Discount Method for Programming Language Evaluation

Svetomir Kurtev
Department of Computer Science,
Aalborg University
svetomirkurtev@gmail.com

Tommy Aagaard Christensen
Department of Computer Science,
Aalborg University
tommyaa@gmail.com

Bent Thomsen
Department of Computer Science,
Aalborg University
bt@cs.aau.dk

Abstract

This paper presents work in progress on developing a Discount Method for Programming Language Evaluation inspired by the discount usability evaluation method (Benyon 2010) and the Instant Data Analysis method (Kjeldskov et al. 2004).

The method is intended to bridge the gap between small scale internal language design evaluation methods and large scale surveys and quantitative evaluation methods. The method is designed to be applicable even before a compiler or IDE is developed for a new language.

To test the method, a usability evaluation experiment was carried out on the Quorum programming language(Stefik et al. 2016) using programmers with experience in C and C#. When comparing our results with previous studies of Quorum, most of the data was comparable though not strictly in agreement. However, the discrepancies were mainly related to the programmers pre-existing expectations of a language. The results show that our evaluation method could serve as a low-cost way of evaluating programming languages for language designers.

Future work include adjusting and improving the method in such a way it becomes usable to novice programming language designers.

Categories and Subject Descriptors D.3 [*Programming Languages*]; H.5.2 [*Information Interfaces and Presentation (e.g., HCI)*]: User Interfaces—User-centered design

General Terms Programming Language Design Evaluation, User Evaluation

Keywords Quorum, Usability Evaluation, Language Design

1. Introduction

Designing a new programming language or extending an existing programming language usually follows an iterative approach:

1 Create ideas for the programming language or extensions

2 Implement the programming language or extensions

3 Evaluate the programming language or extensions

4 If not satisfied, goto 1

An iterative approach to language design and implementation was advocated by Wirth as early as 1974 (Wirth 1974). Ingalls reports on a similar design process used for Smalltalk in 1981 (Ingalls 1981). Text books on programming language design and compiler construction, such as (Watt and Brown 2000), now describe such an iterative approach.

Step 1 of the process is hard to subject to scientific analysis. The decision to create a new programming language or design an extension of an existing languages is often ''a reaction to some language that the designer knows (and likes or dislikes)" (Sestoft 2012). Language designers use whatever means that are available to them during this step. Step 2 is by now a well understood topic, although ever so often, new language constructs or features, leads to an expansion of compiler implementation techniques. Step 3 is much more difficult. In the early stages of a programming language design, the developer may be guide by various informal principles such as readability, simplicity and reliability (Pratt et. al 1984; Sebesta 2016) or somewhat more formal principles such as Tennent's language design principles (Tennent 1981) which still guide many language designers (Sestoft 2012).

However, formal evaluation methods for assessing programming languages are few and limited in their use. In fact until recently most evidence gathered to support claims of a language or features usefulness are anecdotal in nature(Markstrum 2010).

In recent years, however, the scientific community has worked to rectify this. In particular, the focus of the PLATEAU conferences is the scientific evaluation of languages. The basic observation is that language use and preference is highly opinionated, which lead to user-based evaluation being the norm for programming languages. Commonly, the scientific community has made use of methods from social sciences, which usually requires studying a large number of subjects(Meyerovich and Rabkin 2012)(Garlick and Cankaya 2010)(Weintrop and Wilensky 2015)(Armoni et al. 2015). Some research have even experimented with techniques akin to those used in medical science, where a placebo language was constructed to be used in the comparison between Perl and Quorum (Stefik et al 2011).

However, these methods are rather expensive since conducting a quantitative test requires a large group of people. Typically, this means programming language designers cannot do these tests before the language has reached a certain level of maturity and, for some methods, the language need to have already gained widespread use.

Some designers have used more qualitative and lightweight approaches for language evaluation. A commonly used lightweight approach is the discount usability evaluation method (Benyon 2010). Some examples of using the method in practice are: The language HANDS developed by Pane et al.(Pane et al. 2002) designed specifically for children; Koitz and Slany(Koitz and Slany 2014) used it on Scratch and their phone language Pocket Code to compare the two; Faldborg and Pedersen used the method to test their spoken programming language LARM; Faldborg and Nielsen used it while conducting an empirical experiment on Dart and the web-enabled IDE called DartPad (Faldborg and Nielsen 2015).

