From Educational Programming to Professional Programming

Group DPT906E15 - Room X.X.XX

4 February - 1 June

Date	Jais Morten Brohus Christiansen
Date	Henrik Vinther Geertsen
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:

Whist on Mobile Devices

Project period:

4 February - 1 June

Project group:

DPT906E15

Participants:

Jais Morten Brohus Christiansen
Henrik Vinther Geertsen
Svetomir Kurtev
Tommy Aagaard Christensen

Supervisor:

Bent Thomsen

Abstract:

Pages: 26

Appendices: 0

Copies: 2

Finished: 1 June 2015

Contents

1 Introduction **1**

I Problem Analysis **3**

2 Error-Prone Areas for Novices **4**

2.1 Syntax and Semantics 4

2.2 Pragmas 5

2.3 Programming Paradigms 5

3 Programming Languages and Tools **7**

3.1 Text Based Programming Languages 7

3.1.1 Text Based Educational Programming Languages 7

3.1.2 General Purpose Programming Languages 10

3.2 Visual-based Programming Languages 12

3.2.1 Scratch 13

3.3 Comparison 13

4 Teachers and Programming **14**

4.1 Current State 14

4.2 Discussion 15

II Language Comparison **16**

5 Preliminaries **17**

6 Criteria **18**

7 Comparative Analysis **20**

8 Usability Evaluation **21**

9 Results **22**

III Bibliography **23**

Bibliography **24**

Chapter 1

Introduction

Traditionally, programming have been seen as a specialized skill, only relevant for people who work on developing new software. While several attempts are made at making languages more accessible to everyone TODO: reference logo and probably some more, they ultimately failed to get enough widespread use to change this perception. As a result, programming education has been targeted towards college or university students and often had a focus on teaching the student to use professional programming languages like JavaTODO: can probably use source about the widespread use of Java here. The goal here is to create competent software engineers with the ability to work with powerful tools on complicated software.

In recent years, programming is starting to be seen as an essential skill for living in our modern digital society. This has lead to many countries making programming a mandatory subject in primary schools. In this setting, the education is more focused on giving children an idea of what it is like to work on software and to teach more generic skills, such as problem solving and collaborative communication. This leads to this education usually being given in visual educational languages like Scratch [1] for their intuitiveness and simplicity. This means that different levels of education differ in what they teach. First, the difference in programming languages, as to between being taught an educational language and a professional language. While the educational language has the advantage of being intuitive, it does not have the expressiveness and robustness for large projects that professional languages have. Second, the computer science education needs to cover a lot of topics to give a sufficient understanding to work professionally with software, where the kids education leaves out a lot of the topics, although this may vary between teachers. Using the list of topics from "What do Teachers Teach in Introductory Programming?" [2] as a reference, we can for example say that topics like algorithm design and debugging are likely to be taught in the kids education. Meanwhile, topics like algorithm efficiency, pointers and object oriented programming are usually left out.

Kids now being taught programming in primary school is great for the general digital literacy of people, but the education can not teach everything necessary to do professional programming. We want to work with teaching the skills for professional programming using the already taught knowledge from primary school. To do this we first need to establish what level people have after working in the educational

language and what knowledge is needed to work professionally with programming. This paper aims to do this as well as suggesting some ideas for ways to bridge that gap.

TODO: JAIS: Should we mention the problem with who is actually teaching the kids in primary school? There is a problem in Denmark, as it is the physics teacher, as far as I know. TODO: JAIS: We need some more stuff here, perhaps chapters or sections. Such as Preliminaries, Previous Work, and a section where we mention the initial problems.

Part I

Problem Analysis

Chapter 2

Error-Prone Areas for Novices

For a person new to programming, different constructs and concepts can be so confusing that this person might give up without much effort. This area is of great interest to many studies, as it can help future generations in learning programming with ease. But to find solutions, the difficulties that can arise when learning programming and the concepts that follow must be known.

This chapter focuses on the different aspects of learning programming, which can be difficult to grasp for novices. The different aspects and concepts are found by previous studies as well as subjective speculation.

2.1 Syntax and Semantics

As known, a programming language is based on the syntactical rules and the semantic relations. These concepts can be hard to grasp at first, and can be even harder to understand in relation and when used in a practical solution.

One of the most error prone areas for novice programmers is the basic syntax [3]. This consists of brackets, semicolons, commas, and other such symbols, symbolizing control for the program. This problem might relate to an even greater problem in understanding the strict control that is needed when writing code in general. When writing code, even the smallest mistake or forgotten symbol leads to a compiler error. This error margin isn't seen very often in other lines of work, and might discourage the novices from keep trying.

Understanding what a line of code does in itself might be hard for some new programmers. Understanding the connection of the whole program, and what the single line does for the result is even harder. The semantics can lead to confusion, as the program grows bigger. Some times, the novice programmer is even discouraged from even trying, as the connection between the code and what it results in is not clear.

