Discount method for programming language evaluation

Svetomir Kurtev and Tommy Aagaard Christensen

1 February - 28 June

Date	Svetomir Kurtev
Date	Tommy Aagaard Christensen

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Computer Science
Selma Lagerlöfs Vej 300
Telephone 99 40 99 40
Telefax 99 40 97 98
http://cs.aau.dk

Title:

Discount method for programming language evaluation

Project period:
1 February - 28 June

Project group:
dpt108f16

Participants:
Svetomir Kurtev
Tommy Aagaard Christensen

Supervisor:
Bent Thomsen

Abstract:

In methods for programming language design evaluation there is a gap between small internal methods and large scale surveys and studies. A similar gap in HCI has been filled by the discount usability evaluation method. In this report, the discount usability methods applicability on programming languages was examined, and it was found usable but better suited for compiler and IDE evaluation over language design evaluation. To create a method to fill the gap, a modified version of the usability method, where the IDE and compiler was removed, was tested.

Pages: 92

Appendices: 6

Copies: 0

Finished: 13 June, 2016

Preface

The following report was written by Svetomir Kurtev and Tommy Aagaard Christensen in accordance with the conclusion of the tenth and final semester of the Computer Science Master Program at Aalborg University.

We would like to thank Bent Thomsen for the help and guidance he provided us with throughout the development of the project.

Contents

Chapter 1

Introduction

Computer programming has increasing relevance to today's advancement of technologies. Therefore, existing and established programming languages are constantly improved and new ones are created to meet that demand. The languages which are considered most suitable for introductory programming, are being adopted by educational institutions as part of their computer science curriculum e.g. Java, Python and more recently, Scratch [1]. Similarly, some languages are considered arguably better than others in their intended purpose in the software industry. However, formal evaluation methods for assessing programming languages are very few and limited in their use and most evidence gathered to support such claims are anecdotal in nature [2].

In recent years, however, the scientific community has tried to fix this. In particular, the focus of the PLATEU conferences is the scientific evaluation of languages. The basic observation is that language use and preference is highly dependant on the person, which indicates that there is no method that can tell the quality of the language based solely on the language. This has lead to user-based evaluation being the norm for programming languages. Commonly, the scientific community has taken to use methods from social sciences, which usually requires a studying a large number of subjects [3] [4] [5] [6]. The underlying idea is that the more people used, the less likely the result would suffer from bias, and therefore the study will be more scientific.

However, his method has the problem of being expensive. It is difficult to gather a large enough group of people to create a truly quantitative test of a language, especially if the test is based on observation rather than questionnaires. Typically this means a language designer cannot do these tests before the language is already finished and has gained widespread use. This leads to language designers either still omitting any evaluation of their language, or using more qualitative and lightweight approaches to evaluate their language.

In the field of Human-Computer Interaction (HCI) there has been a similar problem with usability evaluation methods [7]. Traditional usability studies often require many subjects and studying the use of the program over a long period of time. This made them prohibitively costly for companies both in resources spent and time-to-market, which prompted the development of a discount usability evaluation

method [8]. The discount usability evaluation method is a method designed to be a lightweight, qualitative evaluation method for the usability of a product. We are considering its viability as an evaluation method for programming languages.

1.1 Problem formulation

During our investigation, we concluded that there is gap between internal language analysis and the big and bulky industry programming evaluation methods [4] [5] [6], which leaves language designers without a flexible, low-cost and efficient solution to test on new programming languages. We want to have a method for evaluating programming languages, which is lightweight enough to be used by language designers, both for improving the language, and supporting claims about the language. The discount usability method is a method that is lightweight enough to be used by small groups of system designers and seems like it could be applicable in that regard. Ideally, we would use that as a starting stage for devising our own lightweight evaluation method which could be used in different stages of programming language design. This leads us to the following initial questions:

1.1.1 Initial questions

- Are the evaluation techniques from HCI applicable to programming languages evaluation?
  - What does the usability of programming languages say about programming language design?
- Can the discount usability method be used for programming language evaluation?
  - What are the potential shortcomings of the method for evaluating programming languages?
  - How can such shortcomings be addressed?

Part I

Problem Analysis

Chapter 2

Previous work

In our previous work [9], we explored how computational thinking becomes a necessary skill in everyday life and how introductory programming has been adopted by several countries (such as UK, USA, Denmark, Germany and others) as a part of their primary school curriculum. A big part of this process involves the choice of a suitable programming language and environment, and given the multitude of choices available, there has not been a well established way of selecting the right choice. Furthermore, there exist different approaches to programming (visual-based vs. text-based), each with their own strengths and weaknesses, which makes that choice even harder. In order to get a better understanding of how this choice could be made easier, we analysed three popular programming languages used in a educational setting - Java (BlueJ), Scratch (Scratch) and Racket (DrRacket), each supporting a different programming paradigm. The analysis was done mainly relying on a set of evaluation criteria, heavily inspired by Sebesta's evaluation criteria [10], on a set of programming tasks in the form of games. Although this approach gave us some pointers towards how effective and suitable each of tested languages are, the evaluation was rather subjective and did not possess the needed scientific rigour and empirical data appertaining for such a method. This however prompted us to look into how techniques from social sciences [11] could be used to gather the necessary data, in particular the discount usability evaluation method by A.Monk [8].

Chapter 3

Related Work

Programming languages have been used for many years, but there still has not been established a robust and efficient way to asses and evaluate them. However, plenty of research has been done on the topic and specific papers address that to a different degree as it will be shown in this chapter.

M. Farooq et al. 2014 [12] wrote a paper introducing an evaluation framework which provides a comparative analysis of widely used first programming languages (FPLs), or namely languages which are used as a first language for teaching introductory programming. The framework is based on technical and environmental features and it is used for testing the suitability of existing imperative and object oriented languages as appropriate FPLs. In order to support their framework, they have devised a customizable scoring function for computing a suitability score for a given language which helps quantify and rank languages based on the given criterion. Lastly, they evaluated the most widely used FPLs by computing their suitability scores. The novelty in their work stems from the definition of the evaluation parameters as well as the related characteristics for evaluating each parameter.

K. Parker et al. 2006 [13] note that the process of selecting a programming language for introductory courses is informal and it lacks structure and replicability. In order to address that, a more structured approach is proposed, which enables a more detailed evaluation of the selection process. The paper presents an exhaustive list of selection criteria, where each criterion is assigned weights in order to determine its relative importance in the selection process. This is tested by scoring different programming languages according to said criteria. The proposed approach is verified by an informal pilot study to assess the completeness of the criteria and gather enough feedback on the process itself. Given the dynamic nature of programming paradigms and languages, the authors acknowledge that the selection criteria and process can be revised.

Stefik & Siebert [14] have conducted an empirical study comparing how accurately novices in programming can write programs by the use of three different programming languages - Quorum, Perl and Randomo. Quorum is an evidence-based programming language, developed by them with the use of survey methods and usability tests, Perl is an already established language and Randomo, as the name suggests, is a language with a randomly generated syntax and whose keywords and symbols were picked

from the ASCII table, making it a Placebo language. The results from the empirical study showed that Quorum had significantly higher accuracy among the novice participants compared to Perl and Randomo, both of which had very similar accuracy rates. The authors contribute the higher percentage of Quorum to its very careful use of evidence while designing the language. However, they admit that the results might be skewed given the small test group, and repeatedly conducting the experiment might yield entirely different results.

The interest in visual programming languages has been steadily increasing for the past few decades as graphic support in both software and hardware becomes more and more prominent. J. Kiper et al. [15] 1997, present a method of evaluating visual programming languages by a set of evaluation criteria. The basis of their work is influenced by the taxonomy provided by Singh and Chignell [16] which divides visual computing in three key areas - *programming computers, end-user interaction with computers and visualization*. This set of five criteria - visual nature, functionality, ease of comprehension, paradigm support and scalability try to constitute the philosophy of a "complete" visual programming language. Each criteria on its own is supported by a set of metrics. This evaluation method not only allows assessment of an individual visual programming language but also the comparison of elements of a set of languages.

Using HCI to assist the design of programming languages is not a new idea. John F. Pane et al. [17] have used HCI principles in the design of their language called HANDS. They used research from Psychology of Programming(PoP), Empirical Studies of Programmers (ESP) and related fields, summarised in "Usability Issues in the Design of Novice Programming Systems" [18], to guide their language design decisions. Then they conducted a user study on HANDS and a version of HANDS with three features omitted (queries, aggregate operations, and the high visibility of data) to evaluate these three features. The study was conducted on 23 fifth-grade children where, of the children who made it through the tutorial, 9 were in the HANDS group and 9 were in the limited HANDS group. In this study the children in the HANDS group performed significantly better, solving a total of 19 tasks across the children in contrast with the limited HANDS group's one solved task.

Raymond P.L. Buse et al. [19] conducted a study on 3110 papers in software engineering research in order to analyse statistics on their usage of user studies. They found that, in general, the number of user studies in software engineering papers have been increasing in the last years. Also, for prestigious conferences (ones with a low paper acceptance rate) the percentage of papers with user evaluations was higher, although they did not find a correlation between user evaluation and citation count outside the papers with a high citation count. This is interesting for us since it shows an increasing interest in including user evaluation, which we are looking for a cheaper way of doing. The paper also found that a user evaluation with 1-5 participants performed equal to or better than using more participants, in terms average citation count. This is particularly interesting for us since it shows that using a discount evaluation method over a heavier one does not impact the citation count (and thus the perceived impact) of a paper.

Daniel P. Delorey et al. [20] conducted a large scale user study, in a different way, by doing statistical analysis on 9,999 open source projects hosted on SourceForge.net. The goal of their study was to

examine whether programming language has an effect on programmer productivity. They found that programming languages does indeed affect productivity, which is a great motivator to improve programming languages. However, they note that this study is insufficient to prove the correlation, partially because they use lines of code to measure programmer productivity, and this might not be an accurate metric. This kind of study does however allow one to gather and analyse a very large dataset to prove ones claims, but it is impossible to do before large repositories of projects in a language have been written This is unlikely to be true unless the language already has widespread use, so it is not a good way to measure performance of a new language.

Chapter 4

Existing evaluation methods

Programming language designers and researchers have been involved in the process of evaluating programming languages and their constructs since the very beginning, dating back to 1950s and 1960s, with the inclusion of languages such as Autocode, COBOL and Lisp [21]. The process of coming up with a programming language on the concept level is rarely under scientific rigour, and it usually involves whatever means the designers have at their disposal. As for the actual implementation of a language, being a constitute of its individual parts (such as lexer, parser, virtual machine etc.), there are certain guidelines and established methods one can follow, which does not usually leave a lot of room for deviation. However, things are different when programming languages are being evaluated since the community has different opinions of how this is done. Different programming languages have different design decisions and intended purposes, while each adheres to one or more programming paradigms. This, in turn, means that the same criterion might have a varying weight if put as the basis of an evaluation of two distinct languages. Researchers have partially agreed on a list of evaluation criteria and several means of validation. R.W. Sebesta [10] classifies the most important ones as *Readibility, Writeability, Reliability* and *Cost*, which are in turn influenced by different characteristics (e.g. Simplicity, Data types, Syntax design etc.)

This chapter will describe some of these means of validation, and how they can be applicable to our work.

4.1 Performance benchmarks

Parallel programming languages are mainly used in real life applications where speed and efficiency are key factors. In order to evaluate the effectiveness of such languages, performance benchmarks have been developed. S. Nanz et al. [22] describe an experiment where four different parallel programming languages (Chapel, Cilk, Go and Threading Building Blocks (TBB)) are used to implement sequential and parallel versions of six performance benchmarks, taking into account factors such as source code size, coding time, execution time and speedup. The benchmarks made use of a partial set of the Cowichan

problems [23] since "*reusing a tried and tested set has the benefit that estimates for the implementation complexity exist and that problem selection bias can be avoided by the experimenter*." as stated by Nanz et al. [22]. Additionally, H.-W. Loidl et al. [24] present in their work a detailed performance measurements on three fundamentally different parallel functional languages: PMLS, a system for implicitly parallel execution of ML programs; GPH, a mainly implicit parallel extension of Haskell; and Eden, a more explicit parallel extension of Haskell designed for both distributed and parallel execution. All three systems are tested on a widely available parallel architecture - a Beowulf cluster of low-cost workstations. The benchmarks made use of three algorithms: a matrix multiplication algorithm, an exact linear system solver and a ray-tracer.

Since we do not consider performance as a criteria we wish to evaluate on, we do not consider such benchmarks applicable to our work.

4.2 Usability Frameworks

Treating programming languages as if they were user interfaces is not something new, and addressing their usability has been one of the topics studied in the field of Human-Computer Interaction (HCI) for quite some time. "The Psychology of Computer Programming" by Gerald Weinberg [25] is one of the earliest works on the psychological aspect of computer programming, which dates back to 1971. Thomas Green and M. Petre [26] came up with the Cognitive Dimensions framework, under which, programming languages are considered as interactive systems, and are evaluated based on a set of dimensions such as *closeness of mapping, consistency, hidden dependencies, viscosity* (which is resistance to local change) and others. The framework has been tested on Prolog by Allan Blackwell [27], some visual programming languages [28] [29] [26] and an early version of C# by Steven Clarke [30]. Brad Myers et al. [31] have worked on the evaluation of programming systems and identifying common usability and design issues of novice programming systems. The goal of this was to, get a better understanding of how language designers can make use of HCI methods in their work. This culminated in a 10-year project called *Natural programming*, which has the goal of bringing programming "closer to the way people think." This resulted in new programming languages and environments [32].