The primary problem with the discount usability method in such context, as identified by Faldborg and Pedersen and Faldborg and Nielsen, is the difficulty of separating the feedback about the language design from the IDE. These observations, along with our own experiences, lead us to believe that the discount usability evaluation method is good when evaluating the full package of a language with its IDE and compiler, but is less suited for evaluations in the early phases of a language design.

Therefore we wanted to create a new evaluation method which will fill the gap between internal language analysis and the big and bulky industry programming evaluation methods(Garlick and Cankaya 2010)(Weintrop and Wilensky 2015)(Armoni et al. 2015). This new method is inspired by the discount usability evaluation and the Instant Data Analysis (IDA) methods, with the main difference being to avoid the use of a compiler or an IDE.

To test this method, we used the evidence-based programming language Quorum. This entailed conducting a qualitative experiment with six experienced programmers as participants. These experiments produced data, comparable though not strictly in agreement, with previously published studies. The discrepancies between our study and previous work mainly relate to the programmers pre-existing expectations of a language. We believe that our method would be a valuable, low-cost tool for user evaluation of programming languages.

Future work include adjusting and improving the method in such a way it becomes usable to novice programming language designers. As part of the Computer Science and Software Engineering Curricula at Aalborg University, students at the 4th semester have to design, define and implement their own programming language (Dolog et. al 2016). It is our hope that the Discount Method for Programming Language Evaluation becomes standard practice used by these students and hopefully elsewhere.

2. Method

The discount usability method was created by Andrew Monk et al. 1993 as presented in Designing Interactive Systems(Benyon 2010). To get acquainted with the discount usability methods applicability on programming languages, we first conducted experiments on C# and F# in Visual Studio. The participants we used were experienced programmers familiar with C and C#, but not familiar with F#. To analyse the data from the test, we used the IDA method (Kjeldskov et al. 2004), as it is a lightweight method for prioritising encountered problems.

Based on our findings, we use both methods as a basis for creating a new evaluation method suitable for evaluating languages without the need of a compiler or an IDE. The method mainly relies on solving a set of programming tasks with the help of sample sheet which demonstrates the use of relevant constructs from the language being tested. Additionally, an interview, either in written or spoken form, serves the purpose of further elaborating on the most interesting problems the observer has noted during the process. The procedure of the method looks as follows:

1. **Create tasks** These tasks are specific to the language, and should explore key features of the language. A useful tool to design tasks can be to create some scenarios you would expect a user to use your language in and what that user would need to do solve their task.
2. **Create a sample sheet** Based on the tasks, you now have a better idea of what a participant would need to know to solve those tasks. Keeping the sample sheet short or having a clear indexing of the samples can help participants browse the sample sheet. Having working code samples can help give a better understanding of the overall structure of code in the language.[1]
3. **Estimate the task length** Taking time of how fast you can solve the tasks will give an idea of how long the experiment will take per participant. Do note that the participants will likely take longer to solve the tasks since they have to get acquainted with the language first. It can be okay to have more tasks than what you expect a participant to be able to solve, but the later tasks would need to

[1] Note that sample created at this stage may later serve the purpose of test cases when a compiler, interpreter or IDE is implemented for a new language.

explore less important features, and the participant needs to be made aware of not being expected to solve all of them.

4. **Prepare setup** The specifics of the setup can vary from a full blown usability lab to pen and paper. The advantage of a flexible setup, like pen and paper or a laptop with a text-editor, is the convenience it allows for potential participants. Often the experiment will be recorded to better review the process of solving the tasks, in which case the necessary utilities for this needs to be prepared.

- (optional) **Conduct a pilot test** A pilot test can let you discover and fix problems in your tasks, sample sheet, task estimate and setup before conducting the experiment on the full number of participants. It does, however, require an additional participant and time.

5. **Gather participants** The golden rule for the number of participants is five. More than that and most of the encountered problems are ones you already have observed, though the repetition can reinforce observations. Less than that and you tend to have several problems left undiscovered, though some data is generally still better than none.

6. **Start the experiment** Make sure to tell the participant that it is the language being tested and not them, to alleviate some unnecessary nervousness.

7. **Keep the participant talking** Try to make the participant talk about what they are thinking about solving the task at hand. During this time the facilitator may answer any questions the participant have about the language. The facilitator should try to avoid talking about how to solve the tasks, but it may be necessary if the participant need help getting started (or stopped in cases of overcomplicating tasks). The facilitator will confirm when a task is done. This is necessary because the system will not give that kind of feedback.

