com/papers/MetaEdit_in_Nokia.pdf. Technical report, MetaCase, 2007.

[MHS05] M. Mernik, J. Heering, A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys* 37(4):316–344, 2005.

[Moo09] D. Moody. The physics of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, pp. 756–779, 2009.

[MPGB00] N. Murray, N. Paton, C. Goble, J. Bryce. Kaleidoquery–a flow-based visual language and its evaluation. *Journal of Visual Languages & Computing* 11(2):151–189, 2000.

[NM90] *Heuristic evaluation of user interfaces*. 1990.

[ISO04] International Standard Organization. ISO/IEC 9126 Quality Standards. 2004. http://www.iso.org/iso/

[PB09] H. Petrie, N. Bevan. The evaluation of accessibility, usability and user experience. *The Universal Access Handbook*, 2009.

[PFHS09] K. Phang, J. Foster, M. Hicks, V. Sazawal. Triaging Checklists: a Substitute for a PhD in Static Analysis. *Evaluation and Usability of Programming Languages and Tools (PLATEAU) PLATEAU 2009*, 2009.

[Pre00] L. Prechelt. An Empirical Comparison of Seven Programming Languages. *IEEE Computer* 33(10):23–29, 2000.

[WL99] D. M. Weiss, C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison Wesley Longman, Inc., 1999.