More often than not, modern implementations of programming languages are closely tied to a specific IDE (Integrated Development Environment) which makes it very hard to treat the language as a disjoint entity. When trying to evaluate a programming language, is it fair to evaluate the language itself or the language and the IDE? Farooq and Zirkler [33] have proposed an evaluation method for application programming interfaces (APIs), which relies on peer reviews and is mostly centered around the APIs rather than the programming languages. Leonel Diaz [34] had a different approach to the problem by identifying two different dimensions of evaluation for programming languages and their environments: The IDE-to-Program user interface and the Language-to-Machine user interface. The first one treats the IDE as a user interface to the program, not the language, which in turn allows elaboration on how the interface can be evaluated. In the second dimension, the language is seen as the user interface to the hardware resources of the machine.

Usability frameworks fall into the discipline of HCI, therefore we consider this an area of interest for our work.

4.3 Case & User studies

Other approaches make use of case studies and user studies which might range from being informal to being controlled empirical experiments. Such controlled experiments [35] [22] [36] [37] ask group(s) of human participants to solve programming tasks with varying difficulty in different languages in controlled environments. The results from such controlled experiments provide valuable data on how programming languages could be evaluated, as well as what impact certain features of a programming language, like syntax and typing, might have. One drawback to that approach is that the variety and number of tasks solved could be limiting. This is in contrast to real-world programming where code bases grow constantly and change over many development iterations. For this reason, empirical studies have been conducted on code repositories (e.g. Github [38]) analysing large quantities of code, written by many experienced developers over long periods of time. This provides valuable feedback on defect proneness and code evolution, but it is not as focused as programming assignments since there is a great disparity between projects and the categories they encompass. Nanz [39] conducted a comparative study of 8 programming languages in Rosetta Code, each representing a major programming paradigm, serving as a middle ground between small programming assignments and big software projects. A total of 7087 solutions have been analysed as a part of a quantitative statistical analysis corresponding to 745 different tasks, taking into account programming features such as conciseness, running time, memory usage and error proneness.

Case and user studies involve controlled empirical experiments. Therefore we consider them an area of interest for our project.

4.4 Quantitative experiments

Another approach of conducting experiments in programming language research and evaluation is to rely on quantitative data rather than qualitative data. In such experiments a hypothesis is either proved or disproved, by executing a statistical analysis on the results, usually filtering out external factors in the process. The scope in such quantitative experiments could vary in the number of participants taking part - ranging from tens (comparative experiment on Quorum, Perl and Randomo by Stefik & Siebert [14]) to tens of thousands (quantitative study on a very large data set from GitHub by Ray & Posnett [40]). Additionally, the nature of the experiment could warrant to use of two different groups, tested under different conditions, where the results are analysed for significant differences between the two groups. Examples of such could be the completion of a set of programming tasks, each group using a different programming language to solve them or simply having the same exact programming setup for both groups, but having different prior knowledge. Stefik and Hanenberg have conducted few such empirical

experiments [41] [42] researching the impact of static and dynamic type systems and programming language syntax. Also, Stefik and Hanenberg address something called "the programming language wars" evident in the programming language community [43]. They elaborate on language *divergence* (too many different programming languages) and what the *impact* of each programming language on the community is. A common belief is that one programming language could solve the problem by being better than all the rest. The authors strongly disagree with this belief since problem domains and the people involved vary, which makes the creation of a single perfect language a myth. They believe that language designers duplicate effort by repeatedly trying designs which have already been tried by others, since design decisions are rarely backed by scientific evidence. They stress the importance of designing languages based on evidence and teaching empirical methods to programming language students.

Given our limited access to participants and our goal of a method using few participants, we will not consider quantitative experiments in our work.

4.5 Language-to-language comparisons

Another evaluation approach, which differs to the ones described earlier, is the direct comparison between a set of programming languages. Although it does not have the same degree of scientific vigour compared to the other approaches, it can still pretty accurately identify the strong and the weak points which constitute a particular language. Plenty of research have been conducted on that topic, dating back to the early eighties and involving a plead of languages (such as C, C++, Pascal, Fortran, Ada 95, C# and Java). Feuer and Gehani [44], made a comparison in 1982 of C and Pascal and argued what would be a good basis for such comparison. This involved the design philosophies of the two languages, followed by a simple program, written in each, and comparison of the features present in the two languages. Ultimately, they wanted to test the suitability of each of the languages in different problem domains. Pascal was also included in a study along Ada 95, by Murtagh and Hamilton [45], for deciding on language for an introductory computer science class. The selection involved testing particular language features against course objectives, resulting in Ada 95 being selected as the appropriate successor to Pascal. An earlier version of Ada was tested against C++ by Tang [46] in order to show contrasts and commonalities of the two languages. In later stages, direct comparisons have been made of Java against C and Fortran [47], as well as more notably C# and Java [48].

4.6 Discount usability evaluation method

When we refer to the discount evaluation method, we mean the cooperative evaluation created by Andrew Monk et al., 1993 [8]. Table 4.1 from Designing Interactive Systems [49] outlines the steps from the method.

The method is focused on finding the more impactful usability problems over finding a lot of them, and has thus proven to be both lightweight and useful. In particular, the Nielsen Norman Group [50] shows

Step	*Notes*
1 Using the scenarios prepared earlier, write a draft list of tasks.	Tasks must be realistic, do-able with the software, and explore the system thoroughly.
2 Try out the tasks and estimate how long they will take a participant to complete.	Allow 50 per cent longer than the total task time for each test session.
3 Prepare a task sheet for the participants.	Be specific and explain the tasks so that anyone can understand.
4 Get ready for the test session.	Have the prototype ready in a suitable environment with a list of prompt questions, notebook and pens ready. A video or audio recorder would be very useful here.
5 Tell the participants that it is the system that is under test, not them; explain the and introduce the tasks.	Participants should work individually – you will not be able to monitor more than one participant at once. Start recording if equipment is available.

(a) table part 1

Step	*Notes*
6 Participants start the tasks. Have them give you running commentary on what they are doing, why they are doing it and difficulties or uncertainties they encounter.	Take notes of where participants find problems or do something unexpected, and their comments. Do this even if you are recording the session. You may need to help if participants are stuck or have them move to the next task.
7 Encourage participants to keep talking.	Some useful prompt questions are provided below.
8 When the participants have finished, interview them briefly about the usability of the prototype and the session itself. Thank them.	Some useful questions are provided below. If you have a large number of participants, a simple questionnaire may be helpful.
9 Write up your notes as soon as possible and incorporate into a usability report.	

Sample questions *during* the evaluation:

- What do you want to do?
- What were you expecting to happen?
- What is the system telling you?
- Why has the system done that?
- What are you doing now?

Sample questions *after* the session:

- What was the best/worst thing about the prototype?
- What most needs changing?
- How easy were the tasks?
- How realistic were the tasks?
- Did giving a commentary distract you?

(b) table part 2

Figure 4.1: combined caption

that you only need about 5 test subjects to get clear feedback on your system. Jakob Nielsen argues that instead of expending huge budget and time, the best results could be achieved by conducting multiple smaller tests. In their experiments across a large number of projects, one test subject managed to find roughly 31% of the the existing usability problems. Every subsequent user contributes to the curve by identifying new problems, but that also includes problems already found by the previous user. Technically, after a certain point, adding additional users contribute less and less to the overall identification of problems, having essentially diminishing returns. The curve presented by the author shows that 15 test users are needed to identify all the usability problems. However, since the goal usually is to improve the design rather than document its weaknesses, they recommend conducting 3 smaller tests with 5 users each rather than one experiment with all 15 users.

4.7 Instant Data Analysis

After conducting a usability study, the data needs to be analysed and formulated into a list of usability problems. There are multiple ways of doing this, but the one we will be highlighting is the Instant Data Analysis (IDA) [7] method. The IDA method works by identifying a list of problems, that participants encountered during the test, from the observers' memories and observations. This is opposed to the Video Data Analysis (VDA) method, where the videos of the tests are rigorously examined to identify the usability problems. The problems are then sorted into three categories: cosmetic, serious and critical.

Cosmetic problems briefly slowed down the participant but were quickly overcome.

Serious problems stumped the participant for longer time, but were eventually overcome.

Critical problems the participant could not overcome without assistance from the facilitator.

This prioritisation then usually becomes a great assistance in determining which problems are the most important to fix.

The IDA method is a good analysis method for our purposes because it is lightweight. Compared to the VDA method, the IDA method saves a lot of time avoiding the rigorous analysis of the video footage. As shown in the paper [7], they spent 4 man-hours with the IDA method as opposed to 40 man-hours with the VDA method. Furthermore, the time to result is greatly reduced since the method uses these man-hours in a joint discussion, causing the real time spent to be around 1-2 hours. As for the results, in the IDA paper [7] the IDA method found 85% of the same critical problems as the VDA method and found roughly the same amount of problems total (41 for IDA versus 46 for VDA). This makes the IDA method a great method for our purposes since the potential loss in problem accuracy is significantly lower than the time saved.

Part II

Usability Evaluation Experiment

Chapter 5

Challenges

Conducting empirical studies in order to prove the validity of a hypothesis usually involves a certain amount of human resources. For studies and experiments with a qualitative nature, this number is in the tens or hundreds, and for ones with a quantitative nature - in the thousands. Gathering data, even from a small number of people, is a daunting task and takes considerable resources and time. Additionally, the problem arises of trying to have a participants sample with diverse backgrounds (age, occupation, experience etc.) in order to address a wider population, which adds an additional layer of complexity.

Although the qualitative nature of our experiment does not require a large number of participants to show viable results, we still had difficulties in gathering the necessary number of participants. Some reasons for this might be that people do not find a good enough incentive to take part in such experiment, or they simply find the process intimidating, time consuming or not important. We found that approaching people directly rather than through electronic means (email, forums, on-line conversations) yielded a higher chance of them wanting to take part as participants.

Chapter 6

Experiment Setup

In order to get an understanding of how applicable the discount usability method is for evaluating programming languages, we decided to use it on two distinct programming languages. These two languages represent two different paradigms, where C# is relevant for object-orientation and F# for functional programming. The Instant Data Analysis (IDA) method [7] was used for evaluating the data.

The usability evaluation was conducted in the usability lab of Aalborg University [51], in test room 1. The Experiment entailed the participant being positioned at a desk with a mouse, keyboard and a screen. On the screen, Visual Studio Community 2015 was open with a project in the current language being tested. The participant was then given an hour to try and solve specific task(s) in the language, while under the supervision of a facilitator sitting next to him/her. The facilitator's job was primarily to keep the participant talking about what they where thinking, and secondarily to assist if the participant got stuck or encountered issues with the test. Meanwhile, an observer would take notes from an adjacent observation room. The control room was used for this. A camera would record the particpant's interaction with the keyboard and mouse, while the screen was recorded using an output splitter, and the sounds was recorded using a microphone.

6.1 C# Tasks

Designing the task for C# had several topics in consideration such as object-orientation, use of class hierarchy, working with extended methods and inherited properties, which were all considered as very important in representing the philosophy of the language.

The task was designed for subjects who already had experience with C# and know about object-oriented programming. The wording was intentionally left vague by design and specifically avoided directly mentioning programming constructs, in order to avoid biasing the solution in a particular direction, as well as more accurately show how the language was used. The task itself entailed the following:

Task Sheet

Create a system for managing stats and interactions between RPG characters with different classes. Each character has:

- a name
- a character class
- a stat that represent their hit points
- one or more stats that represents offensive prowess
- one or more stats that represents defensive prowess
- optional class specific stats
- the ability to attack another character, which reduces their hit points by value related to the difference in character1's offense and character2's defense
- optional the ability to perform class specific skills

Here are some examples of characters:

Name	George	Bob	John
Class	None	Mage	Medic
HP	200	100	200
Offense	50	60	20
Defense	30	10	60
Other stats		mana = 100	
Abilities	Attack	Attack Fireball = more damage but costs mana	Attack Mending = increases HP of target character

Create the system for these RPG characters with some character classes, some characters in this system and some code showing some interactions.

Now we discover that we also want a way to refresh characters to their original hit points, as well as preventing healing beyond this point. To do this we need to also store a maximum hit points value for each character as well as a method for restoring a characters resources. Some class specific resources might also need this to allow them to be replenished as well.

6.2 C# Results

In our experiments on C# we got two subjects. This is less than the desired minimum of five test subjects for a usability test. However, this is not considered a significant problem since the focus of the experiments is more on the method itself rather than on the results of it. The results of the experiment are evaluated using the IDA method, and therefore focused on a discussion based on the observations made. The results of the discussion could be seen on Table 6.1.