8. **Interview the participant** After the test, have a brief interview with the participant where you can discuss the language, tasks etc. It can be useful to have some questions pre-written or create a questionnaire if there are many participants.

9. **Analyse data** After all the tests have been conducted, use the data to identify a list of problems encountered during the test. You can then categorise the problems using the following guidelines:

   Cosmetic problems are typos and small keyword and character differences that can easily be fixed by replacing the wrong part.

   Serious problems are structural errors that usually impacts how the code is structured, but is usually small enough that it can be fixed with a few changes.

   Critical problems are fundamental misunderstandings of how the language structures code and large structural errors that would require a revision of the algorithm.

   Following this categorisation you will now have a prioritized list of things to improve on the language.

Our method has two major differences from the discount usability method using the IDA method for evaluation. The first big difference is the addition of a sample sheet step. The reason for this is that without an IDE or a compiler the language would be presented as blank paper, which does nothing to teach the user about how programs in the language are written. Providing the user with examples and explanations of how the language works is necessary to let them meaningfully program in the language.

The other big difference is how the problems are prioritised. The IDA method uses time spent on overcoming a problem to categorise its severity. Our setup makes the user either work in a generic text editor or on a piece of paper neither of which give any feedback about the correctness of the code written. This means the participants would not necessarily discover any problems in their solutions and would therefore not spend time on solving those problems. This makes time a poor measure for problem severity, which caused us to instead estimate the severity based on how much code would need to be changed to fix the problems.

2.1 Experiment setup

The new method was tested in an experiment setup by using it on an unfamiliar programming language to avoid bias from pre-existing knowledge about the language. We used the evidence-based programming language Quorum, which our participants were unlikely to be familiar with. Most of our participants were experienced programmers, studying for a degree in Computer science, and having knowledge about C and C# and several programming languages (e.g. Java, F#, Python, Pascal). We had a total of six participants taking part in the main experiment, with one additional person as a pilot test participant.

The experiment was conducted using a text-editor on a laptop. The main reason for this over pen and paper, was to make recording easier. The text-editor we used was Notepad++(Ho 2016). Notepad++ has some features to assist programming, most notably an auto-completer which uses words already written in the text as suggestions. However, since these features are language-agnostic and the auto-completer would only prevent false positives from minor typos and not language misunderstandings, this was deemed acceptable. To record the screen Microsoft Game DVR was used. Due to the poor quality of the inbuilt laptop microphones, a smartphone was used to record the audio.

2.1.1 Task Sheet

For our task sheet, we devised several smaller tasks, each addressing different features and constructs of Quorum. We drew heavy inspiration for some of the tasks from Codekata (Thomas 2016) since some of the katas were simple to understand yet conveyed the essence of a particular feature, present in the tested language. Although each task had an intended purpose with a clear goal, their design allows more than one possible solution which gave the participants the freedom to experiment with the language.

- The first task had the intended purpose of testing arithmetic expressions and the use of data types.
- The second task had the purpose of testing containers in the language (such as arrays) and control structures. It also tested responsible code modification since there was a certain degree of intended repetitiveness in the subtasks which warranted careful reusing of code segments .
- The third task was for testing the concept of classes and inheritance.
- The final task was testing operations on strings, including the exercise of in-build actions specifically useful for splitting text segments.

The tasks were scheduled to be solved in about an hour, though the final task was expected to extend beyond that timeframe.

2.1.2 Sample Sheet

Alongside the task sheet, a sample sheet was created in order to provide examples of code that the participant could use to learn what was necessary from the language in order to solve the tasks. Working samples of code were used since it tends to give a more wholesome picture of how the code should look, without resorting to detailed description of how everything works.

2.1.3 Interview sheet

For the interview, we created an interview sheet with five questions addressing the overall experience of the experiment. These questions were not meant to replace the open discussion but rather serve as a baseline for the direction of the discussion and to ensure some specific areas were covered in the discussion. Questions #1, #2 and #3 were about the language and primarily served to get the participants own thoughts about it. While there were some potential overlap in these questions, they could help some people talk more, and they helped categorise the feedback. Question #4 focused on getting feedback about our task and sample sheet. Question #5 asked about the experience of coding without a compiler since it is the biggest change for our method. During the interview, the subject would usually be made aware of most of the otherwise unmentioned errors, to be able to provide a more informed discussion.

