Quality in Use of Domain-Specific Languages: a Case Study

Ankica Barišić Vasco Amaral Miguel Goulão Bruno Barroca

CITI, Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Campus de Caparica, 2829-516 Caparica, Portugal
a.barisic@campus.fct.unl.pt, vasco.amaral@di.fct.unl.pt, miguel.goulao@di.fct.unl.pt, bruno.barroca@di.fct.unl.pt

Abstract

Domain-Specific Languages (DSLs) are claimed to increment productivity, while reducing the required maintenance and programming expertise. In this context, DSLs usability is a key factor for its successful adoption.

In this paper, we propose a systematic approach based on User Interfaces Experimental validation techniques to assess the impact of the introduction of DSLs on the productivity of domain experts. To illustrate this evaluation approach we present a case study of a DSL for High Energy Physics (HEP).

The DSL on this case study, called Pheasant (PHysicist's EAsy Analysis Tool), is assessed in contrast with a pre-existing baseline, using General Purpose Languages (GPLs) such as C++. The comparison combines quantitative and qualitative data, collected with users from a real-world setting. Our assessment includes Physicists with programming experience with two profiles; ones with no experience with the previous framework used in the project and other experienced.

This work's contribution highlights the problem of the absence of systematic approaches for experimental validation of DSLs. It also illustrates how an experimental approach can be used in the context of a DSL evaluation during the Software Languages Engineering activity, with respect to its impact on effectiveness and efficiency.

Categories and Subject Descriptors D.3 PROGRAMMING LANGUAGES [*D.3.2 Language Classifications*]: Specialized application languages; Very high-level languages; D.2 SOFTWARE ENGINEERING [*D.2.8 Metrics*]: Process metrics

General Terms Experimentation, Human Factors, Languages, Measurement

Keywords Experimental Software Engineering, Domain-Specific Languages, Usability, Language Evaluation, Software Language Engineering

1. Introduction

Domain-Specific Languages (DSLs) are meant to close the gap between the Domain Experts and Solution computation platforms. The closer we get to fill this gap, the closer we are to increase the user's productivity. The shift of the developers' focus to use 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 using General Purpose Languages (GPLs) [11].

Software Languages Engineering (SLE) is becoming a mature and systematic activity, built upon the collective experience of a growing community, and the increasing availability of supporting tools [13]. 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, by capturing the referred concepts and their relationships. Then, the language is implemented, typically by using workbench tools, followed by documentation. A development process goes on to the testing, deployment, evolution, recovery, and retirement of these languages. However streamlined the process is becoming, it still presents a serious gap in what should be a crucial phase: Evaluation.

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 favour an increase in the 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 [5].

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** [8].

2. Background

The methods used to evaluate usability of GPLs are not always adequate for DSLs because they are not systematic and are centred only on computation domain concepts. The GPLs intended users are expected to have high knowledge of technical and computational concepts, while the DSLs intended user group are domain experts that are more familiar with the domain concepts. Therefore, we need a different approach to perform evaluation of DSLs.

We conducted a systematic literature review to assess the extent to which DSLs are evaluated and how they are evaluated [6]. The level of DSL evaluation found in our survey can be considered to be low, and the details on the few performed evaluations are clearly insufficient. We observed that there was a predominance of toy DSLs with unsubstantiated claims to their merits. Most authors present reports of usability evaluations that are impossible to replicate and to extract a precise rationale from, e.g., it is hard to reason about the representativeness of their DSL's users due to the poor characterization of subjects involved in the evaluation.

PLATEAU'11, October 24, 2011, Portland, Oregon, USA.

With a few remarkable exceptions [14], [12], there is little evidence supporting the improvement claims on DSLs usage. This does not mean that no usability evaluation is being performed, but it sends the wrong message to practitioners who should be concerned with the usability evaluation of the DSLs they produce. 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. [18]), and their impact on the software developer productivity. Why should this be different with DSLs? Apparently, "some tradition" is not enough. As noted by Markstrum [15], it is also often the case where, even for GPLs, many claims on language properties (including their usability) are mostly unproven. While in this paper we are mostly concerned with DSLs and their evaluation, we regard this issue as a challenge to GPL developers, as well.

Among other possible explanations, this state of practice may stem from a lack of enough software experts that completely understand the SLE process, or from a lack of experimental evidence that clearly backs up the qualitative improvement claims that we often find in the literature. Without such evidence, it may be the case that decision makers consider proper language evaluation as a waste of time and resources. If so, they may prefer to risk using or selling inadequate DSLs rather than evaluating them properly.

The incremental nature of a typical DSL life cycle may also give the erroneous feeling that the language is being implicitly validated due to the intense interaction with the domain experts. However, the domain experts involved in the language development are not necessarily the end users, and may therefore introduce biases in the perception of the language design and its usability.