Most of these issues were only experienced by one of the subjects, and usually reflected their experience. The 4th semester student was the one experiencing the serious issue due to not having had much need for inheritance in any of his previous projects. On contrast, he did not have any difficulties remembering keywords due to recent use of the language. This is in contrast with the 10th semester student, who had more experience using inheritance but had not used C# as recently and therefore had difficulties regarding specific keywords. Neither of them invoked the auto generator for getters and setters and instead chose to make the variables public instead. They did, however, both express knowledge that this was not ideal and that it was done due to laziness and the small scope of the project.

Critical	Serious	Cosmetic
	Inheritance: the subject experienced difficulties identifying when inheritance could be useful	Could not invoke the auto generator for getters and setters
		Difficulties remembering the keyword for declaring a method override-able (Virtual)
		Troubles remembering the special syntax for invoking the base constructor

Table 6.1: The results of the C# experiment

6.3 F# Tasks

Similarly to how the C# experiment was conducted, the F# one involved a specific task addressing some key concepts from functional programming such as recursion, working with immutable data and functions. The task itself involves finding a path from a starting position to a goal position on a 2-dimensional tile-based world. The full task description as follows is:

Task Sheet

Create a path-finding function for a robot from an initial position to a goal position.

The world is a 2D world represented by coordinates with a defined maximum size (e.g. 20, 20) The position of the goal and the robot are represented by coordinates that fit in the world.

Given that the robot and goal positions are known, write a function which returns a path from the robot position to the goal position. The path could be represented as a list of moves of the type *"up"*, *"down", "left" or "right"* each representing the robot moving one in that direction

Write some code to set up a scenario to test your function.

Now write a path-finding function where the robot does not have access to the goal position but only to a function that says whether the current position is the goal or not.

6.4 F# Results

The F# task was completed by only one test subject which is not nearly enough for the minimum of five test subjects in order to conduct a viable usability test. Although the results are insufficient, they might raise some interesting points about the usage of the language and what concepts people find difficult to work with. The results were again evaluated by using the IDA method and they can be seen on Table 6.2

Critical	Serious	Cosmetic
The subject had difficulties with implementing tail recursion		
The subject had difficulties with pattern matching		
Some basic syntax did not allow the subject to continue with the experiment without the intervention from the facilitator		

Table 6.2: The results of the F# experiment

The participant was familiar with F#, but did not have a lot of experience working with the language. Furthermore the participant had reported having difficulties thinking in the functional programming patterns. The problems encountered reflect this inexperience. All of the difficulties were encountered early in the test and were mostly about remembering how the language worked. These are considered critical because the facilitator stepped in to help the participant. However, after the initial difficulties the participant started to grasp the language, and solved the later tasks on his own. This showed us that when trying to program in a language one has little or no experience in, knowing the syntax of the language is a major hurdle.

Chapter 7

Discussion

Although we were light on participants for the usability test, doing the test still gave us some insight into evaluating language design. In particular we discovered that the Integrated Development Environment (IDE) had a great influence on the usability of the language. It would often greatly assist in reminding the user of various keywords and constructs, like reminding a participant of the `virtual` keyword being the keyword for allowing method overloading, which means a lot of potential problems was fixed by it. This matches some observations made by Faldborg and Nielsen [52] and Pedersen and Faldborg [53], where they observed the participants' difficulties in differentiating the IDE from the language. There has been support for the discount usability method being good for testing the usability of the full package of the language and IDE like has been done by the developers of Pocket Code [54]. However, these observations lead us to believe that the method is less suited for evaluating language design. Another discovery was the difficulty of coding in a language one is not strongly familiar with even with the help of the IDE. The IDE only helps if it can guess what you are trying to do, which would require the user to already have an idea of how the code should look.

Part III

Evaluation Method

Chapter 8

Introduction

For our method we want to focus on evaluating the language design of a programming language. Following our previous observations on the IDE's impact on the result, the next step would be to remove the IDE from the test. Since the idea is to focus on how well the language fits with how one want to code regardless of any tools, we also want to avoid using a compiler or other specialized tools for the language. What this means for the method is that the test can be done in any text-editor or indeed on physical paper. This has the added bonus of it being possible to do the test before a compiler has been created for the language.

Another thing our previous test showed was the high difficulty of suddenly programming in a language where the rules are not well known to the user. Since a large portion of the intended use of this method is on new programming languages, this is also an important concern for the method. Our idea for addressing this is to, along with the task sheet, add a sample sheet, which would contain examples of code written in the language with an explanation of the functionality of the code.

To test our method, we used it on a language that was unlikely for our participants to be familiar with. The language we used was the language Quorum [55].

Chapter 9

Experiment Design

In this chapter the design considerations and decisions will be described, including tasks, samples and interview questions.

9.1 Tasks

The concept of programming is essentially devising a mental plan for solving a particular problem, and transforming that into a workable solution by making use of some programming language. The solution to a problem might vary for different implementations since different paradigms have different approaches, but usually the essence of said solution does not change. Devising a set of programming tasks which are not targeted at a specific language implementation is a daunting task, since many paradigms have to be taken into account at the same time. Additionally, the tasks have to be interesting and engaging enough so that the participants can focus on the solution. As described previously in section 6.1, for our usability experiment we devised one big task with several subtasks for the C# part, each as a prerequisite for the next one. It followed the theme of a role playing game and it mainly revolved around the use of inheritance. The second task, addressing F# in section 6.3, was still modelled as a game and it revolved around a robot finding a goal in a world of tiles, essentially relying on the use of recursion as a concept. Although we did not count time as an important factor for both our experiments, we still had to put a specific time constraints on how long a participant can work on the tasks. The main reason for that is our aim is process-oriented rather than result-oriented, which make it easier for programming language designers in the future to replicate our experiments.
Based on the feedback from the participants, we got a better estimate of how much time each task takes for its completion and how relevant it is in the given context. Furthermore, this served as a stepping stone towards gaining a better understanding of how to design tasks with more general context i.e. being solvable in more than one programming paradigm. The design behind our task sheet was mainly influenced by the use of Quorum as our experiment language, but that could easily be used by language designers as a template for devising their own tasks, tailored towards a specific language of choice. Additionally, the scope of each of the tasks can be used as a guideline, giving a more controllable and predictable time

frame (all of the tasks together amount for approximately 1 hour).

For our evaluation method, we devised several smaller tasks, each addressing different features and constructs of the selected language - in this case Quorum. We drew heavy inspiration for some of the tasks from Codekata [56] 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 design of this task was inspired by the task used in the C# section of our usability experiment in section 6.1.
- The final task was testing operations on strings, including the exercise of in-build actions specifically useful for splitting text segments.

The final task sheet can be found in Appendix B.

9.2 Samples

The idea behind the sample sheet was 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. The samples often included smaller details, not specifically relevant for what was explained at the time in order to cut down on the number of samples needed. An example of this is the `Paws()` method in the classes sample being used to demonstrate returning values from methods. The constructors in Quorum were omitted from the samples as they are optional and their functionality does not support using parameters which is what most of our participants would consider the purpose of constructors. The sample sheet can be found in Appendix C.

9.3 Interview

In the context of qualitative research, the interview is considered the most widely employed method. There are two main types of interviews associated with qualitative research - the *structured* interview and the *semi-structured* interview [11]. Qualitative interviewing is generally very different from interviewing in quantitative research in the methods being employed. The qualitative approach is much

less structured, focusing on formulation of research ideas rather than maximizing the validity of measurement of important concepts. Additionally, the discussion is tailored towards the interviewee's point of view more than than that of the researcher's and deviation in the responses is actually encouraged since it provides a degree of flexibility and thus, the possibility of rich and detailed answers. As already mentioned, the qualitative interview process can have two approaches. In the *unstructured* approach the interviewer uses just a simple aide as prompts to where the conversation should lead. The interviewee is allowed to respond in a free manner and there could be a follow-up on interesting points. On the other hand, the *semi-structured* approach warrants the interviewer to have a list of questions, addressing specific topics, working as an *interview guide*, but still leaving a certain degree of leeway in how the interviewee responds. Still, it is not mandatory to follow the guide as it is and questions do not have to have a specific order in being asked.

Initially, for our usability evaluation experiment we made use of the first approach where rather than specific questions, we had an open discussion with the participants. The discussion started with addressing some general areas of interest (task completion, inheritance, information hiding etc.) and continued from there based on the responses from the participants and the direction they headed in. We had the intention to follow up with the same approach for the experiment involving our evaluation method as well, but based on the feedback from the pilot test, we decided to try a more structured approach. Therefore, as a part of the interview, we created a small questionnaire with several 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. Appendix D lists all the questions while the answers from the participants in the form of notes could be found in Appendix E.

Chapter 10

Experiment Setup

Quorum is an evidence-based language for novices that uses results from experiments to decide on language design decisions. Since our participants were computer science and software students who are all experienced programmers, we expected there would be some errors from this targeting mismatch for us to analyse. As we are still focusing on the problems encountered during the process rather than the resulting code, we decided to record the experiments. To make recording easier we decided to use a text-editor on a computer. The text-editor we used was Notepad++ [57]. 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. Since we wanted to allow for a more flexible schedule for participants, to increase participant attendance, we decided not to use the usability lab. Instead, the tests were conducted using the laptop in the canteen of Cassiopeia. 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.

The process had two parts: For the first part the participant was given a task sheet and a sample sheet, and asked write programs that could complete the tasks on the task sheet. The participant was told not to worry too much about the code being written correctly since it would not be compiled, but to still try to get as close as they could. During the test the participant could freely ask the facilitator for assistance, though the facilitator preferred to answer by referencing the sample sheet if possible. The facilitator would also occasionally act as a surrogate compiler by telling the subject about some identified errors.The first part would take about an hour. For the second part the subject would be interviewed about the language and the test. There was an interview sheet with some questions to be answered during the interview, but other than that the interviews were more informal talks. During this part the subject would usually be made aware of most of the otherwise unmentioned errors, to be able to provide a more informed discussion.

10.1 Participants sample

All the participants taking part in the usability evaluation and the evaluation method were experienced programmers. They all had very similar age and occupation - Computer science students from the 4th semester and up, with programming experience in C and C# as well as other programming languages (e.g. Java, F#, Python, Pascal).

We had six participants taking part in the main experiment, who are numbered participant #3 to participant #8. Participant #1 was the pilot test participant we used to improve our setup before the first test. Participant #2 has been omitted from all the direct evaluation results as he was not an experienced programmer and therefore does not fit the target group of the experiment. We have kept his data, however, as it shows some interesting pointers for problems encountered by novice programmers.

Chapter 11

Experiment Experiences

In this chapter the observations and experiences made during the process of conducting the experiments will be described.

11.1 Tasks

Apart from the pilot test, we found the tasks to strike a good balance between difficulty and relative time it takes to solve them. Although we did not consider time as a sensitive factor for the experiment, we still wanted to stay in a certain time frame amounting to around one hour. All of the participants, except one, managed to finish the tasks around this time frame. Additionally, they found the tasks challenging enough and good at conveying their intended purpose - using specific constructs from the target programming language.

11.2 Sample Sheet

When first reviewing the sample sheet, most participants on skimmed the code samples instead of reading it thoroughly. The sample sheet would then be used as reference, which the participant would look in when they were in doubt about something. The participant could easily spend a considerable amount of time, shuffling through the three sample sheets looking for the code example that demonstrated what they were looking for. This was often alleviated by the participant simply asking the facilitator their questions instead, at which point the facilitator could point out where the code would be in the sample sheet. Occasionally the specific question would not be in the sample sheet, in which case the facilitator would usually answer depending on their knowledge of Quorum. Usually these questions were about features not present in Quorum.

11.3 Interview Questions

Initially, the interview questions did not even have a written form. This changed after a few iterations since we thought that writing the questions down rather than just verbal pointers will give us a more structured approach and it was easier to keep track off during the interview process. For this reason, rather than focusing on specific points, we tried to keep their number to a minimum and encapsulate a given interview direction with each, which the participants can share their thoughts on freely. Although they had a certain degree of freedom in their answers, common similarities and points were still addressed.

Chapter 12

Results

This Chapter will highlight the results from the evaluation method. Initially, we conducted a pilot test with one participant in order to test the overall setup of the experiment and get some pointers of what areas could be further improved. Similarly to how the results were evaluated in Chapter 6, we made use of the IDA method [7] to categorize the problems we identified based on their importance and severity. Furthermore, we compared our results with the exiting findings about the evaluation of Quorum in order to prove the validity of our method.

12.1 Pilot Test

The pilot test was conducted on a single participant which served the purpose of giving us some feedback on how we can improve the test setup. One of the bigger things was that the coding part of the test took two hours rather than the one hour we had intended despite skipping one of the tasks. Following the pilot test we made several changes to our setup:

- Task 1 in the task sheet was largely rewritten to have more concrete examples and to try and focus on the simpler calculations. This was due to our participant spending a lot of time working on constructs for a larger shopping system, instead of focusing on the arithmetic core of the task we had intended.
- Task 2 was shortened from 5 subtasks to three subtasks due to the repetitive nature of the subtasks and to cut down on time spent.
- The participant skipped task 3 due to its similarity with task 2. This caused us to swap the position of task 3 with task 4. This was also helped by the observation that getting data about classes in Quorum was deemed more important than operations on strings.
- In the discussion the access modifiers for methods and properties in classes was brought up which caused us to realise this was not discussed in the sample sheet. Some description and examples of access modifiers for classes was then added to the end of the sample sheet.