3. Results

In this section, we describe and divide the identified problems, according to the categorization described earlier. Table 1 lists all the problems in their respective category, but for the sake of brevity, we will focus only on the most interesting ones.

1. **Not using the `end` keyword at all** - this would affect the overall validity of the program because the scoping rules in Quorum are defined in conjunction with the `end` keyword. This shows a fundamental misunderstanding of how scoping works in the language
2. **The lack of constructors with parameters in Quorum** - Quorum does not support constructors with parameters which might be problematic for the participants, having experience with other languages where this feature is common. It both causes the participants to invoke syntax in the class that is not supported, and have difficulties instantiating classes. Since this is a significant difference in how the code should be structured, it is considered critical.
3. **Misunderstanding the effect of `Sort()` on arrays of objects** - The inbuilt sorting function for arrays does not have access to the properties of the objects and therefore does not sort them by any of those. This would have the consequence of code, written with the assumption that it works, be most likely wrong, which means recovery would require a full rewrite of the algorithm. This makes the problem critical. It is possible that with the use of a compiler the participant would discover and recover much easier, which could mean the problem would potentially be considered serious.

3.1 Comparison with Quorum's Evidence

In their empirical experiments, Stefik and Gellenbeck (Stefik and Gellenbeck 2011) gathered many statistically significant results regarding keyword choices. For one of their experiments (Stefik and Gellenbeck 2011), they divided the participants in two groups - novices and experienced programmers to find out if there is a significant discrepancy in the results between the two groups. Every keyword choice was ranked in two tables by mean value and standard deviation. Since we conducted our experiment with people having programming experience, we are primarily interested in the results of the second group. For the purpose of our evaluation method, we will not mention every single word choice they rated but rather the ones which are coinciding with the problems we identified using our method. Additionally, we would also relate two of the empirical studies from Stefik and Siebert (Stefik and Siebert 2013) and their findings about keyword choices. This would help us to make a comparison between the previous findings and the results from our evaluation.

Critical	**Serious**	**Cosmetic**
Not using the end keyword at all	Not using the end keyword to end the scope of if-statements	Using keywords and symbols from C# instead of Quorum[a]
The lack of constructors with parameters in Quorum	Forgetting to increment the iterator in a repeat while loop	The lack of aggregate operators in Quorum (e.g.+=)
Misunderstanding the effect of Sort() on arrays of classes	The lack of common looping constructs (for-loops or foreach loops)	Writing output instead of return as the keyword for a return statement
	Forgetting to import a library for containers (array)	Typos in library importing
	Not using elseif to avoid having to close an additional scope	Mistyping integer as integar
	Not resetting inner loop iterator between loops	Accidentally used 0 instead of O in variable name
		Mistyped the is keyword as ia
		Forgot to add the repeat keyword

Table 1. The table of identified problems categorised by severity

[a] dot (.) instead of colon (:), && instead of and, || instead of or, float instead of number, string instead of text, == in conditional statements instead of =, int instead of integer and bool instead of boolean

- In the results for the concepts of AND, OR and XOR, Stefik and Gellenbeck (Stefik and Gellenbeck 2011) found that for the AND concept, using && and and performed quite well. Their results showed that these words are actually popular and thus intuitive to use. As for the logical OR concept, the or keyword was placed first, being significantly better the the second highest one - the || operator, which is present in many popular programming languages. Last but not least, the XOR logical operator, the or was rated highest which can be attributed to the fact that the participants did not know how to call an operation which ”took a behavior when one condition was true but not both”.
- Stefik and Gellenbeck (Stefik and Gellenbeck 2011) settled on using a single equals (=) sign for assignment statements and for testing equality since that is what they thought would make most sense. Although this might be true for the novice group (single equals (=) sign was ranked highest), the experienced programmers group did not even rate the single equals in the top 3, rating the double equals (==) as highest instead. This was not verified until one of the later empirical studies by Stefik and Siebert (Stefik and Siebert 2013).
- For the concept of ”Taking a behaviour” the authors considered several word choices such as function, action and method. The novices ranked the action word the highest, while the experienced programmers - operation, followed by action, method and function. However, the authors admit that this particular results should be further investigated, since the participants might have understood the description of the concept as something other than completely capturing the idea of a function.
- Quorum makes use of the keyword repeat over for, while or cycle (see Sanchez and Flores (Snchez and Aguayo 2005)) following a study which shows that repeat represents the concept of iteration significantly better than the the aforementioned words (Stefik and Gellenbeck 2011).