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. A good DSL is hard to build because, as noted by Mernik *et al.* [16], it requires both domain knowledge and language development expertise, and few people have both. In that case we should ask what is a cost of producing inadequate DSL for their intended users.

3. Domain-Specific Languages as User Interfaces

Since we are focusing on the evaluation of usability aspects of DSLs, we need to provide a suitable definition of DSL so that we are able to evaluate them w.r.t. those usability aspects.

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 interfaces between humans and computers, i.e. User Interfaces (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 UIs 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 UI 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-computer communication. Languages can be deduced in two directions, human-computer and computer-human, since the feedback from the computer has to be given in such a way that it can be correctly interpreted by the humans.

Semiotics, the study of the structure and meaning of languages, is a part of linguistics that studies the dependencies and influences among *Pragmatics*, *Syntax*, and *Semantics*. The **Syntax** of a language defines what signs we can use in that language, and how we can compose those signs to form sentences. The **Semantics** of a language defines the conceptual meaning of the sentences in that language by stating how they can be logically interpreted. Finally, the **Pragmatics** of a language defines the *context of use* from which the sentences of that language can have some logical meaning.

The **Context of Use** i.e. *'the users, tasks, equipment (hardware, software and materials), and the physical and social environments in which a product is used'* [8] should be used to evaluate DSLs usability, to distinguish between different products: in our case, different languages may have different *Contexts of Use*, so 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* [2] 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 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**. Notice that, in these pragmatic perspective languages that do not even share the same base syntax may actually share the same *domain concepts*, i.e. the intersection of their *domain concepts* is not empty for a given non-empty intersection of *contexts of use*. If the intersection of their *contexts of use* is empty then they actually do not share any of the identified *domain concepts*.

For example: consider both the SQL and C languages. The SQL language has a reserved word called $table$ to represent a database table from a DBMS. There is no $table$ in the list of reserved words of C language that the user of C can immediately read as $table$ with the same meaning as read in SQL (i.e. a database table from a DBMS). However, one of the *contexts of use* of SQL where $table$ is applied: $createtable$ can be emulated by means of a high level C (Application Programmers Interface) API function that have the same purpose of creating a table in the same DBMS. Moreover, if there is no C API supported by the DBMS, then we can even imagine how it would be to write it completely in C as part of the implementation of *the context of use* stated in $createtable$.

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 a **bottom-up** DSL 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 if the domain of the problem is not yet fully bounded (categorized), there may be irregular composition patterns that can be nonsense 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 DSL to describe their solutions while using the identified *problem concepts* w.r.t. its *context of use*. If we find a mapping between all the possible expressible solutions which might be very difficult in some cases in our new DSL and the existing *concepts* of a reusable infrastructure, then we have assembled a **top-down** DSL.

DSLs that are built in a top down fashion are mostly called *horizontal* DSLs, while DSLs that are built in bottom-up fashion are called *vertical* DSLs ([13]). 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.

4. Usability Evaluation

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 only on system features but especially on user-system interaction characteristics. ISO 9241-11 [9] defines **Usability** 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*'.

ISO 9126 [8] extends this definition with the notion of *'Goal Quality'*, which has to be evaluated through the already mentioned **Quality in Use** that is perceived by the user during actual utilization of a product in its real **Context of Use**. The definition of *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.

Not all usability aspects can be given equal weight in a given language, so it is not always possible to achieve optimal scores for all usability attributes [3]. To evaluate the achieved Quality in Use of DSLs we find it most relevant to evaluate

- *Effectiveness* that determine the accuracy with which a developer completes language sentences
- *Efficiency* which tells 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 focus on learnability and memorability of the language terms.

We need to define suitable quantitative measurements and qualitative indicators, to support a reliable assessment of the achieved quality in use. When apparently conflicting usability requirements are identified, a first approach is to look for a win-win solution that can reconcile both requirements. If this is not feasible, we need to define which usability characteristics are priority in the specific context of the project under scrutiny and favour those. These priorities can be defined based on users and tasks analysis.

To know the users we should identify the characteristics of target user population. For several kinds of end users we should analyse all kinds of them using techniques like questionnaires, interviews and observation to capture [20]: *Who* are the users?; *What* do users do?; *Why* do they do it?; *How* do they do it?; *When* do they do it?; *What* tools do they use?; Understanding 'how' and 'why' should give us deeper knowledge about the tasks. Performing task analysis by studying of the way the people perform tasks with existing systems or through high level abstraction study of cognitive processes we should identify the individual tasks the language should perform. From this we can build the desired cognitive model for language context based on user-task scenarios.