- When conducting the interview, we realised that the interview lacked direction. This caused us to create the interview questions to provide a guideline for things to discuss.

12.2 Problems categorisation

The premise of the experiment is to try to divide the IDE from the language which does not warrant using a compiler. However, since this means the system cannot give any feedback, The participant would not spend time on problems, which means the usual rules for categorisation, used in the IDA method, can not apply. For this reason, we would try to reason where each problem should be, and would it make a difference if a compiler had been used instead. The general guidelines we have used for this categorisation are:

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.

Critical problems

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.

Serious problems

Critical	Serious	Cosmetic
Not using the `end` keyword at all	Not using the `end` keyword to end the scope of if-statements	Using colon (`:`) instead of dot (`.`)
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)	Using conditional AND and OR as `&&` and `||` instead of `and` and `or` keywords, as defined in Quorum
	Forgetting to import a library for containers (array)	Using the `float` instead of `number` keyword
	Not using `elseif` to avoid having to close an additional scope	Using `string` instead of the `text` keyword
	Not resetting inner loop iterator between loops	Writing `output` instead of `return` as the keyword for a return statement
		Using `==` in conditional statements instead of =
		Using `int` instead of `integer`
		Using `bool` instead of `boolean`
		Typos in library importing
		Mistyping `integer` as `integar`
		Accidentally used `0` instead of `O` in variable name
		Mistyped the `is` keyword as `ia`
		Forgot to add the `repeat` keyword

Table 12.1: The table of identified problems categorised by severity

1. **Not using the `end` keyword to end the scope of if-statements** - Although this problem looks similar to the first problem defined in Critical problems, the difference is that it is more likely to be an overlook than a misunderstanding of the scoping rules in Quorum. Also single-line if-statements might be present in other languages and not in Quorum.

2. **Forgetting to increment the iterator in a repeat while loop** - This could be considered an oversight on the participant's part, attributed to how the repeat-while construct works in Quorum, compared to how usually for-loops are used, and therefore it was not critical. However, it is still

considered a serious problem because of the impact it has on the structural correctness of the code.

3. **The lack of common looping constructs (for-loops or foreach loops)** - This is considered a serious problem for few reasons. Firstly, it warrants the use of the repeat-while construct as a part of Quorum, which might not be so intuitive for people coming with backgrounds in other languages, where these constructs are present. Secondly, this might compound to the previous problem described in this section which would have a high impact on the validity of the written program.

4. **Forgetting to import a library for containers (array)** - Containers in Quorum, and specifically arrays, have to imported first before being used. This is considered a serious problem since it might have a high impact on the validity of the program.

5. **Not using `elseif` to avoid having to close an additional scope** - This problem is serious because it shows a lack of understanding the finer points of scoping in an if-else chain. This is problematic since `else if` is also a valid syntax, but carries unintended consequences.

6. **Not resetting inner loop iterator between loops** - Similarly to serious problem 2, this could be considered an oversight on the participant's end due to previous experience with other programming languages.

Cosmetic problems

1. **Using colon (`:`) instead of dot (`.`)** - This is considered a cosmetic problem since does not affect the structure of the program being only an exchange of a single character. It could be said that most of the participants had a programming bias given their background in other programming languages where the dot notation is common.

2. **The lack of aggregate operators in Quorum (e.g.+=)** - This is considered a cosmetic problem since it does not affect the correctness of the program but it is rather a matter of convenience for the participants.

3. **Using conditional AND and OR as `&&` and `||` instead of `and` and `or` keywords, as defined in Quorum** - This problem is considered cosmetic because the participants did not use the correct keywords in the context but had the proper intentions to do so. This could be attributed to the simple matter of not properly reading the sample sheet to find the proper keywords and using the ones they know from other languages instead.

4. **Using the `float` instead of `number` keyword** - This is considered a cosmetic problem because it does not have a big impact on the program's correctness but rather is using a naming convention from other programming languages

5. **Using `string` instead of the `text` keyword** - this is the same as with the previous cosmetic problem

6. **Writing `output` instead of `return` as the keyword for a return statement** - This is cosmetic because it is mostly a result of our sample sheet using `output` often, while the participants were more commonly expected to write code returning something and had a familiarity with the `return` keyword from other languages.

7. **Using `==` in conditional statements instead of `=`** - Similarly to previous cosmetic problems, the main reason behind this problem is that most of the participants had experience with other programming languages where the == notation is common and in turn had a particular bias against using the = notation.

8. **Using `int` instead of `integer`** - same as with cosmetic problem 4

9. **Using `bool` instead of `boolean`** - same as with cosmetic problem 4

10. **Typos in library importing** - This is a simple case of having small typos when writing the import code. Easily fixed and a cosmetic problem.

11. **Mistyping `integer` as `integar`** - this is a cosmetic problem since it is a simple typing mistake and it does not have any impact on the validity of the program.

12. **Accidentally used `0` instead of `O` in variable name** - Again another small typo and therefore cosmetic.

13. **Mistyped the `is` keyword as `ia`** - similar to cosmetic problem 12

14. **Forgot to add the `repeat` keyword** - This problem is considered cosmetic since it does not have a significant impact on the correctness of the program.

12.3 Comparison with Quorum's evidence

In their empirical experiments, Stefik and Gellenbeck [58] gathered many statistically significant results regarding keyword choices. Given that they try to primarily address visually impaired people and novice programmers, selecting the most intuitive words seems like a logical choice. For one of their experiments [58], 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 from the IDA evaluation in section 12.2. Additionally, we would also relate two of the empirical studies from Stefik and Siebert [37] and their findings about keyword choices. This would help us to make a comparison between the authors findings and the results from our evaluation, essentially giving an additional degree of credibility to the method if similarities are found.

- In the results for the concepts of AND, OR and XOR, Stefik and Gellenbeck [58] 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 [58] 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 [37].
- 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 [59]) following a study which shows that `repeat` represents the concept of iteration significantly better than the the aforementioned words [58].

Based on the results from Stefik and Siebert [37], 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 [37] 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 look at 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 [37]. 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. This is in contrast to the papers results where their programmers did not find `for` among the most intuitive and found `foreach` among the least intuitive keywords for looping. 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 proven to be more user-friendly [60]. 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.

12.4 Interview Results

During the interview we collected feedback addressing various key points such as the usability of programming languages (in particular Quorum), programming without the help of a compiler or an IDE, impressions of how effective some constructs are and how they can be improved. The most common observations among the participants were:

The use of Quorum as a programming language - the majority of the participants (#3, #4, #5 and #8) found Quorum easy to use and understand. Additionally, some compared it and found it similar to other languages such as C, C#, Pascal and Python and generally less verbose than standard OO languages they had experience with (e.g. C#,Java).

Managing scoping rules by the use of the `end` keyword instead of brackets - Generally, most of the participants found the use of the `end` keyword for defining scopes very confusing. Participants #4 and #6 preferred the use of brackets, similar to OO languages they were familiar with (e.g. Java and C#), while participants #7 and #8 preferred indentation similar to languages like Python. Participant #7 further suggested to extend the *end* construct to *begin-end*, similar to Pascal, which he believes would make the language more user-friendly for novices.

Quorum uses the colon (`:`) notation instead of dot(`.`) - Given their prior experience with programming languages, where the dot (`.`) notation is common, participants #4, #6 and #7 found it confusing to use the colon (`:`) notation instead. This confusion was further reinforced by the fact that the dot notation is still used when importing libraries.

The lack of common control statements was confusing - The lack of common control statements (such as *for* or *for-each* loops) in Quorum seemed like a hurdle for the participants, and consequently they found it not so intuitive to use the *repeat while* construct as a substitute of that. This is evident by the fact that some of them completely forgot to include `repeat` in the loop's signature (participant #3) or found it unnecessary altogether (participant #7).

12.5 Interview Suggestions

During the interview we collected several suggestions as to what could be done to improve the experiment. These suggestions have been described here.

Providing a skeleton for the task solutions One of the suggestions was to add some code on the code sheet that would show the skeleton of the expected solution. The participant would then only have to worry about filling in the code for the functionality rather than how the solution should be structured.

Adding a cheat sheet to the sample sheet Another suggestion was to add an extra sheet to the sample sheet, that would contain just a list of all the keywords and constructs, to have a single page to look at when looking for a specific thing.

Using separate pages for each task A third suggestion was to have each task on a separate page, which would allow addition of some samples specific to that task on the page. In essence by having a smaller subset of the samples for each task, the amount of sample code to look through at any time would be reduced.

Chapter 13

Discussion

The results from Chapter 12 address some potential problems 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.

13.1 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 as mentioned in section 10.1. 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 refered to him rather than the sample sheet.

13.2 Tasks & Samples

It is difficult to create good tasks, as we have experienced in our experiments. Firstly, we had to determine which parts of the language we wanted to test. For our C# test it was object-oriented programming, for our F# test it was recursive functions and for our quorum test it was arithmetic operations, operations on containers, objects and classes and operations on strings. Then we tried to build a scenario that would give us tasks that exercise that decision. The specific scenario was not too important, but having one helps convey the task, and gives some ideas for examples and names, which eases some unintended creative burden on the participant. Next we had to decide how to formulate the tasks. Here we ran into the problem of how strictly one wants the process to be. For our first tests on C# and F# we intentionally kept the tasks vague to ensure we would not corrupt the data. However, this allowed the significant variance in task completion we had between our two participants. It is possible this could have been avoided if we had more strictly defined tasks. Although, that would also have caused us to lose the data we got about the intuitiveness of using inheritance. For our test on Quorum, we used more smaller tasks in several different scenarios, instead of one big scenario as in the previous tests. This meant we had to have a more strict formulation of the tasks. Otherwise, there was a higher risk of any tasks taking up all the time as it was interpreted to be larger or more complex than intended, which we discovered in our pilot test. However, even with a stricter formulation, we ran into participants overcomplicating tasks by interpreting them in a larger scope of the scenario than we had intended. At the same time, we also encountered some participants not being able to start on the first task as they simply did not know where

to start on a blank piece of paper. One way to make the formulation even stricter, without dictating what to write, could be to provide the skeleton of the solution we expect. This might also fix the issue with the blank page paralysis, as it gives the participant a starting point to work from. It does, of course, lose the data about how the participant would structure the solution, as we would have done so for them.

Similar problems was found when trying to define a sample sheet. Obviously, we wanted to at least have examples demonstrating the minimum of functionality we expected to be necessary to solve our tasks. However, only using the bare minimum limited the data we could gather, as the participant would be pushed into using one particular functionality where another might also have been used. An example of this could be that in our sample sheet we only included the *repeat while* loop, which means we would not be able to get any data about the *repeat times* or *repeat until* loops. To avoid this problem it would be an option to include examples of more functionality than what is strictly needed, to give the participants more freedom in choosing their constructs. Although, this would require a larger sample sheet, which would likely be another problem. Our participants would often only skim through the sample sheet, and some directly expressed that it felt cumbersome to shuffle through three pages to find the example they were looking for. Increasing the size of the sample sheet would likely only exacerbate this problem.

To fix this problem, one of our participants suggested adding a "cheat sheet" to the samples that would have all the functionality summarised on a single page. Creating such a "cheat sheet" would, however, present a challenge of its own, as it can be difficult to convey all the functionality on a single page. Also a lot of our participant expressed appreciation at the use of working samples in our sample sheet, since it gave them a more complete image of how the code should look. From this, we believe it is more important to focus on the bigger sheet, but if you can create a "cheat sheet" it could be a worthwhile addition.

Another thing to consider when creating a task sheet is the order of your examples relative to the order of the tasks. Some of our participants expected the samples in the sample sheet to follow the tasks in order of relevance. This occasionally gave a few errors, like for example the problem where a participant used `output` instead of `return`, was a result of `output` being demonstrated in the early examples while `return` was not demonstrated until the late examples. The participant had not seen the `return` example before doing the first task, and upon seeing the `output` keyword just assumed that was it. Being more mindful of ordering our examples could possibly have avoided this problem, but there might be a risk of it biasing the participants. Especially if one is going for demonstrating an excess of functionality, there is a risk that whichever alternative is shown first, would be the preferred alternative due to this bias. One last thing to note is that in our sample sheet one of the examples is a solution to one of the subtasks. While this could be an interesting way to test the participants attention to the sample sheet, it is probably best to avoid this in order to avoid biasing the solution to that task.

Part IV

Conclusion

Chapter 14

Conclusion

Initially, we identified in our problem formulation (section 1.1) that there is a gap in programming language evaluation between internal language analysis and the big industrial evaluation methods. In that regard, as well as a follow-up of our previous work (Chapter 2), we wanted to explore whether HCI techniques could be applicable for the evaluation of programming languages.