Based on the results from Stefik and Siebert (Stefik and Siebert 2013), the programmers group found == to be intuitive as the boolean equals operator, which matches our observation of the participants often using == instead of = notation. For many of our other problems where our participants used the wrong syntax, Stefik and Siebert(Stefik and Siebert 2013) did have comparable results were both the wrong and the correct syntax was found intuitive by their programmers group. These are:

- Dot (.) versus colon (:)
- using an aggregate operator (x += 1) versus an arithmetic operator (x = x + 1)
- && versus and for logical AND
- || versus or for logical OR

- data types wording (`float` versus `number`, `string` versus `text` and `bool` versus `boolean`)

It is possible that a lot of these cases were the result of a participant just glancing over the sample sheet, instead of having a more thorough introduction to the syntax on the sample sheet. Since the constructs looked intuitive, they did not notice or remember that it was different and thus just used the syntax they were used to from other programming languages. Interestingly, our results about the looping constructs contradicts the results from (Stefik and Siebert 2013). Our participants often lamented the lack of a *for* or *foreach* loop and had a lot of errors in using the iterator for the *while* loop. However, the results are not directly contradictory as the paper's questions about intuitiveness was focused on the syntax, while our participants problems were more about lacking the functionality of a looping construct with inbuilt iterator handling. A more direct contradiction is that our participants often found the `repeat` keyword unnecessary, despite the paper listing it as one of the most intuitive keywords for looping. This could be a side effect of us only demonstrating the *repeat while* loop in our sample sheet, since that loop looks exactly like the *while* loop they are used to but with an extra keyword in front.

A lot of our results were unsurprising. Quorum is a language that uses evidence about programming to design a language that is intuitive for novices. Since our participants were experienced programmers though, it would be expected that a lot of the errors encountered would be related to this mismatch. Especially the critical error with lacking constructors with parameters, showed a large mismatch in what an experienced programmer expected from a class compared to what was shown to be more user-friendly(Stylos and Clarke 2007). Likewise the lack of a for- or foreach loop and the resulting iterator handling problems experienced by our participants, showed that they had a habit of handling the iterator in the looping construct. This functionality, however, could make the construct less practical for novices, as they might get a better understanding of the same functionality by writing the statements separately. More surprisingly we had a participant who never used the `end` keyword. In the discussion he explained this was because he thought indentation was used to control scope. He felt that since indentation is a good practice that all programmers should use anyway, it would make sense to make the language use and enforce this. This would be especially true for a beginner language, as the beginners are those who need to learn to use indentation. This again showed that experienced programmers were likely to draw from their previous experiences rather than thoroughly examine the sample sheet.

4. Discussion

The results from the previous section addressed some potential problems encountered when working with a language such as Quorum. This section will elaborate on how that can be extended to other programming languages and how our method could be used in a customized manner.

Comparing our results with Quorum's evidence has shown that our method gets comparable results to other methods, but with a significantly lower amount of participants. Most of Quorum's evidence about experienced programmers has, however, been focused on just the syntax. This means that most of the comparable data lies in our cosmetic errors, which are usually the least interesting problems from a usability standpoint. The more serious problems tend to either contradict or not be addressed by Quorum's evidence, though in most cases, this is a result of the mismatch in target group between novice and experienced programmers. One noticeable problem we encountered in the execution of our test was participants freezing at the very beginning. They were unsure how to start as they could not figure out what format of the solution they should use. For most of the participants this was not a big hurdle, as they would either just pick one way of doing it or consult the facilitator. However, for some participants, having a discussion while programming was unnatural. One way to prevent this could be to have some pre-written code that the participant should instead fill out. It does however sacrifice some of the potential data about the language that a more free-form task can give.

After conducting both of our experiments, a very interesting observation can be made that an Integrated Development Environment (IDE), when used in conjunction with a programming language, contributes primarily to resolving cosmetic problems and mistakes associated strictly with the syntax of the given language. However, An IDE does not contribute that much to the facilitation process if the user gets "stuck", a state attributed to the critical problems from the IDA evaluation. Additionally, we noticed that even if people are experienced with a specific paradigm (participant #2 from the usability evaluation in regards to C#), they can still get into a position where they cannot continue with the experiment, given that they have to solve a task in a paradigm, different from what they are familiar with (participant #2' experience with the F# task). Further observations from the evaluation method showed that when people familiar with a specific paradigm (imperative, object-oriented) have to make use of an unfamiliar language, supporting such paradigm (Quorum is both imperative and object-oriented), they tend to disregard the syntax of the new language in place of a language they are familiar with. In the case with Quorum, most of the participants (#1, #3, #4, #5, #6, #7 and #8) made use of syntax native to languages such as Java and C#, supporting the same paradigms as Quorum.