The cognitive activities involved in language are: (i)*Learning* both syntax and semantics; (ii)*Composition* of the syntax required to perform a function; (iii)*Comprehension* of the function syntax composed by someone else; (iv)*Debugging* of syntax (semantics) written by ourselves or others; (v)*Modification* of a function written by ourselves or others. Experimenters in human factors have developed a list of tasks to capture these particular aspects [19].

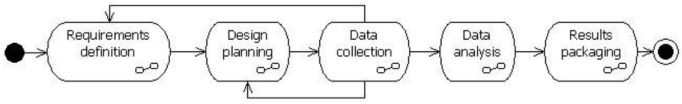

Figure 1. Experiment Activity Model Overview

Testing different tasks in the language usage is interesting, but to perform an exhaustive evaluation of them would be very expensive. Therefore, the evaluation should focus on the most critical activities.. In the case of Pheasant's evaluation, used as case study in this paper, the main concern was the task of *query writing* where users are given a question stated in natural language and have to write a sentence in the given language.

This is justified by the fact that the main function of a language is to provide is to provide users with an effective tool successfully perform some task. The goal was to know how easy it is to learn and use the language. Therefore, evaluation was restricted to three tests;

- *Immediate comprehension* - helps to identify why particular learning problems occur and they are given during teaching, immediately after some function has been taught, to determine whether the participants can use the function.
- *Reviews* - helps to identify why particular learning problems occur and they are given during teaching and cover functions taught up until that time. The participants are required to know which function to use.
- *Final exams* - tests how easily a language can be learned. These exams take place at the end of teaching the language under evaluation.

Usability evaluation is found as an important and beneficial activity in the UI development practice. It is recognized that usability must be considered from beginning of the development cycle using user centred methods. The objective of introducing user centred methods is to ensure that UI can be used by real people to achieve their tasks in the real world. This requires not only easy-to-use interfaces, but also the appropriate functionality and support for real business activities and work flows. Developing easy-to-use products makes business effective; makes business efficient; makes business sense [4]. User centred design can increase sales,reduce development, support costs and staff costs for employers.

5. Capturing achieved quality in use: a Pheasant case study

To illustrate how to evaluate the achieved Quality in Use of a DSL, we present an example of a visual query language for High Energy Physics (HEP) called Pheasant [1]. The goal of Pheasant's development was to improve the efficiency, reduce the error rate and have a less steep learning curve then the existing GPL.

The target users of the Pheasant language are specialists in HEP, with varying experience in software development. The evaluation was performed according to the mentioned ISO 9241-11 usability definition, which is an essential part of the achieved Quality in Use.

5.1 The Evaluation process

Fig.1 outlines the activities needed to perform the Pheasant language evaluation, following the scientific method. A detailed discussion on how this process can be followed in a software engineering experimentation context can be found in [7]. During *Requirements definition* problem statement (i.e. research questions),

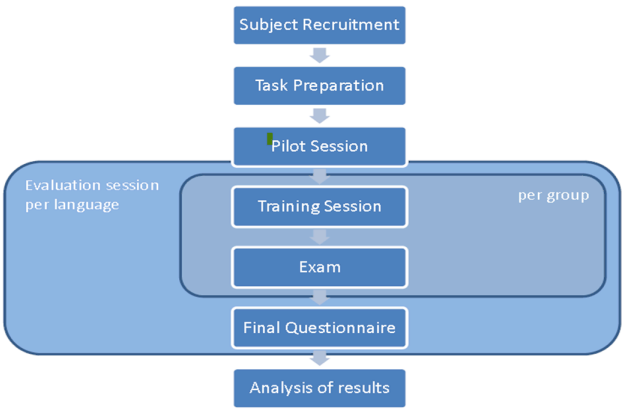

Figure 2. The evaluation process steps

experimental objectives and context are defined. The next step is *Design planning* where context parameters and hypotheses are defined, subjects and the sequence of observations and treatments are identified, and the data collection activities plan is set. This is followed with *Data collection*, which includes a pilot session, to correct any remaining issues, and the evaluation itself, following the designed plan. This step is followed with *Data analysis* where data is described in the form of statistical tables and graphs, and, if necessary, the data set is reduced. Hypotheses are then tested. During *Results packaging*, the results are interpreted and possible validity threats and lessons learned are identified.