In Chapter 4, we briefly described existing evaluation methods for programming languages and how they differ, along with their strengths and weaknesses, which helped us get a better understanding of the design of such methods. In the context of HCI techniques, we have examined the usefulness of the discount usability method (Chapter 6) the IDA method (Section 4.7) for data analysis on C# and F#. In this examination we found that the discount usability method can be used to evaluate programming languages, but has some shortcomings. The programming language's 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 (Chapter 8). 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 is in a programming paradigm they were familiar with.

During the analysis process we had to alter the criteria used for categorising the problems from the IDA method - from time spent on a problem to severity of fixing the problem. This was 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 Quorum's evidence showed us that most of the data was comparable though not strictly in agreement. However, Quorum's 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.

Chapter 15

Method procedure

This chapter serves as a guideline of how the full method could used for conducting an experiment. In summary, the steps of applying the method to a language are as following:

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.
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 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 any 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 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 over-complicating tasks). The facilitator will confirm when a task is done, because the system won't 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.

Chapter 16

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 chapter 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 commonly are 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. It might however be literally impossible to create such a task set, as programming languages can be quite varied and some might not have any common ground. Specialised languages tend to omit a lot of features of a general-purpose language, and even within more general-purpose languages there can be huge differences (for example when switching programming paradigm) that would make a general task set impossible. A more likely way to use our method would be to compare languages for a specific application area, as it is possible to create tasks that are common in that area and apply the same task to several languages for comparison.

Ideally, conducting empirical experiments relies on the use of a very diverse test group - participants with

a different age, occupation and geological location in order to get a sufficient variance in the results. In our case, the participants were mostly in the same age group (20-25), had very similar occupation (4th to 8th Semester students in an engineering degree - CS, DAT, SW) and had a very similar geological location (North Jutland, within the Aalborg area). This certainly leaves a room for improvement, where conducting the experiment with a more diverse test group might yield different results.

Given the qualitative nature of the IDA method and the minimum number of participants needed for a viable experiment, we did not need a significant amount of participants for drawing adequate results. However, it could be interesting to see how the evaluation method would fare in a quantitative setup, involving hundreds or thousands of participants. We leave this as something to be considered in the future since testing on such as group at this point is beyond the scope of this project.

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. Although, it present a challenge for our method, as novices are less familiar with the act of programming and are more prone to feel lost. One example of this is participant #2 who we have otherwise omitted from the results since he was not an experienced programmer. In this test the participant was completely unable to write anything before the facilitator stepped in, and essentially dictated exactly what should be written for the first subtask. After this, the participant was, however, capable of solving the second and third subtask on his own. This shows that it is definitely not impossible to test on novices, but it probably requires some additional thought put into the task and sample sheet. Conducting the experiment on novices would give us a better idea of what kind of alterations the experiment setup would need to facilitate those experiments.

Based on the previous point, a good thing to do would be to design and conduct the experiment with a pre-made skeleton set up for the tasks, like we mentioned in Section 13.2. This would give us a much better idea about how that affects the experiment, and would allow us to better gauge how well it prevents the freezing issue it was designed to combat. If we were to conduct the experiment on novices, this would also be a likely addition to that test, as we believe they are the most likely to need the additional guidance.

Part V

Bibliography

Bibliography

[1] S. Davies, J. A. Palack-Wahl, and K. Anewalt, "A snapshot of current practices in teaching the introductory programming sequence," *SIGCSE '11 Proceedings of the 42nd ACM technical symposium on Computer science education*, pp. 625–630, 2011. 1

[2] S. Markstrum, "Staking claims: a history of programming language design claims and evidence: a positional work in progress," *PLATEAU '10 Evaluation and Usability of Programming Languages and Tools*, no. 7, 2010. 1

[3] L. A. Meyerovich and A. S. Rabkin, "Socio-plt: principles for programming language adoption," *Onward! 2012 Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, no. 4, pp. 39–54, 2012. 1

[4] R. Garlick and E. C. Cankaya, "Using alice in cs1: a quantitative experiment," *ITiCSE '10 Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*, no. 36, pp. 165–168, 2010. 1, 1.1

[5] D. Weintrop and U. Wilensky, "To block or not to block, that is the question: students' perceptions of blocks-based programming," *IDC '15 Proceedings of the 14th International Conference on Interaction Design and Children*, no. 21, pp. 199–208, 2015. 1, 1.1

[6] M. Armoni, O. Meerbaum-Salant, and M. Ben-Ari, "From scratch to "real" programming," *ACM Transactions on Computing Education (TOCE) TOCE Homepage archive Volume 14 Issue 4, February 2015*, no. 25, p. 15, 2015. 1, 1.1

[7] J. Kjeldskov, M. B. Skov, and J. Stage, "Instant data analysis: conducting usability evaluations in a day," *Proceedings of the third Nordic conference on Human-computer interaction*, no. 37, pp. 233–240, 2004. 1, 4.7, 6, 12

[8] A. Monk, L. Davenport, J. Haber, and P. Wright, *Improving your human-computer interface: A practical technique*. Prentice Hall, 1 ed., 1993. 1, 2, 4.6

[9] J. M. B. Christiansen, H. V. Geertsen, S. Kurtev, and T. A. Christensen, "Comparative analysis of educational programming languages and environments," p. 72, 2015. 2

[10] R. W. Sebesta, *Concepts of Programming Languages*. University of Colorado at Colorado Springs: Pearson, tenth ed., 2012. 2, 4

[11] A. Bryman, *Social Research Methods*. Oxford University Press, 4th ed., 2012. 2, 9.3

[12] M. S. Farooq, S. A. Khan, F. Ahmad, S. Islam, and A. Abid, "An evaluation framework and comparative analysis of the widely used first programming languages," February 24th 2014. 3

[13] K. R. Parker, J. T. Chao, T. A. Ottaway, and J. Chang, "A formal language selection process for introductory programming courses," *ICER 2006 Proceedings of the second international workshop on Computing education research*, no. 2, pp. 73–84, 2006. 3

[14] 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," *PLATEAU 2011 Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*, pp. 3–8, 2011. 3, 4.4

[15] J. D. Kiper, E. Howard, and R. C. Ames, "Criteria for evaluation of visual programming languages," *Journal of Visual Languages and Computing*, vol. 8, pp. 175–192, 1997. 3

[16] G. Singh and M. H. Chignell, *Components of the visual computer - a review of relevant technologies*. University of Tokyo, Tokyo, Japan: Springer-Verlag New York, 1992. 3

[17] J. F. Pane, B. A. Myers, and L. B. Miller, "Using hci techniques to design a more usable programming system," *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, 2002. 3

[18] J. F. Pane and B. A. Myers, "Usability issues in the design of novice programming systems," *Carnegie Mellon University, Pittsburgh, PA, School of Computer Science Technical Report CMU-CS-96-132*, 1996. 3

[19] R. P. L. Buse, C. Sadowski, and W. Weimer, "Benefits and barriers of user evaluation in software engineering research," *OOPSLA '11 Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pp. 643–656, 2011. 3

[20] D. P. Delorey, C. D. Knutson, and S. Chun, "Do programming languages affect productivity? a case study using data from open source projects," *Emerging Trends in FLOSS Research and Development, 2007. FLOSS '07. First International Workshop on*, p. 8, 2007. 3

[21] C. Hope, "What was the first computer programming language?." `http://www.computerhope.com/issues/ch001621.htm`, 2016. Used: 01/05/16. 4

[22] S. Nanz, S. West, K. S. da Silveira, and B. Meyer, "Benchmarking usability and performance of multicore languages," *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 183–192, 2013. 4.1, 4.3

[23] G. V. Wilson and R. B. Irvin, "Assessing and comparing the usability of parallel programming systems," *Technical Report CSRI-321*, p. 40, 1995. 4.1

[24] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. Michaelson, R. Pena, S. Priebe, A. Rebon, and P. Trinder, "Comparing parallel functional languages: Programming and performance," vol. Volume 16 Issue 3, pp. 203–251, 2003. 4.1

[25] G. M. Weinberg, *The Psychology of Computer Programming - Silver Anniversary Edition*. University of Colorado at Colorado Springs: Dorset House, silver anniversary ed., 1998. 4.2

[26] T. R. Green and M. Petre, "Usability analysis of visual programming environments A cognitive dimensions framework," *Journal of Visual Languages & Computing*, pp. 131–174, 1996. 4.2

[27] A. Blackwell, "Cognitive dimensions of tangible programming languages," *Proceedings of the First Joint Conference of EASE*, pp. 391–405, 2003. 4.2

[28] T. R. G. Green, "Instructions and descriptions some cognitive aspects of programming and similar activities," *Proceedings of Working Conference on Advanced Visual Interfaces (AVI 2000)*, pp. 21–28, 2000. 4.2

[29] T. R. G. A. Blackwell, "Cognitive dimensions of information artefacts a tutorial," 1998. 4.2

[30] S. Clarke, "Evaluating a new programming language," *PPIG 13*, pp. 275–289, 2001. 4.2

[31] B. A. Myers, A. Ko, S. Y. Park, J. Stylos, T. D. LaToza, and J. Beaton, "More natural end-user software engineering," *WEUSE IV?08*, 2008. 4.2

[32] B. A. Myers, J. F. Pane, and A. Ko, "Natural programming languages and environments," *Communications of the ACM - End-user development: tools that empower users to create their own software solutions*, pp. 47–52, 2004. 4.2

[33] U. Farooq and D. Zirkler, "Api peer reviews A method for evaluating usability of application programming interfaces," 2010. 4.2

[34] L. V. M. Diaz, "Programming languages as user interfaces," *Proceedings of the 3rd Mexican Workshop on Human Computer Interaction*, 2010. 4.2

[35] S. Hanenberg, "Criteria for evaluation of visual programming languages," *OOPSLA '10 Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pp. 22–35, 2010. 4.3

[36] L. Prechelt, "An empirical comparison of c, c++, java, perl, python, rexx, and tcl," *IEEE Computer*, pp. 1–8, 2000. 4.3

[37] A. Stefik and S. Siebert, "An empirical investigation into programming language syntax," *ACM Transactions on Computing Education (TOCE)*, vol. 13, 2013. 4.3, 12.3

[38] T. Preston-Werner, C. Wanstrath, and P. Hyett, "Github." `https://github.com/`, 2016. Used: 19/05/16. 4.3

[39] S. Nanz and C. A. Furia, "A comparative study of programming languages in rosetta code," *ICSE '15 Proceedings of the 37th International Conference on Software Engineering - Volume 1*, 2015. 4.3

[40] B. Ray, D. Posnett, V. Filkov, and P. T. Devanbu, "A large scale study of programming languages and code quality in github," *ACM FSE'14*, 2014. 4.4

[41] S. Hanenberg, "An experiment about static and dynamic type systems doubts about the positive impact of static type systems on development time," *OOPSLA/SPLASH'10*, 2010. 4.4

[42] S. Hanenberg and S. Siebert, "An empirical investigation into programming language syntax," *Transactions on Computer Education*, 2013. 4.4

[43] A. Stefik and S. Hanenberg, "The programming language wars," *SPLASH - Systems, Programming, and Applications*, pp. 283–299, 2014. 4.4

[44] A. R. Feuer and N. H. Gehani, "Comparison of the programming languages c and pascal," *ACM Computing Surveys (CSUR)*, pp. 73–92, 1982. 4.5

[45] J. L. Murtagh and J. A. H. Jr., "A comparison of ada and pascal in an introductory computer science course," *SIGAda '98 Proceedings of the 1998 annual ACM SIGAda international conference on Ada*, pp. 75–80, 1998. 4.5

[46] L. Tang, "A comparison of ada and c++," *TRI-Ada '92 Proceedings of the conference on TRI-Ada '92*, pp. 338–349, 1992. 4.5

[47] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman, "Benchmarking java against c and fortran for scientific applications," *JGI '01 Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pp. 97–105, 2001. 4.5

[48] S. S. Chandra and K. Chandra, "A comparison of java and c#," *Journal of Computing Sciences in Colleges*, pp. 238–254, 2005. 4.5

[49] D. Benyon, *Designing Interactive Systems - A comprehensive guide to HCI and interaction design*, pp. 232–233. Pearson Education Limited, 2 ed., 2010. 4.6

[50] N. N. Group, "Why you only need to test with 5 users." `https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/`. Used: 26/04/16. 4.6

[51] J. Kjeldskov, M. B. Skov, and J. Stage, "Hci lab technical report." `http://vbn.aau.dk/en/publications/the-usability-laboratory-at-cassiopeia(47a9c780-c9dc-11dd-a016-000ea68e967b).html`, 2008. Used: 13/05/16. 6

[52] M. Faldborg and T. L. Nielsen, "Type systems and programmers A look at optional typing in dart," p. 77, 2015. 7

[53] L. C. Pedersen and M. Faldborg, "Designing larm: Programing with nothing but your voice," 2014. Used: 08/02/16. 7

[54] R. Koitz and W. Slany, “Empirical comparison of visual to hybrid formula manipulation in educational programming languages for teenagers,” *PLATEAU ’14 Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, no. 3, pp. 21–30, 2014. 7