5. Threats to validity

Conducting the experiment had some informal and qualitative conditions, which makes the validity face some threats. This section will describe the most prevalent of such threats.

- Participant sample - Although we collected some qualitative results, the experiment did not have a good representation of the general populace. The participant sample involved a small group, with very similar educational backgrounds, occupation, age and geological location. The involvement of a bigger and more diverse group might skew the results in a different direction.
- Facilitating the participants - Since we did neither use a compiler nor an IDE for the experiment, the facilitator had to help on several occasions and the participants referred to him rather than the sample sheet.

6. Conclusion

We identified a gap in programming language evaluation between internal language analysis and the big industrial evaluation methods. In that regard, we wanted to explore whether HCI techniques could be applicable for the evaluation of programming languages.

In the context of HCI techniques, we have examined the usefulness of the discount usability method (**??**) with the IDA method (**??**) for data analysis on C# along with examining literature for similar experiments. In this examination we found that the discount usability method can be used to evaluate programming languages, but has some shortcomings. The programming languages IDE has a large effect on the results, often providing significant assistance which effectively eliminates many of the errors which might otherwise get caught in the language. Based on the results, we believe that the discount usability method is good for testing a compiler and an IDE, but is less well suited for examining language design.

To create a method better suited for evaluating language design, we conducted an adapted usability experiment where we specifically avoided the use of an IDE or a compiler (**??**). An added advantage of such a method is that it does not require the creation of any tools for the language before the language design can be tested, making it a low-cost and efficient solution. For this experiment, we used the evidence-based language called Quorum, as it was less likely for our participant group to be familiar with it, yet it belongs to a programming paradigm they were familiar with. We changed the way the problems are prioritised due to the discovery that since the system does not have a way of giving meaningful feedback to the participant, the participants would not encounter problems nor spend time fixing them.

Comparing the resulting data from the method with Quorums evidence showed us that most of the data was comparable though not strictly in agreement. However, Quorums data was mostly centered around syntax choice and therefore was mostly only related to the cosmetic problems, which are the least interesting problems to consider. The data suggests that our method will be better suited for getting some of the deeper problems with a programming language when compared to the syntax questionnaires used as evidence for Quorum.

We believe that we have created a method that can help fill the gap in programming language evaluation between internal language analysis and the big industrial evaluation methods.

7. Future Works

The results from the evaluation method show that this is a viable way of evaluating a programming language in a low-cost setup. However, there are still plenty of opportunities which can be taken into consideration for improving the method. This section provides a discussion of the most promising things that can be done in the future.

One of the things that could be done would be to conduct the usability experiment on Quorum, still with experienced programmers, using the Sodbeans environment. This would give a more direct comparison of the differences in the data gotten from using the usability experiment method versus our method.

Of course, simply conducting our method on more languages, and ideally by the language designers of these languages, would also give a lot of data about the method. One way to facilitate this could be to spread the method to the 4th semester students in the engineering faculty (SW, DAT, IT) at Aalborg University. Since these student have to design a language as part of their semester project, and could use the data to argument for their language design decisions, it would be an opportune way of testing the method in a low-risk environment.

While our method is good for finding problems in a programming language, it is less well suited to be used to compare the quality of programming languages. It could be interesting to look at creating a method designed for that purpose, as such comparisons are commonly of interest to language designers looking to promote their language over existing ones. One way of creating such a method from our method, could be to create a set of generalised tasks that would be applicable on all programming languages, which would give a solid common ground for the comparison.

Another thing worth exploring is using our method on novices. Using experienced programmers makes it easier to convey how to program in a language, as they already know how to program, and it makes sense when programmers are the target group for the language. It does, however, mean that the data tend to be biased towards the languages the programmers already know. Using novices avoids this bias and is obviously useful for languages designed for them.

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

We would like to thank all of the students who were willing to share their time with us as participants in our experiments.