The evaluation process followed in this case study is presented in *Fig.2*. The process starts with the **Participant Recruitment**, where the users are analyzed and grouped into clear categories. This way, the variables concerning the user profile that lead to different results for different groups are controlled. This step is followed by the **Task Preparation**. The aim here is to organize the evaluation by determining which tasks have to be done and which tests are elaborated in order to provide the proper results. This will generate the information required to be analyzed afterwards. The next step is the **Pilot Session**, which is meant to simulate the exam and test that the material for the training and the evaluation procedures is well organized. The main advantage of this rehearsal is to check that the time constraints and other possible external variables like proper equipment are controlled, and do not interfere with the results. Once everything is tested, we proceed to on the assessment, which we call **Evaluation Session**, for each group and language being compared. A **Training Session** is used to introduce the language. At this stage, *Immediate Comprehension* and *Review tests* are conducted with participants, while introducing the language features. The final exams, in the **Exam Session**, involve sentence writing activities. During the exam session, participants' activities are observed and recorded, so that information such as completion times and error rates can be collected. The goal is to determine the ease of learning. After each group has been evaluated in the different languages, the participants are asked for a debriefing in the form of a **Final Questionnaire Session**. The goal is to obtain the user's qualitative perspective of the comparison between the languages. In order to evaluate unbiasedly, the users should test the same environment and as realistically as possible. Evaluation process terminate with **Analysis of Results**.

5.2 Subject Recruitment

For this case study we identify two types of physicists involved, according to the context of HEP experiments:

1. *informed programmers (Inf)* are regular users of programming languages such as C, C++, Java or Fortran and they are used to program with the present analysis framework,
2. *uninformed programmers (non-Inf)* are regular users of programming languages such as C, C++, Java or Fortran and they are not used to program with the present analysis framework.

We wanted to use a third group that would consist of non-programmers but finding enough available physicist which were able to participate in this assessment turned out to be a problem. We used two different groups of programmers since the informed ones may introduce a bias on the learning phase of the compared query methodologies. This assumption is taken into account even if uninformed programmers are the majority of the population in the experiment.

At the end, fifteen graduate students were assigned to the proper group with an interview and an analysis of the participant's previous experience, to minimize the risks of biases that might otherwise be introduced by participants in a self-evaluation.

5.3 Task Preparation

Johnson [10] suggests that six individuals per subset of the population is the minimum required for a controlled experiment. Of course it is sensible to take a larger number, but the costs should be kept to a minimum. The task of gathering groups of six persons in a HEP research lab is already nontrivial. All the participants should have a degree in physics or be near its completion at least, and they should be skilled in experimental analysis. A basic knowledge of programming concepts is mandatory, since this subject is taught in the first years of the physics courses.

Introducing one query system to the whole group of participants and only afterwards the other query system would bias the evaluation, as the knowledge acquired while learning the first language would be partially reused while using the second language. In order to mitigate this threat to the validity of the results we have to split the group in two. This way, we reduce the influence of the first language while presenting the second. Mixing the two groups might lead to new variables in the evaluation that are hard to track. Therefore, we had to organize four sessions, with each group taking part in two sessions (one for each language).

The features we wanted to have evaluated are:

- query steps in Phesant v.s. the object-oriented coding
- expressing a decay
- specification of filtering conditions
- vertexing and the usage of user-defined functions
- aggregation
- path expression (navigation queries)
- expressing the result set
- the expressiveness of user-defined functions

In this study, the *independent* variables are the subject's background and the language being used. The *dependent* variables are the time to finish the task, the error rate while doing it and the confidence in the successful completion of the task.

5.4 Pilot Session

Our evaluation technique was tested with two individuals (two physics experts) in order to verify it and to test the teaching materials and questionnaires. This also helped to avoid that the evaluation had to be redone from scratch because of uncontrolled external variables, like inadequate equipment or lab conditions, or time constraints that can interfere with the results. After pilot session there was no need for significant change in experiment materials.

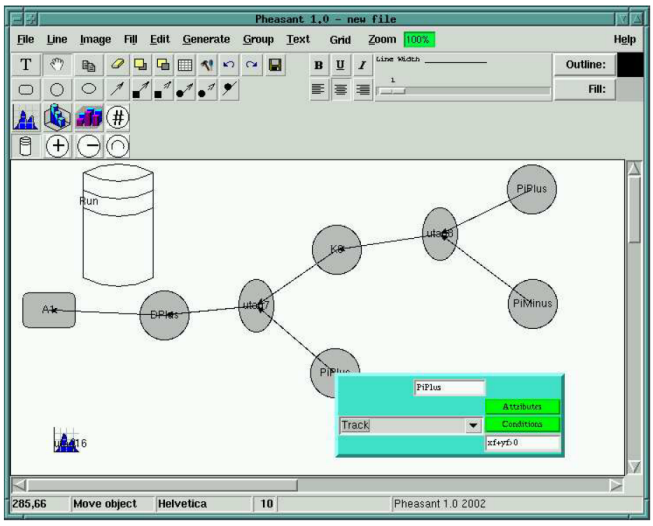

Figure 3. Query solution in Pheasant

5.5 The Evaluation Session

In the evaluation session, we try to answer the following:

RQ1: Is querying with Pheasant more effective than with C++/BEE?

RQ2: Is querying with Pheasant more efficient than with C++/BEE?

RQ3: Are participants querying with Pheasant more confident on their performance than with C++/BEE?