[55] A. Stefik, E. Pierzina, and K. Ritter, “Quorum’s home page.” `http://www.quorumlanguage.com`, 2014-2016. Used: 11/05/16. 8

[56] D. Thomas, “Codekata.” `http://codekata.com/`. Used: 12/05/16. 9.1

[57] D. Ho, “Notepad++ home page.” `http://notepad-plus-plus.org`, 2016. Used: 11/05/16. 10

[58] A. Stefik and E. Gellenbeck, “Empirical studies on programming language stimuli,” *Software Quality Journal*, pp. 65–99, 2011. 12.3

[59] J. Sánchez and F. Aguayo, “Blind learners programming through audio,” *CHI EA ’05 CHI ’05 Extended Abstracts on Human Factors in Computing Systems*, pp. 1769–1772, 2005. 12.3

[60] J. Stylos and S. Clarke, “Usability implications of requiring parameters in objects’ constructors,” *ICSEInternational Conference on Software Engineering*, pp. 529–539, 2007. 12.3

Part VI

Appendices

Appendix A

List of Abbreviations

PLATEAU Evaluation and Usability of Programming Languages and Tools

HCI Human-Computer Interaction

UK United Kingdom

USA United States of America

FPL First Programming Language

TBB Threading Building Blocks

IDE Integrated Development Environment

API Application Programming Interface

IDA Instant Data Analysis

VDA Video Data Analysis

RPG Role-Playing Game

DVR Digital Video Recorder

OO Object-Oriented

Appendix B

Task Sheet

Task 1:
Imagine a simple supermarket billing system which can specify orders and calculate the total price of ordered items. For the sake of simplicity, we work with oranges and bananas as our products. Oranges cost 5$ per piece and bananas 4$ per piece, respectively. Create a system that:

- Can calculate the total price given a number of oranges and bananas bought.
- Adds a different price for buying a specific amount of an item
- Make triplets of oranges cost 10$ in total instead of 15$
- Make 5 bananas cost 10$ instead of 20$
- Adds a discount of 10% to the total price for regular customers

Task 2:
Imagine you have 2 football teams and each team has an equal amount of players. Each player has both his first and last name written down as well as their age. Try to find the following things:

- 2 or more players with the same first or last name in the same team
- 2 or more players with the same first or last name across the two teams
- 2 or more players with the same first name and age in the same team

Task 3:
Imagine you have a simple Role playing game. You have a base character which can be specialized in different classes such as Warrior, Mage etc. Every character has a certain amount of hitpoints and has the ability to attack other characters.

- Create a system for characters who all have:

  - Hit points and the ability to replenish them
  - The ability to attack other characters
- Allow a character to have a specific class
- Add a specific unique resource to every class (Warriors get fury, Mages get mana)
- Add a special unique attack to every class (Warriors get “Execute”, Mages get “Fireball” etc.)
  - These unique attacks spend the unique resource, respectively (e.g. Fireball costs 10 mana)
- Add the ability for every class to replenish their unique resource.

Task 4:
For some given text (for example your full name), write a procedure which:

- Prints the text in reverse order
- Prints the letters from the text in an alphabetical order
- Finds if there are duplicate letters in the text and if there are, list how many are duplicated (e.g. “Tommy” will give the result of 1, while “Christensen” has 3)

Appendix C

Sample Sheet

General information & code examples

Quorum is an evidence-based programming language, designed from the outset to be easily understood and picked up by beginners. One of the design decisions taken includes the full omit of brackets when defining scopes. Keywords in the language make use of a more natural mapping to the real world, such as "text" for strings, "number" for doubles and floats and "repeat" for loops. Conditional statements such as if-statement are always ended with the keyword "end" which specifies the end of scope.

Data types

```
integer a = 5
number b = 10.2
text c = "John"
boolean d = true
```

Type conversion:

```
text someText = "5.7"

number someNumber = cast (number, someText)
```

Simple operation with arrays and conditional statements

The following code creates an array a with some randomly placed elements. It then sorts the array and iterates through the array to create an output with all the elements.

```
use Libraries.Containers.Array
action Main
        text unordered = "fdebaac"
        Array<text> a = unordered:Split("")
        a:Sort()
        integer i = 0
```

```
        text out = ""
        repeat while i < a:GetSize()
                out = out + a:Get(i) + ";"
                i = i + 1
        end
        output out
end
```

Output is: `a;a;b;c;d;e;f;`

This is an example of an action using if- else statements

```
action checkIntervals(integer i)
    if i < 10
        output "it is less than 10"
    elseif i = 10 or i > 10 and i <= 15
        output "it is less than or equal to 15 but greater or equal to 10"
    else
        output "it is greater than 15"
    end
end
```

Classes & Inheritance

To demonstrate classes and inheritance in quorum, we use the example of the animal family felidae and a cat belonging to that family:

First the superclass felidae looks like this:

```
class felidae
    text name = "Sebastian"

    public action Paws() returns integer
        return 4
    end

    action Purr()
        output name + ": rhrhrhrhrhrhrhrhrhrhrhrh"
    end
end
```

We then create the cat subclass like this:

```
class cat is felidae
    action Meow
        output parent:felidae:name + ": meow"
    end
end
```

To show the code in action we then use a main action that looks like this:

```
action Main
    cat sampleCat
    sampleCat:Purr()
    sampleCat:Meow()
    output sampleCat:Paws()
end
```

Where we instantiate a cat and call both the action from the superclass and the subclass giving the output of:

```
Sebastian: rhrhrhrhrhrhrhrhrhrhrhrh
Sebastian: meow
4
```

Worth noting is that we need to specify that the action Paws is public before we can call it from outside the class since it returns something (actions that does not return something are public by default). Likewise, if we in main where to write something like:

```
output sampleCat:parent:felidae:name
```

In order to access the name property, it would give an error since the name is not public.

Appendix D

Interview Questions

1. What do you think about the language? Was it easy to learn?
2. Did you find some of the design odd?
3. How does Quorum relate to other languages you have experience in?
4. How did you find the tasks? Were they too challenging or too easy?
5. What do you think about coding without a compiler?

Appendix E

Interview notes

As previously mentioned, the interviews with the participants were recorded in order to preserve the necessary feedback which helped in analysing the results. Instead of providing the entirely of the interviews in the form of transcripts, we decided to condense the information in key points instead. This helped us to analyse the data from the interviews much easier and find out how many occurrences of a given problem there are across all the participants. Additionally, this section encapsulates the essential parts of each interview and highlights what every participant had to say in terms of feedback, suggestions for further improvements and encountered problems.

E.1 Participant #1 (Pilot Test)

- Thought that using colon (`:`) instead of dot (`.`) was weird, both because it goes against the norm (the participant had experience with several languages which use the dot notation) and because dot is easier to type.
- Thought that tasks are trivial to understand but take time to code
- Task 1 was too broad in the definition causing the task to be too large and time-consuming and the participant to spend time on unintended things.
- Mentioned that although repetitive to a certain extend, task 2 was tricky and very good at conveying that you have to pay attention when copying code. (*He actually fell into that trap and he did not realise it up until the facilitator intervened and pointed that out.)
- Thought that tasks 2 and 4 were quite good in terms of their intended purposes while task 3 (operation on strings) was trivial and very similar to task 2.
- He found the samples of conditional use not being able to clearly convey the differences between Quorum and other known languages he had experience in. In particular == vs. = and `and` vs `&&` did not stand out.

- Thought that it was good that the sample sheet was split in categories to make it easier for the facilitator to reference them when asked.

E.2 Participant #2

- Thought that a lot of the notations were unintuitive because they differed from the mathematical norm
- Found the keyword `action` confusing
- Thought that using `loop` would be easier than `repeat`
- It was difficult for him to devise the code needed for solving the tasks, although he found the mathematics behind quite easy
- It was daunting to not have any fallback or assistance when trying to code and learn how to code without a compiler

E.3 Participant #3

- Suggested that we should add specific values for task 3
- Wondered how to define return types of an action
- Quorum does not have parameterized constructors
- Suggested that we add how to get the size of an array with an in-build action
- Forgot to increment loop counters
- Forgot to add the `repeat` keyword
- Thought that Quorum has a limited number of looping constructs, but it is easy to learn, write and understand
- Quorum is very terse
- Thought that `output` makes more sense than using `print`
- Thought that `returns` of an action seems intuitive
- Liked the `is` keyword for class inheritance
- Thought that Quorum is similar to C, with a different syntax (programmer-friendly C)
- Thought that the lack of parameterized constructors is not that limiting, but does not have enough experience to say with a certainty

- Found the tasks not too challenging
- Thought that not using a compiler is not much of a hindrance
- Found the example sheet indispensable and very helpful
- Suggested that we could add more examples for loping constructs

E.4 Participant #4

- Found it strange to use words as a means of closing scopes instead of brackets as well as using colons instead of dots
- Thought the languages is straightforward and easy to use
- Used a `float` instead of a `number` keyword, as well as `string` instead of `text`
- Forgot to add `returns` keyword at the end of an action
- Forgot to increment the counters on loops
- Had some problems with scoping by making use of the `end` keyword
- Found the tasks specific, understandable and clear
- Thought that `.` makes more sense than `:`
- Suggested that we add an example of method inheritance on the example sheet
- Suggested that we change the / on task 2 with an "or"
- Suggested that we add a sort action on the example sheet constructs

E.5 Participant #5

- Found Quorum is similar to C
- Quorum has similar design to other languages `string` instead of `text`
- Tended to over-complicate things and thus - over-engineer the tasks
- Made use of the example sheet quite frequently
- Coding without a compiler was unpleasant and felt like being in an exam, unable to get a feedback from what's being written down (Does not allow a great deal of experimentation)
- Had difficulties with the syntax of arrays - using the `[]` notation instead of the `get(i)` inbuilt method

- Forgot to write the import for using arrays, as specified on the example sheet
- Found the tasks very good at conveying our intended purposes and easy to understand
- Found the amount of tasks good and reasonable
- Found Task 3 to be a bit tricky since you have to specifically think in terms of inheritance from the start
- Found the example sheet informative and referred to it several times

E.6 Participant #6

- Quorum's design seems a bit confusing
  - Closing the scopes of If-statements with `end`
  - Lack of parameterized constructors
  - Lack of a for-loop
- Found the tasks very good and the example sheet - very concise
- Found coding without a compiler scary without "the safety net"
- Typed = instead of == for an inequality operator
- Thought it might be more intuitive to use a Get method directly compared to how it is being used in the language
- Typed `.` instead of `:`
- Thought that ending classes with something different than the `end` keyword will make more sense
- Found closing the scope of if-statements with the `end` keyword confusing and said that brackets would make it more readable (similarly to OO languages such as Java and C#)
- Found Quorum less verbose than other OO languages
- Forgot to increment the counter variable outside of a loop
- Although the participant over complicated the tasks based on the provided description, he found them very good and efficient at what they try to convey
  - Task 2 - the description of the task seems rather confusing, which made the participant to over-engineer the solution
  - Task 3 - doable
- Found the example sheet contains enough content in order to solve the tasks
- Had a few suggestions how to improve the overall look of the example sheet

E.7 Participant #7

- Found Quorum similar to Pascal and C#
- Liked certain parts of the language and disliked others
- Found the use of `repeat` unnecessary since it does not make sense in conjunction with the standard loop wording
- Noticed that you have to close a class/action with an `end` keyword
- Suggested that implicit type casting would be better for novices
- `:` used in different scenarios might be confusing
- Thought that the `returns` keyword can have a better placement in the action's signature
- Noticed that you have to use a library for an array
- Said that the `end` keyword does not make much sense and rather see a `begin-end` scoping construct, similar to Pascal and Python - only indentation
- Casting data types could be dangerous for novices
- Found the `returns` keyword's placement not so intuitive
- Found the `end` keyword for if-statements not so adequate, can use indentation instead similar to Python
- Found the tasks very good:
  - Task 2's challenge of reusing code is a good exercise
  - Task 2 could have a 2 predefined lists with names
- Said that the task encompass a good portion of constructs
- Suggested we could add a setup for easier start with the tasks
- Suggested we give better titles on the examples sheet and better indexing when looking for things
- Coding without a compiler did not matter that much in his opinion
- Found it great that the facilitator could say if the task is done or not
- `GetSize()` and `Add()` in-build methods examples were missing
- Acknowledged that the code samples are highlighted and there are working examples
- Said that we should be consistent with the working titles

E.8 Participant #8

- Found Quorum intuitive to use, but limited in terms of available constructs
- Suggested that `returns nothing` would be intuitive
- Found the naming of keywords inconsistent (Arrays with capital A and everything else with small letters)
- Found it confusing not to use indentation for scopes
- Found the lack of semicolons a very good thing
- Liked the `is` keyword for class inheritance
- Pointed out the lack of a `continue` construct for loops
- Would have liked more features from functional programming
- Suggested we could make the “or” and “and” statements bolded in task 2
- Noticed the lack of an aggregate += operator
- Quorum reminds him of OO languages and similar to Python
- Would have liked a summary of all the examples on the examples sheet
- Found the examples not so sufficient per task
- Suggested that we could highlight important parts on the task sheet
- Found the lack of a compiler while coding “dangerously scary”
- Over-engineered task 1
- Suggested that we could have an additional sheet with solutions to the tasks
- Separate each task on a separate paper so it is easier to navigate