2.2 Pragmas

TODO: I still need to figure out why it's called pragmas, or if it's just a made-up word. For now, I will leave it out of anything but the title. Also, I haven't been able to find anything on this yet.

A programming language is built on syntax and semantics. To learn how this works can be effective in programming, but programmers often don't think in these terms. Experienced programmers know their way around the basic programming principles and constructs, which is more or less the same in all languages. Programmers often think in patterns, some standardized way for them to program. The logic composition of the elements at hand is often the key for most, working on the idea, in stead of the specific language's behaviour. Even though all programmers have a pattern of programming, good or bad, there is a base line for standard programming patterns. They vary slightly from paradigm to paradigm, but at some level, there is a common thought on the structuring of code. This common coding practice is hard to find and to measure, which leads to no TODO: very little? work around this area. Although, it should be taken into account that this way of finding code patterns might be much more useful than teaching syntax and semantics.

2.3 Programming Paradigms

Different paradigms each have their different difficulties. Many programmers first touch programming through an imperative or procedural approach. Others start out with an object oriented programming language.

Procedural programming has its values in its very straight forward and easily trackable nature. On the other hand, it is hard to see the connection to real world problem solutions, as the very strict text-based structure doesn't resemble these much. Nevertheless, certain tools are used today for teaching, such as Scratch (have we described these yet?), which makes procedural programming a valid learning approach.

Object oriented programming (OOP) has it's values in representing real world problems, and how a solution can be modelled. As OOP is mostly based on classes, being the static description of an object, and objects, being the dynamic model of a real world phenomenon, the concepts of the paradigm can be easily grasped. Of course, this fact demands a teaching method suitable for the novice programmers being taught. On the other hand, OOP is often in relation to procedural programming seen as not being something else, but the same, only with OO features [3]. This leads to a problem of both understanding the very basic concepts of programming, such as control structures (loops and selections), and understanding the OO approach.

Functional programming is a paradigm which uses functions, instead of procedures or objects, as the building blocks of a program. Having its roots in lambda calculus, computations in FP are treated as the evaluation of functions where change of state is avoided and the data is immutable which ,in turn, prevents the introduction of side effects in programs. Additionally, many concepts of the imperative approach can be simulated in Functional programming(ex. control structures are expressed in terms of

recursion) giving as much of a control or power when building programs. However, the notion of using functions as a natural abstraction instead of objects modelled after the world introduces difficulties for novice programmers in understanding the fundamental concepts of Functional programming. Therefore the paradigm is rarely selected as a choice of teaching programming in introductory courses.(TODO: we might need a reference here) TODO: JAIS: So, what keeps people from using functional programming in stead of imperative?

It is discussed widely what approach is the most efficient teaching method (TODO: need some refs here). For instance, some say it is necessary to learn the concepts of OOP before learning to code, and some say the basic constructs are necessary before learning about different paradigms and advanced structures. The procedural approach is being taught in elementary school in various languages (TODO: assumption, need refs). In OOP, the question is often what teaching methods are used to make students understand the concepts of the paradigm. Studies have shown a better effect when teaching about the concepts before actually coding [4]. Another approach to pursue could be the "object-first approach". As the name implies, this approach searches to understand the logic behind objects before anything else. This method can both imply teaching concepts or coding before anything else, and can both be used by novices or by students that have already learned the basics of programming. Although, there is still discussion on whether novices should be taught though this approach, or the classic "algorithms-first approach" [**?**]. One drawback is the fact that the concepts of OOP might be seen by the novices as the basics for programming itself, instead of the basic concepts, which could lead to a block when exploring other paradigms.

Some modern languages, such as C# and Java, have become what one might call a "multi-paradigm language". In these examples, they started out being OOP languages, but now they have implemented various features of other paradigms, such as functional and logic programming. These languages could be considered when learning to program, as the drawback from changing to other languages could be limited.

Chapter 3

Programming Languages and Tools

With the growing distribution of computers and mobile devices, e.g. as laptops, smart phones and tablets, the need of programming languages and tools to operate on these machines becomes more and more relevant. Using such tools and languages efficiently, however, often requires much skill and technical aptitude, which in turn takes considerable time and dedication to develop. From the perspective of novice programmers, programming can be extremely hard and overwhelming to get into, especially if they are given no introductory tools and guidance.

This chapter focuses on the distinction between visual and text based programming languages, analysis of both categories as well as some notable examples. Additionally, these categories are further explored by means of how well they fit in an educational setting.

3.1 Text Based Programming Languages

TODO: Watch the Smalltalk talk, and maybe add something about it here

Text-based programming languages have two distinct subcategories; one is the text-based educational programming languages for novices, and the other is the general purpose programming languages. Event though general purpose programming languages can be used to teach programming to novices, the two categories are distinguished in where the focus of the design has been. This chapter will explore a set of programming languages in both of these categories.