Our **goal** is to

- *analyze* the performance of Pheasant programmers plug-ins
- *for the purpose of* comparing it with a baseline alternative (C++/BEE)
- *with respect to* the efficiency, effectiveness and confidence of defying queries in Pheasant
- *from the point of view of* a researcher trying to assess the Pheasant DSL,
- *in the context of* a case study on selected queries.

We will test the following hypotheses:

- $\mathbf{H1_{null}}$ Using Pheasant or C++/BEE has no impact on the effectiveness of querying the analysis framework
- $\mathbf{H1_{alt}}$ Using Pheasant or C++/BEE has a significant impact on the effectiveness of querying the analysis framework
- $\mathbf{H2_{null}}$ Using Pheasant or C++/BEE has no impact on the efficiency of querying the analysis framework
- $\mathbf{H2_{alt}}$ Using Pheasant or C++/BEE has a significant impact on the efficiency of querying the analysis framework
- $\mathbf{H3_{null}}$ Using Pheasant or C++/BEE has no impact on the confidence of querying the analysis framework
- $\mathbf{H3_{alt}}$ Using Pheasant or C++/BEE has a significant impact on the confidence of querying the analysis framework

5.5.1 Training Session

Due to the complexity and the time constraints, we could not teach the complete C++ query language plus the interface of the analysis frameworks' libraries. Therefore, we focus on presenting six examples, each focusing in some of the features we chose to evaluate. The last query should make use of all the features taught in the session.

```
1)  Declare: List Runs , List Events, List Results
2)  Result is a list of particle1, particle2, Computed Vertex and Vertex
    # Step number 1
3)  while(run=nextRun()) {
4)      if(conditions) Runs.append(run)
5)  }
    # Step number 2
6)  foreach run in Runs {
7)      while(event=run.nextEvent()) {
8)          if(conditions) Events.append(event)
9)      }
    # Step number 3
10) Declare: List Particles and List Vertexes
11) foreach event in Events {
12)     Particles= event.GetParticle(conditions)
13)     Vertexes= event.GetVertex(conditions)
14)     particles=Particles
15)     While (particles.notempty()) {
16)         headParticle=particles.head()
17)         particles=particles.tail()
18)         foreach auxParticle in particles {
19)             if (condition(auxParticle) and condition(head, auxParticle)) {
20)                 Declare: distance=∞ and MinVertex={}
21)                 computedVertex=ComputeVertex(headParticle,auxParticle)
22)                 foreach vertex in Vertexes {
23)                     if(distant(vertex,ComputedVertex)<distance) {
24)                         distance=distant(vertex,Vertex)
25)                         MinVertex=Vertex
26)                     }
27)                 }
28)                 if(MinVertex.notNULL)
29)                     Result.append(headParticle,
30)                         auxParticle, computedVertex, MinVertex)
31)         }
32)     }
33) }
    # Step number 4
34) Histogram(userSetup, Results)
```

Figure 4. Query solution in C++/BEE (pseudocode based on a real query)

Murray [17] suggests that the participants should give themselves a mark for their feeling of correctness of their trial. This introduces them to the system of self-assessment. Besides, it helps the trainer to infer if there are difficulties experienced and an extra explanation is required. This session should take the time required for each group to understand the six examples.

5.5.2 Exam

We have evaluated the participants' performance in the *query writing*. Every participant has four queries, specified in English, to be rewritten in the previously learned language. An example of a query solution for the task *'Build the decay of a D0 particle to a Kaon Pion'* is given in *Fig.3* for Pheasant and *Fig.4* for C++/BEE (presented here in pseudocode, for the sake of readability)..

At the end, the subject makes a self-assessment of his reply by rating his feeling of the correctness of the answer. For each of the queries, we measured the time taken by each participant to reply in time slots of 15 minutes.

5.5.3 Questionnaire

After each session, the participants ware asked to judge the intuitiveness, suitability and effectiveness of the query language. The goal was to evaluate:

- *Overall reactions* - to obtain an overall reaction to one of the query languages through queries.

- *Query language constructs* - with the participants rating how easily specific aspects of the query language are to use.

After the tests ware completed, the participants ware asked to compare the two query languages. It is rated which query language they prefered, and into what extent.

- *Query language comparisons* - the participants are asked to compare specific aspects of both query languages and rate the preferences they have.
- *Participants' comments* - allows the participants to comment freely on the query language.

Since with the evaluation questionnaire we can only identify problems but not infer how to solve them, we ask the participants to contribute creative comments. Sometimes improvements are obvious and the comments can be fruitful. Therefore, after the evaluation session the participants are asked to write down informal comments and suggestions for improving the language.