Appendix F

Participant code

F.1 Participant #1

```
use Libraies.Container.Array
use public

action Main
Array<Cucumber> a
a:add(Cucumber c1)
  a:add(Cucumber c2)
  a:add(Cucumber c3)
  a:add(Cucumber c4)
CalcTotal(a)
end


action CalcTotal(Array<Cucumber> arr)
  number total = 0
  integer i = 0
  total = Cucumber.Price(arr.GetSize())
  output "total = " + total

end

Class Cucumber
  integer id
  number price
  number bulkPrice
  integer bulkCount
  number percentageDiscount
  boolean bORp

```

```
30
31    public action Price(integer amount)
32      integer remains = mod amount
33      integer numdiscount = amount / bulkCount
34
35      number value
36      if bORp
37        value = remains* price + numdiscount * bulkPrice
38      else
39        value = 100 - percentageDiscount * price *amount
40      end
41
42      return value
43    end
44  end
45
46
47  //TASK2
48
49  Class Player
50    public text FN
51    public text LN
52    public integer age
53
54    action make (text first, text last, integer _age)
55    FN = first
56    LN = last
57    age = _age
58    end
59  end
60
61  action Main
62  Aray <Player> T1
63  Array <Player > T2
64
65  Player p1
66  Player p2
67  Player p3
68
69  Player p4
70  Player p5
71  Player p6
72
73  p1:make("a","b",10)
74  p2:make("a",zebra,1)
75  p3:make("gi", "joe", 65)
```

```
76
77  T1:add(p1)
78  T1:add(p2)
79  T1:add(p3)
80
81  p4:make("anotherguy","b",20)
82  p5:make("c","c",11)
83  p6:make("d","d",25)
84
85  T2:Add(p4)
86  T2:Add(p5)
87  T2:Add(p6)
88
89    public action FindFFNLNbetweenTeams returns integer
90      integer i = 0
91      integer j = 0
92      integer found = 0
93      repeat while i < T1:GetSize()
94        repeat while j < T2:GetSize()
95           if T2:Get(j):FN = T1:Get(i):FN
96            found = found +1
97          else if T2:Get(j):LN = T1:Get(i):LN
98            found = found +1
99          end
100       end
101     end
102     return found
103   end
104
105     public action FindFFNLNinTeam(Array<Player>) returns integer
106     integer i = 0
107     integer j = 0
108     integer found = 0
109     repeat while i < T:GetSize()
110       repeat while j < T:GetSize()
111          if T:Get(j):FN = T:Get(i):FN
112           found = found +1
113         else if T:Get(j):LN = T:Get(i):LN
114           found = found +1
115         end
116       end
117     end
118     return found
119   end
120
121     public action FindAgebetweenTeams returns integer
```

```
122     integer i = 0
123     integer j = 0
124     integer found = 0
125     repeat while i < T1:GetSize()
126       repeat while j < T2:GetSize()
127          if T2:Get(j):Age = T1:Get(i):Age
128           found = found +1
129         end
130       end
131     end
132     return found
133   end
134 
135     public action FindAgeinTeam(Array<Player>) returns integer
136     integer i = 0
137     integer j = 0
138     integer found = 0
139     repeat while i < T:GetSize()
140       repeat while j < T:GetSize()
141          if T:Get(j):Age = T:Get(i):Age
142           found = found +1
143         end
144       end
145     end
146     return found
147   end
148 
149 
150     public action FindAgeinTeam(Array<Player>) returns integer
151     integer i = 0
152     integer j = 0
153     integer found = 0
154     repeat while i < T:GetSize()
155       repeat while j < T:GetSize()
156          if T:Get(j):Age = T:Get(i):Age and T:Get(j):FN = T:Get(i):FN
157           found = found +1
158           j = j+1
159         end
160         i = i+1;
161       end
162     end
163     return found
164   end
165 
166 end
167 
```

```
168
169 //TASK 4
170
171 Class Base
172
173 integer hp
174 integer dmg
175
176   action do()
177   end
178   action Attack(Base target )
179   target:takeDamage(dmg)
180   end
181
182   action takeDamage(integer damage)
183   hp = hp - damage
184     if hp >= 0
185       kill()
186     end
187   end
188
189   action replnishHP(integer amount)
190   hp = hp + amount
191   end
192
193   action kill ()
194   delete me
195   end
196
197 end
198
199 Class Warior ia Base
200 hp = 150
201 dmg = 10;
202 integer fury = 100
203
204   action do()
205   fury = fury +1
206   end
207
208   action strongAttack
209     if fury > 10
210       target:takeDamage(dmg+10)
211       fury = fury - 10
212     else
213     output "might knight whines like tiny baby men"
```

```
214       end
215     end
216   end
217
218   Class Mage is Base
219   hp = 70
220   damage = 12
221   integer mana
222
223     action do()
224     mana = mana +1
225     end
226
227
228     action heal (target)
229       if mana > dmg
230       target.replnishHP(dmg)
231       mana = mana - dmg
232       end
233     end
234
235   end
236
237   Action Main
238   for each base
239   do()
240   end
241   end
```

F.2 Participant #2

```
1   action gettotal (integer Oranges,integer Bananas,boolean isregular)
        returns integer
2   integer total=0
3   total=total+Oranges*5+Bananas*4-5*Oranges/3-10*Bananas/5
4   if isregular = true
5   total=total*0.9
6   end
7   refurn total
8   end
9   output gettotal (3,5,true)
10
11  first name  last name  age  team
```

F.3 Participant #3

```
class Test1
  integer OrangePrice = 5
  integer BananaPrice = 4

  action TotalPrice(integer oranges, integer bananas)
    Output oranges * Orangeprice + bananas * BananaPrice
  end

  action TotalPriceWithDiscount(integer oranges, integer bananas, boolean
      regular)
    number result = 0
    repeat while oranges > 3
      result = result + 10
      oranges = oranges - 3
    end
    result = result + oranges * OrangePrice

    repeat while bananas > 5
      result = result + 10
      bananas = bananas - 5
    end
    result = result + bananas * BananaPrice

    if regular
      Output result * 0.9
    else
      Output result
    end
  end
end

class Test2
  Array<Player> Team1
  Array<Player> Team2

  action SameFirstLastNameSameTeam(Array<Player> team) returns boolean
    integer i = 0
    repeat while i < team.GetSize()
      text firstName = team:Get(i):FirstName
      text lastName = team:Get(i):LastName
      integer j = 0
      repeat while j < team.GetSize()
        if not(j == i) and (firstName == team:Get(i):FirstName or lastName
            == team:Get(i):LastName)
```

```
43            return true
44          end
45        end
46      end
47      return false
48    end
49
50    action SameFirstLastNameDifferentTeams() returns boolean
51      integer i = 0
52      repeat while i < Team1.GetSize()
53        text firstName = Team1:Get(i):FirstName
54        text lastName = Team1:Get(i):LastName
55        integer j = 0
56        repeat while j < Team2.GetSize()
57          if firstName == Team2:Get(i):FirstName or lastName ==
                Team2:Get(i):LastName
58            return true
59          end
60        end
61      end
62      return false
63    end
64
65    action SameFirstLastNameSameTeam(Array<Player> team) returns boolean
66      integer i = 0
67      repeat while i < team.GetSize()
68        text firstName = team:Get(i):FirstName
69        integer age = team:Get(i):Age
70        integer j = 0
71        repeat while j < team.GetSize()
72          if not(j == i) and (firstName == team:Get(i):FirstName and age ==
                team:Get(i):Age)
73            return true
74          end
75        end
76      end
77      return false
78    end
79  end
80
81  class Player
82    text FirstName
83    text LastName
84    integer Age
85  end
86
```

```
class Character
  number Health

  action ReplenishHealth(integer amount)
    Health = Health + amount
  end

  action Attack(Character target)
    target:Health = target:Health - 10
  end
end

class Mage is Character
  number Mana

  action Fireball(Character target)
    if Mana >= 10
      Mana = Mana - 10
      target:Health = target:Health - 20
    end
  end

  action ReplenishMana(integer amoutn)
    Mana = Mana + amount
  end
end

class Warrior is Character
  number Fury

  action Execute(Character target)
    if Fury >= 25
      Fury = Fury - 25
      if target:Health < 30
        target:Health = 0
      else
        target:Health = Target:Health - 10
      end
    end
  end

  action ReplenishFury(integer amount)
    Fury = Fury + amount
  end
end

```

```
class Test4
  text Text = "Rasmus Moeller Jensen"
  Array<Text> a = Text:Split("")

  action PrintReverse()
    integer i = a:GetSize() - 1
    text Result = ""
    while i >= 0
      Result = Result + a:Get(i)
      i = i + 1
    end
    Output Result
  end

  action PrintAlphabetical()
    Array<Text> b = a
    b:Sort()
    text Result = ""
    integer i = 0
    repeat while i < b:GetSize()
      Result = Result + b:Get(i)
      i = i + 1
    end
    Output Result
  end

  action FindDuplicates()
    integer i = 0
    integer j = 0
    integer Result = 0
    Array<Text> AlreadyTested

    repeat while i < a:GetSize()
      j = 0
      repeat while j < a:GetSize()
        if not(i == j) and not(a:Get(i) == " ") and not(a:Get(j) == " ")
            and not(AlreadyTested:Contains(a:Get(i))) and a:Get(i) ==
            a:Get(j)
          Result = Result + 1
          AlreadyTested:Add(a:Get(i))
        end
        j = j + 1
      end
      i = i + 1
    end
    Output Result
```

```
  end
end
```

F.4 Participant #4

```
use Libraries.Containers.Array


action CalculatePrice(integar nBananas, number pBananas, integar nOranges,
    number pOranges, boolean regular) returns number
  number totalgroup = ((nBananas / 5) * 10) + ((nOranges / 3) * 10)
  number rBananasPrice = (nBananas % 5) * 4
  number rOrangesPrice = (nOranges % 3) * 5
  number total = totalgroup + rBananasPrice + rOrangesPrice
  if regular total = total * 0.9
  return total
end

//each array entry is a string with name, second name
action FindSameFirstNames(Array<text> teamone, Array<text> teamtwo)
    returns string
  Array<text> SameNames;
  integar i = 0
  integar j = 0
  repeat while i < teamone:GetSize()
    repeat while j < teamtwo:GetSize()
      string pAF = teamone.Get(i).Split(",").Get(0)
      string pBF = teamtwo.Get(j).Split(",").Get(0)
      string pAL = teamone.Get(i).Split(",").Get(1).Trim()
      string pBL = teamtwo.Get(j).Split(",").Get(1).Trim()
      if(pAL = pBL or pAF = pBF) return players.Get(i) + " : " +
          players.Get(j)
      i = i + 1
      j = j + 1
    end
  end
end

//each array entry is a string with name, second name, age
action FindSameFirstNamesAndAge(Array<text> teamone) returns string
  Array<text> SameNames;
  integar i = 0
  repeat while i < teamone:GetSize()
    integar j = 0
```

```
37      repeat while j < teamone:GetSize()
38        string pAF = teamone.Get(i).Split(",").Get(0)
39        string pBF = teamone.Get(j).Split(",").Get(0)
40        integar pAA = cast (integar, teamone.Get(i).Split(",").Get(2))
41        integar pBA = cast (integar, teamone.Get(j).Split(",").Get(2))
42        if(pAF = pBF and pAA = pBA) return players.Get(i) + " : " +
              players.Get(j)
43        j = j + 1
44      end
45      i = i + 10
46    end
47  end
48
49
50
51  class character
52    public integar hp = 100
53    public integar resourceAmount = 100
54
55    public action Attack(character defender, integar amount)
56      defender:hp = defender:hp - amount
57    end
58
59    public action Recover(integar amount)
60      hp = hp + amount
61    end
62
63    public action RecoverResource(integar amount)
64      resourceAmount = resourceAmount + amount
65    end
66  end
67
68  class mage is character
69
70    public string resourceName = Mana
71
72    public action Fireball(character defender, integar amount)
73      parent:character:Attack(defender,amount)
74      parent:character:resourceAmount = parent:character:resourceAmount - 10
75    end
76
77  end
78
79  class warrior is character
80
81    public string resourceName = Rage
```

```
82
83    public action Pummel(character defender, integar amount)
84      parent:character:Attack(defender,amount)
85      parent:character:resourceAmount = parent:character:resourceAmount - 20
86    end
87
88  end
89
90  class taxAccountant is character
91
92    public string resourceName = Money
93
94    public action ChargeWithTaxEvation(character defender, integar amount)
95      parent:character:Attack(defender,amount)
96      parent:character:resourceAmount = parent:character:resourceAmount - 50
97    end
98
99  end
100
101 public action ReverseText(text texttotreverse) returns text
102   text out = ""
103   Array<text> characters = texttotreverse:Split("")
104   integar count = character:GetSize() - 1
105   repeat while count >= 0
106     out = out + characters:Get(count)
107     count = count - 1
108   end
109   return out
110 end
111
112 public action SortCharacters(text string)
113   Array<Text> characters = string:Split(""):Sort()
114   integar count = character:GetSize()
115   integar i = 0
116   repeat while i < count
117     output characters:Get(i)
118     i = i + 1
119   end
120 end
121
122 public action FindDuplicates(text string) returns integar
123
124   Array<text> characters = string:Split("")
125   Array<text> foundChar
126
127   integar i = 0
```

```
128
129   repeat while i < characters:GetSize()
130     integar j = 0
131     repeat while j < characters:GetSize()
132       if characters:Get(i) = characters:Get(j)
133         integar k = 0
134         boolean found = false
135         repeat while k < foundChar:GetSize()
136           if characters:Get(i) = foundChar:Get(k)
137             found = true
138         end
139         if not found
140           foundChar:Add(characters:Get(i)
141
142       j = j + 1
143     end
144     i = i + 1
145   end
146
147   return foundChar:GetSize()
148 end
```