References

S. Markstrum. Staking Claims: a history of programming language design claims and evidence: a positional work in progress. In *PLATEU '10 Evaluation and Usability of Programming Languages and Tools*, 2010.

J. F. Pane, B. A. Myers and L. B. Miller. Using HCI Techniques to Design a More Usable Programming System. In *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC02)*, 2002.

R. Koitz and W. Slany. Empirical Comparison of Visual to Hybrid Formula Manipulation in Educational Programming Languages for Teenagers. In *PLATEAU '14 Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, 2014.

L. C. Pdersen and M. Faldborg. Designing LARM: Programing with Nothing but your Voice. In *Det Digitale Projektbibliotek*, 2014.

M. Faldborg and T.L. Nielsen. Type Systems And Programmers A Look at Optional Typing in Dart. In *Det Digitale Projektbibliotek*, 2015.

D. Benyon. Designing Interactive Systems - A comprehensive guide to HCI and interaction design. Published by *Pearson Education Limited*, 2010.

J. Kjeldskov, M. B. Skov and J. Stage. Instant data analysis: conducting usability evaluations in a day. In *Proceedings of the third Nordic conference on Human-computer interaction*, 2004.

L. A. Meyerovich and A. S. Rabkin. Socio-PLT: principles for programming language adoption. In *Onward! 2012 Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, 2012.

M. Armoni, O. Meerbaum-Salant and M. Ben-Ari. From Scratch to Real Programming. In *ACM Transactions on Computing Education (TOCE) TOCE Homepage archive Volume 14 Issue 4, February 2015*, 2015.

D. Weintrop and U. Wilensky. To block or not to block, that is the question: students' perceptions of blocks-based programming. In *IDC '15 Proceedings of the 14th International Conference on Interaction Design and Children*, 2015.

R. Garlick and E. C. Cankaya. Using alice in CS1: a quantitative experiment. In *ITiCSE '10 Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*, 2010.

D. Ho Notepad++ home page. URL *http://notepad-plus-plus.org*, 2016. Used: 11/05/16.

D. Thomas Codekata. URL *http://codekata.com/*. Used: 12/05/16.

A. Stefik and E. Gellenbeck Empirical studies on programming language stimuli. In *Software Quality Journal*, 2011.

A. Stefik, S. Siebert, M. Stefik, and K. Slattery, An empirical comparison of the accuracy rates of novices using the quorum, perl, and randomo programming languages. In Proceedings of the *3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*, pp. 3-8, ACM, October 2011.

A. Stefik and S. Siebert An Empirical Investigation into Programming Language Syntax. In *ACM Transactions on Computing Education (TOCE)*, 2013.

J. Snchez and F. Aguayo Blind learners programming through audio. In *CHI EA '05 Extended Abstracts on Human Factors in Computing Systems*, 2005.

J. Stylos and S. Clarke Usability Implications of Requiring Parameters in Objects' Constructors. In *ICSE International Conference on Software Engineering*, 2007.

N. C. C. Brown, M. Klling, D. McCall and I. Utting Blackbox: A Large Scale Repository of Novice Programmers Activity. In *The 45th SIGCSE technical symposium on computer science education (SIGCSE 2014)*, 2014.

A. Stefik, E. Pierzina and K. Ritter Quorum's home page. URL *http://www.quorumlanguage.com*. Used: 11/05/16.

N. Wirth, On the Design of Programming Languages. In *IFIP Congress (Vol. 74, pp. 386-393)*, August 1974.

D. H. Ingalls, Design principles behind Smalltalk. in *BYTE magazine, 6(8), pp. 286-298*, 1981.

P. Sestoft, Programming language concepts *Springer Science & Business Media (Vol. 50)*, 2012.

T. W. Pratt, M. V. Zelkowitz and T. V. Gopal, Programming languages: design and implementation Prentice-Hall, 1984.

H. W. Sebesta, Concepts of Programming Languages. Pearson College Division, 2016.

R. D. Tennent, Principles of programming languages, Prentice Hall PTR, 1981.

D. A. Watt and D. F. Brown, Programming language processors in Java: compilers and interpreters, Pearson Education, 2000.

P. Dolog, L. Leth Thomsen, and B. Thomsen. Assessing Problem-Based Learning in a Software Engineering Curriculum Using Blooms Taxonomy and the IEEE Software Engineering Body of Knowledge. *Trans. Comput. Educ*. 16, 3, Article 9 (May 2016).