5.6 Results analysis

In this section, we summarize the most relevant results of our evaluation tests. First, we deal with effectiveness by having a look at the test results with regard to the errors produced by the user while interacting with both evaluated approaches. Then, we will describe the results related to efficiency, which are mainly concerned with time measurements. At the end we analyse results concerning the confidence level of the participants that is measured in terms of their self-assessment.

In order to assess whether the observed differences with respect to the effectiveness and efficiency of using Pheasant, when compared to C++/BEE are statistically meaningful, we performed a Wilcoxon matched-pairs signed-rank test, as well as a sign test. These are adequate for testing our sample, as our data is ordinal. Answers ranked with 0 in correctness are used in correctness comparison (as they are meaningful to contrast the success with each of the languages - 0 means developers were not able to produce the query). However, with respect to the amount of time taken to answer, and the confidence of developers in their answer, answers ranked as 0 are considered missing answers. When they were unable to build a query, participants did not fill in the information concerning the time spent trying to build the query, nor the information concerning their confidence in their answer.

The results obtained with Pheasant were clearly better than those with the existing alternative. In order to reduce the variables that could influence the results, the queries were explained orally by an expert. This reduces the required interpretation time (which has a significant impact, especially in the group of uninformed programmers). Code re-usage was not allowed, although the subjects could use all the necessary documentation and especially the notes from the training session.

5.6.1 Effectiveness

Effectiveness and user accuracy can be assessed by observing results of the errors produced by the user while interacting with both evaluated approaches. As it can be observed in the histograms of *Fig.5*, or more detail in *Table 1*, while using C++ as a query language, the error rate was tremendous for novice users. We must state that the user did not have any sort of feedback from the system execution in order to spot the mistake and correct it before it came to the hands of the evaluator. In his daily life, the user tries to execute the algorithm and watches the result data after the execution. Then, in a cyclic way, he corrects himself and runs the query against the storage base. This is one of the main reasons why the query generation in the physics analysis phase is so time consuming. We can also observe that different groups of users get different results. As expected, their quality is directly proportional to the user's experience. Some of the most complex queries were not even tried due to the fact that they were difficult for uninformed users.

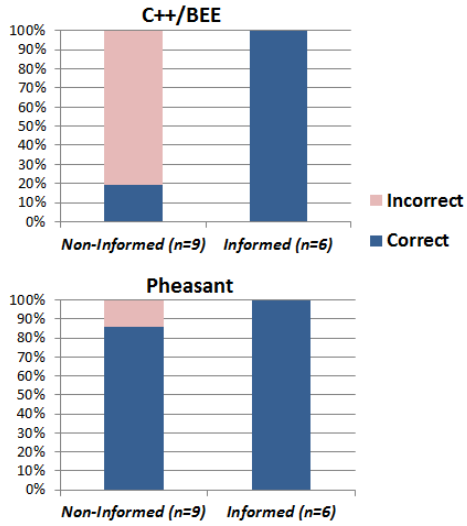

Figure 5. Effectiveness

As far as the Pheasant Query language is concerned, the results are much more promising. As the query mechanisms are much simpler and controlled, we do not observe invalid queries, and only a few wrong answers (which can be explained by some inexperience of the users in doing the analysis itself). Generally, the results show that the user did not have to essentially change the way he thinks about the query generation, which means that we have reached the goal of introducing a query language closer to the physicist's conceptual level of analysis.

Table 1. Error analysis - percent values

C++ BEE	Non-Inf	Inf
Correct	2,78	54,17
Minor data error		33,33
Minor language error	16,67	12,5
Essentially correct	*19,45*	*100*
Wrong answer	30,55	
Invalid	11,11	
Not attempted	38,89	
Totally incorrect	*80,55*	*0*
Pheasant	Non-Inf	Inf
Correct	80,5	95,83
Minor data error		
Minor language error	5,5	4,17
Essentially correct	*86*	*100*
Wrong answer	11,11	
Invalid		
Not attempted	2,89	
Totally incorrect	*14*	*0*

According to statistical analysis, presented in *Table 2*, the observed differences are statistically significant according to both tests. Also for both cases; when we analyse each query separately, as well when we look at them all together. These tests lead us to reject the null hypothesis that the obtained effectiveness is similar when using Pheasant and BEE/C++, and accept the $H1_{alt}$.

Table 2. Statistical analysis for effectiveness

	Q1	Q2	Q3	Q4	all
	Wilcoxon Signed Ranks Test				
Z	-3,097	-2,714	-2,949	-3,037	-5,833
Exact Sig.	,002	,007	,003	,002	,000
	Sign test				
Exact Sig.	,000	,004	,001	,003	,000

5.6.2 Efficiency

From our time analysis in *Fig.6* and *Table 3*, it becomes clear that more time has to be spent learning and using C++/BEE than with Pheasant. This can be justified by the complexity of C++ and the BEE library. At the same time, the test participants had less confidence in the quality of his/her query. This subjective impression is confirmed, as we have seen, by the huge error rate when using BEE.