F.5 Participant #5

```
1  action Main
2          action calculateFruit(integer banana, integet orange) returns
               integer
3                  integer orangePrice = 5
4                  integer bananaPrice = 4
5
6                  return orange*orangePrice + banana*bananaPrice
7          end
8
9          action calculateFruit(integer banana, integet orange, boolean
               regular) returns integer
10                 integer orangePrice = 5
11                 integer bananaPrice = 4
12
13                 orangesDiscount = orange/3
14                 orangeRemainder = orange mod 3
15                 bananaDiscount = banana/5
16                 bananaRemainder = banana mod 3
17
```

```
18                sum = orangeRemainder*orangePrice + orangesDiscount*10 +
                      bananaRemainder*bananaPrice + bananaDiscount*10
19
20                if regular
21                        return sum-sum*0.1
22                else
23                        return sum
24                end
25        end
26
27        // firstname,lastnam
28
29        // 0,firstname
30        // 1,lastnam
31        action findPlayers1(Array<text> team) returns Array<text>
32
33                integer i = 0
34                Array<Array<text>> players
35                repeat while i < team:GetSize()
36                        Array<text> player = team:Get(i).Split(",")
37                        players:Add(player)
38                        i = i + 1
39                end
40
41                integer i = 0
42                integer j = 0
43                repeat while i < players:GetSize
44                        repeat while j < players:GetSize
45                        players:Get(i):Get(0) == players:Get(j):Get()
46
47
48
49        end
50
51        class Warrior is character
52
53        end
54
55        class Mage is character
56
57        end
58
59        class character
60                integer hitPoints
61                public action attack(character c)
62
```

```
63                          character:decreaseHitpoint(200)
64                  end
65
66                  public decreaseHitpoint(integer amount)
67                          hitPoints = hitPoints - amount
68  end
```

F.6 Participant #6

```
1   action main
2     integer sum = 0
3     Array <Product> prod = basket:Get()
4     integer count = 0
5     repeat while count<prod:GetSize()
6       sum  = sum + prod:GetProd():GetPrise()
7     end
8
9     integer count2 = 0
10
11    repeate while count< prod:GetSize()
12    if (prod:GetProd == 'oranges' )
13      integer numOfOrn = numOfOrn + 1
14    else
15      integer numOfBan = numOfBan + 1
16    end
17
18      integer tripOrn = numOfOrn / 3
19      integer discountprice = tripOrn  * 10
20      integer normalprice = (numOfOrn - tripOrn) * 15
21      integer totalpriceOrn = discountprice + normalprice
22
23      integer fiveBan = numOfBan / 5
24
25    if basket:GetCustomer():IsRegular == true
26      discountprice = price * 0.9
27
28  end
29
30
31  Task 2
32
33  action Main
34
35    Array <Player> team1 = GetTeamPlayers()
```

```
36    Array <Player> team2 = GetTeamPlayers()
37    team1:Sort()
38    team2:Sort()
39
40
41    integer i = 0
42    repeat while i < team2:GetSize()
43      if team1:GetPlayer(i):GetPlayerFName() = team1:Get(i+1):GetPlayerFName
            or team1:Get(i):GetPlayerLName() = team1:Get(i+1):GetPlayerLName
44      output  'Same team : ' + team1:Get(i):GetPlayerFName  +
            team1:Get(i+1).GetPlayerFName
45
46      else if team1:Get(i):GetPlayerFName() = team2:Get(i):GetPlayerFName or
            team1:Get(i):GetPlayerLName() = team2:Get(i):GetPlayerLName
47      output "different teams :" + team1:Get(i):GetPlayerFName  +
            team1:Get(i+1).GetPlayerFName
48
49      else team1:GetPlayer(i):GetPlayerFName() =
            team1:Get(i+1):GetPlayerFName or team1:Get(i):GetPlayerAge() =
            team1:Get(i+1):GetPlayerAge
50
51      output  'Same team : ' + team1:Get(i):GetPlayerFName  +
            team1:Get(i+1).GetPlayerFName + "Same age"
52      end
53    end
54  end
55
56  Task 3
57
58  class StartGame
59    action Main
60
61    end
62  end
63
64  class Hero
65    integer hitpoints = 100
66    integer replRate = 10
67
68    action replanishHealth()
69    output "Health has been replaneshed from " + hitpoints " to " +
          hitpoints+replRate
70    end
71
72    action attack(Hero H)
73    end
```

```
74
75    action replRes
76    end
77  end
78
79  class Warrior is Hero
80    int fury = 100
81
82    action attack( Hero H)
83    integer attackp = hitpoints - 15
84    H:hitpoints = attackp
85    output H + " has been slayen for " + attackp
86    end
87
88    action spattack( Hero H)
89    integer attackp = hitpoints - 17
90    H:hitpoints = attackp
91    integer furyleft = fury - 10
92    fury = furyleft
93    output H + " has been slayen for " + attackp
94    end
95
96    action replRes
97    fury = fury+10
98    end
99
100 end
101
102 class Mage is Hero
103   int mana = 100
104
105   action attack( Hero H)
106   integer attackp = hitpoints - 12
107   H:hitpoints = attackp
108   output H + " has been slayen for " + attackp
109   end
110
111   action spattack( Hero H)
112   integer attackp = hitpoints - 15
113   H:hitpoints = attackp
114   integer manaleft = mana - 10
115   mana = manaleft
116   output H + " has been slayen for " + attackp
117   end
118
119   action replRes
```

```
120   mana = mana+10
121   end
122 end
123
124 end
```

F.7 Participant #7

```
1  //Task 1
2  action Main
3
4    output CalculateTotal(5, 5)
5    //test the method
6  end
7
8  action CalculateTotal(integer numberOfOranges, integer numberOfBananas,
       boolean regular) returns number
9    integer banana = 4
10   integer orange = 5
11
12   number currenTotal = 0
13
14   currenTotal = (numberOfOranges mod 3) * orange + (numberOfOranges/3)*10
15   currenTotal = currenTotal + (numberOfBananas mod 5) * banana +
         (numberOfBananas/5)*10
16
17   if regular
18     currenTotal = currenTotal*0.9
19   end
20
21   return currenTotal
22 end
23
24 //Task 2
25 use Libraries.Containers.Array
26 action Main
27   Array<player> team1
28   Array<player> team2
29
30
31 end
32
33 action SameTeamNames(Array<player> team) returns Array<player>
34   Array<player> returnArray
```

```
35    integer i = 0
36    integer j = 1
37
38    repeat while i < team:GetSize()
39      repeat while j < team:GetSize()
40        if team:Get(i):FirstName() = team:Get(j):FirstName() or
              team:Get(i):LastName() = team:Get(j):LastName()
41          returnArray.Add(team:Get(i))
42          returnArray.Add(team:Get(j))
43        end
44        j = j+1
45      end
46      i = i + 1
47      j = i+1
48    end
49
50    return returnArray
51  end
52
53  action DiffTeamNames(Array<player> team1, Array<player> team2) return
        Array<player>
54    Array<player> returnArray
55
56    integer i = 0
57    integer j = 0
58
59    repeat while i < team1:GetSize()
60      repeat while j < team2:GetSize()
61        if team1:Get(i):FirstName() = team2:Get(j):FirstName() or
              team1:Get(i):LastName() = team2:Get(j):LastName()
62          returnArray.Add(team1:Get(i))
63          returnArray.Add(team2:Get(j))
64        end
65        j = j+1
66      end
67      i = i + 1
68      j = 0
69    end
70
71    return returnArray
72  end
73
74  action SameTeamAge(Array<player> team) returns Array<player>
75    Array<player> returnArray
76
77    integer i = 0
```

```
78    integer j = 1
79
80    repeat while i < team:GetSize()
81      repeat while j < team:GetSize()
82        if team:Get(i):FirstName() = team:Get(j):FirstName() and
              team:Get(i):Age() = team:Get(j):Age()
83          returnArray.Add(team:Get(i))
84          returnArray.Add(team:Get(j))
85        end
86        j = j+1
87      end
88      i = i + 1
89      j = i+1
90    end
91
92    return returnArray
93  end
94
95  //Task 3
96
97
98  //Task 4
99  action Main
100   text t = "HenrikGeertsen"
101
102   integer i  = t:GetSize()-1
103   text out = ""
104   repeat while i > 0
105     out = out + t:GetCharacter(i)
106     i = i - 1
107   end
108   output out
109
110   Array<text> a = t:Split("")
111   a:Sort()
112   i = 0
113   out = ""
114   repeat while i < a:GetSize()
115     out = out + a:Get(i)
116     i = i + 1
117   end
118   output out
119
120   i = 1
121   boolean found = false
122   integer duplicates = 0
```

```
123   repeat while i < a:GetSize()
124     if (a:Get(i) = a:Get(i-1))
125       found = true
126     else
127       if (found)
128         duplicates = duplicates + 1
129       end
130       found = false
131     end
132   end
133   output duplicates
134 end
```

F.8 Participant #8

```
1  use Librarises.Containers.Array
2
3  class fruit
4    number _price = 0;
5
6    public action RaisePrice(number newPrice)
7      me:price = newPrice
8
9  class banana is fruit
10   number _price = 4
11
12 class orange is fruit
13   number _price = 5
14
15 class bananas
16   Array<banana> _bananas
17   public action addBanana()
18     _bananas:add(banana)
19
20 class oranges
21   Array<orange> _oranges
22   public action addOrange()
23     _oranges:add(orange)
24
25 class calculator
26   public action isRegular(bool isRegular, number price) returns number
27     if isRegular = true
28       return price = price * 1.10
29
```

```
30
31
32  action Main
33    integer i = 0
34    number OTotal = 0
35    number BTotal = 0
36
37    oranges orangesLst
38    bananes bananasLst
39    calculator c
40
41    repeat while not(i = 10)
42      orangeLst:addOrange()
43      bananasLst:addBanana()
44
45    repeat while i < orangeLst:GetSize()
46      Ototal = OTotal + orangeLst:_oranges:Get(i):_price
47      if (i mod 3) == 0
48        0Total = 0Total - 5
49
50    regularPrice = c:isRegular(true, OTotal)
51    normal = c:isRegular(false, 0Total)
52
53    repeat while i < bananasLst:GetSize()
54      Ototal += bananasLst:_bananas:Get(i):_price
55      if (i mod 3) == 0
56        BTotal = BTotal - 10
57
58    repeat while i
59
60
61
62  action Main2
63    int nrPlayers = 11
64
65    Array<text> fullNames1
66    Array<text> fullNames2
67
68    fullNames1:add("martin, fruensgaard, 24")
69    fullNames2:add("Tommy, something, 23")
70
71    int i = 0, j = 0;
72    repeat while i < fullNames1:GetSize()
73      repeat while j < fullNames2:GetSize()
74        Array<text> name1 = fullNames1:Get(i):Split(",")
75        Array<text> name2 = fullNames2:Get(j):Split(",")
```

```
76
77        if(name1:Get(0) = name2:Get(0) or name1:Get(1) = name2:Get(1))
78          output "BINGo!<3 2 players: " + fullNames1(i) + " and "
                fullNames2(j)
79
80
81    int i = 0, j = 0;
82    repeat while i < fullNames1:GetSize()
83      repeat while j < fullNames2:GetSize()
84        Array<text> name1 = fullNames1:Get(i):Split(",")
85        Array<text> name2 = fullNames1:Get(j):Split(",")
86
87        if not(name1 = name2)
88          if(name1:Get(0) = name2:Get(0) or name1:Get(1) = name2:Get(1))
89
90    int i = 0, j = 0;
91    repeat while i < fullNames1:GetSize()
92      repeat while j < fullNames2:GetSize()
93        Array<text> name1 = fullNames1:Get(i):Split(",")
94        Array<text> name2 = fullNames1:Get(j):Split(",")
95
96        if not(name1 = name2)
97          if(name1:Get(0) = name2:Get(0) and  name1:Get(2) = name2:Get(2))
```