Table 3. Time analysis - percent values

		Training time *(min)*	Mean total exam time *(min)*	Mean confidence / query *(5-0)*
Non-Inf	**C++ BEE**	190	80	1,04
	Pheasant	130	65	4,75
Inf	**C++ BEE**	0	110	4,88
	Pheasant	60	60	4,83

According to the results obtained in *Table 4*, the observed differences are statistically significant according to both tests. These tests lead us to reject the null hypothesis that the obtained efficiency is similar when using Pheasant and BEE/C++, and accept the $H2_{alt}$.

Table 4. Statistical analysis for efficiency

	Q1	Q2	Q3	Q4	all
	Wilcoxon Signed Ranks Test				
Z	-2,887	-3,000	-2,762	-2,392	-5,298
Exact Sig.	,004	,003	,006	,017	,000
	Sign test				
Exact Sig.	,004	,004	,004	,016	,000

5.6.3 Confidence

The test participants were supposed to rate how they were satisfied with the realization of each feature in the corresponding framework. Our goal was to identify potential weaknesses of each framework. As we can see in *Table 3*, non-Informed participants were much more confident while using Pheasant than C++/BEE. As for the informed programmers, their confidence level is almost the same with both languages. This can be regarded as a success for Pheasant. With little experience in the new language, participants felt as confident with it as with the one they were used to working with, meaning that they found the new language easy to learn.

According to the analysis presented in *Table 5*, the observed differences are statistically significant according to both test, with exception of the confidence in answering questions 3 and 4. This is likely due to a relatively higher difficulty in answering these last two questions, particularly with BEE/C++. This eventually led to the situation where uninformed programmers did not want to

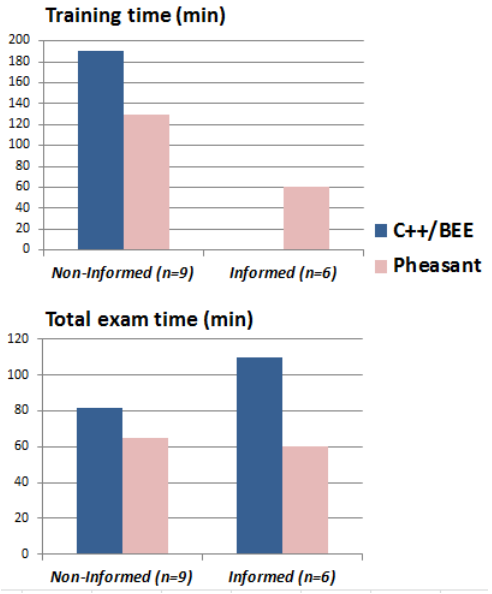

Figure 6. Efficiency

record their confidence in their answers. Because they were not able to come up with answers in BEE/C++. As we had no confidence information to compare with, for several users in these two questions and those who answered were, in general, the users with BEE/C++ expertise, the difference of confidence for these two questions follows the general trend, but is too small to be statistically meaningful. In contrast, when we aggregate the data for all questions, the advantages of using Pheasant are statistically significant. In summary, these tests lead us to reject the null hypotheses that the obtained confidence level is similar when using Pheasant and BEE/C++, and accept the $H3_{alt}$.

Table 5. Statistical analysis for confidence

	Q1	Q2	Q3	Q4	all
	Wilcoxon Signed Ranks Test				
Z	-2,232	-2,232	*-,966*	*-,736*	-3,594
Exact Sig.	,026	,026	*,334*	*,461*	,000
	Sign test				
Exact Sig.	,031	,031	*1,000*	*,625*	,001

The enthusiasm towards the language was significant. The several comments focused more on implementation issues to improve interactivity and did not criticize the language itself. This is a typical situation in UIs when dealing with prototypes. It is explained by the fact that the prototype needs to evolve into the next engineering life cycle phase to result in a properly engineered software product. Only this way the product is able to provide a real analysis environment and the user can compare it in his daily life with the other alternative solutions. Although the system experts recognize that the solution is a more comfortable approach for analysis, they still worry that the query tool might be less expressive. In order to confirm or reduce these fears, we propose to carry out further tests on the feared limitations of the language, to capture if the subjects are able to write queries with the existing language constructs.

From the comments given by participants we can infer, for instance, that a query reuse mechanism should be provided in a final implementation solution. Also, a query history mechanism where

the user can browse on past queries and respective solutions, is an extra feature which might have a great impact on user satisfaction.

5.6.4 Interpretation

We have determined that, by using Pheasant, the users increase effectiveness during their query specification. It was shown that the DSL was less error-prone than the alternative, by observing that it allowed non-programmers to correctly define their queries. The evaluation also showed a considerable speedup in the query definition by all the groups of users that were using Pheasant. In general, the feed-back obtained from the users was that it is more comfortable to use Pheasant than with the alternative.

We find that the preliminary pilot study was fundamental to ensure that the subject's time was well spent. The valuable feedback of users concerning the tool support for the language, as well as their fears concerning language expressiveness support the idea of an iterative evaluation process where improvements to the language and its tool support would be performed, and then assessed in a new round of evaluation.

At this point, some legitimate questions might arise concerning to a Full-blown experimental process to evaluate a DSL, as the one we described here. Could it be that the overhead of organizing all these complex tasks is exaggeratedly too heavy compared to doing nothing? Are'nt there better, and similarly valid, lightweight alternatives to this evaluation process? Further research should be done in this direction.

6. Conclusions and Future Work

One of the main goals while producing a DSL should be to foster a more productive usage of that language by the users who will use it than the existing alternatives. The interaction should favor an increase in the 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 the environment, long periods of learning, assistance and maintenance.

Usability evaluation is most effective when it is done directly with users or in combination with expert evaluators, and the reliability of that approach usually requires lots of preparation work and a large number of people involved in it. Usability evaluation is perceived as costly and is often minimized in real-life language development processes. Nevertheless, the costs of poor usability are likely to exceed those of usability testing, in the long run. For GPLs, the user base is frequently potentially larger and more heterogeneous than the user base of a DSL so generalizing conclusions for a diverse population is harder (although, on the other hand, finding subjects for assessing a GPL is probably easier than for a DSL). Finally, it should be stressed that usability is just one of the important attributes in language evaluation. 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.

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

As future work, we will propose a 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.

References

[1] V. Amaral. Increasing productivity in high energy physics data mining with a domain specific visual query language. In *Phd. Thesis, University of Mannheim*, 2005. URL `http://madoc.bib.uni-mannheim.de/madoc/volltexte/2005/870/`.

[2] C. Atkinson and T. Kühne. Model-driven development: A metamodeling foundation. *IEEE Softw.*, 20:36–41, September 2003. ISSN 0740-7459. doi: 10.1109/MS.2003.1231149. URL `http://portal.acm.org/citation.cfm?id=942589.942704`.

[3] A. Barišić, V. Amaral, M. Goulão, and B. Barroca. Quality in use of dsls: Current evaluation methods. In *Proceedings of the 3rd INForum - Simpósio de Informática (INForum2011)*, Coimbra, Portugal, September 2011. University of Coimbra.

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

[5] T. Catarci. What happened when database researchers met usability. *Information Systems*, 25(3):177–212, 2000. ISSN 0306-4379.

[6] P. Gabriel, M. Goulão, and V. Amaral. Do Software Languages Engineers Evaluate their Languages? In *Proceedings of the XIII Congreso Iberoamericano en" Software Engineering"(CIbSE'2010)*, pages 149–162, 2010.

[7] M. Goulão and F. Brito e Abreu. Modeling the experimental software engineering process. In *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, Lisbon, Portugal, 2007. IEEE Computer Society.

[8] International Standard Organization. Iso/iec 9126-1 quality model, June 2001. URL `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749`.

[9] International Standard Organization. Iso/iec 9241-11 ergonomic requirements for office work with visual display terminals (vdts) – part 11: Guidance on usability, June 2001. URL `http://www.iso.org/iso/catalogue_detail.htm?csnumber=16883`.

[10] P. Johnson. *Human computer interaction*. McGraw-Hill, 1992. ISBN 0077072359 9780077072353.

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

[12] R. Kieburtz, L. McKinney, J. Bell, J. Hook, A. Kotov, J. Lewis, D. Oliva, T. Sheard, I. Smith, and L. Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th international conference on Software engineering*, page 552. IEEE Computer Society, 1996. ISBN 0818672463.

[13] A. Kleppe. *Software language engineering: creating domain-specific languages using metamodels*. Addison-Wesley, 2009. ISBN 0321553454.

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

[15] S. Markstrum. Staking claims: a history of programming language design claims and evidence: a positional work in progress. In *Evaluation and Usability of Programming Languages and Tools*. ACM, 2010.

[16] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[17] N. Murray, N. Paton, C. Goble, and J. Bryce. Kaleidoquery–a flow-based visual language and its evaluation. *Journal of Visual Languages & Computing*, 11(2):151–189, 2000. ISSN 1045-926X.

[18] L. Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, 2000.

[19] P. Reisner. Query languages. *Handbook of Human–Computer Interaction, North-Holland, Amsterdam, The Netherlands*, pages 257–280, 1988.

[20] J. Rubin and D. Chisnell. *Handbook of Usability Testing: How to plan, design and conduct effective tests*. Wiley-India, 2008.