3.1.1 Text Based Educational Programming Languages

This section will provide a description of some different text based educational programming languages and constructs.

Turtle Programming

One of the first programming languages that added constructs for learning programming was LOGO. It did not have the constructs from the start, but after 12 seventh-grade students[1] worked with LOGO for a year (1968-1969), Seymour Papert, one of the developers of LOGO, proposed the Turtle as a programming domain that could be interesting to people at all ages. He proposed it since the demonstration had confirmed that LOGO was a learnable TODO: JAIS: Everything is learnable for novices, some stuff is just easier programming language for novices, but he wanted the demonstration extended to lower grades, ultimately preschool children. Constructs for Turtles was then added to LOGO and has since been widely adopted in other programming languages such as SmallTalk and Pascal, and more recently Scratch [5].

A Turtle can be a visual element on a screen or a physical robot. In Scratch, the Turtle can be any sprite chosen by the user. In LOGO the Turtle is controlled by a set of commands which are:

- FOWARD X, moves the Turtle X number of Turtle steps in a straight line
- RIGHT X, turns the Turtle X number of degrees in a clockwise direction
- LEFT X, turns the Turtle X number of degrees in a counter clockwise direction
- PENDOWN, makes the Turtle draw
- PENUP, makes the Turtle stop drawing

These commands make up the essence of Turtle programming and the functionality is also present in the other languages which has implemented Turtle programming, maybe using different keywords. Some languages has expanded on these commands, e.g. in Scratch, one can change the color, size and shade of the pen. The commands can be part of user defined functions. Examples could be functions called *SQUARE* or *TRIANGLE* which would draw a square and a triangle respectively using the commands shown. These functions can be part of other functions, e.g. a function called *HOUSE* would use a mix of the commands for correct positioning and then the *SQUARE* and *TRIANGLE* functions to draw the house itself [6].

Turtle programming is not only meant as a tool for learning to program. Seymour Papert states that it is meant as an *Object-to-think-with*. This means that it is supposed to give children a way of relating new topics to something they already know. Alan Kay shows an example of this in a talk, where he uses a car sprite (the turtle in this case) to visualize acceleration. He does this by programming a loop for the car that for each iteration makes a circle showing where the car have been and then moves the car forwards to a new position. In each iteration he also increases the distance that the car moves forwards, meaning that the distance between the circles becomes greater and greater, thus visualizing the car accelerating [7].

[1] From Muzzy Junior High School in Lexington, Massachusetts.

Small Basic

Small Basic is a text based programming language with its whole purpose being teaching novices to program. It is not meant as a language one should keep using, but as a tool for learning programming principles and then "graduating" to learn more advanced languages. In the developers internal trials they have had success with teaching programming to kids in the ages of 10 and 16, but it is intended for novices in general, so it is not specifically made for that age group [8]. It is perhaps one of the last languages with this purpose that is still being updated[2]. It seems as if the focus of teaching programming to novices has changed to visual programming languages, tools for learning existing languages that is meant to be used professionally or languages that are novice friendly, but still meant to be used professionally.

The Small Basic project consist of three pieces: The language itself, an IDE and libraries. This means that one has to use the bundled IDE to program in Small Basic. According to their FAQ [8], the language takes inspiration from an early variant of BASIC but is based on the modern .NET Framework Platform. It is much smaller than Visual Basic, it consists of 14 keywords, and supports a subset of the functionality that Visual Basic .NET supports. It does not have a type system and all variables are global and always initialized as to avoid confusion regarding scopes. It is also imperative and does not use or expose beginners to the concept of object orientation.

To learn programming with Small Basic, its website provides a tutorial for getting familiar with the language and the IDE [9], and a curriculum. The curriculum can be downloaded for offline use, and teaches general programming topics using Small Basic [10]. Through the introduction one will learn that there are two different window types a program can be run in, either the *TextWindow* which is a regular console, or in the *GraphicsWindow* where graphics can be drawn and so on. So it is possible to create e.g. games in Small Basic. Turtle programming is also supported.

The Small Basic IDE aims to help novices as well. The IDE is made up of different parts, which can be seen in Figure 3.1. The first part is the menu bar in the top. It contains a very limited subset of functionality that we are used to see in regular programming IDEs and a Small Basic specific functionality, the Graduate button. The Graduate button can export the code that has been written in Small Basic to Visual Basic so that the developer can keep programming on their project even if they have "graduated" to using Visual Basic instead. The second part is the editor itself, where the code is written. The third part is intelliSense and auto complete. It is works the same as in Visual Studio, it shows what is possible to write with a description of what that functionality does. The fourth part is where the error messages are shown and the fifth part shows the properties of a selected element.

[2] With that being said there was a four year hiatus between version 1.0 and 1.1

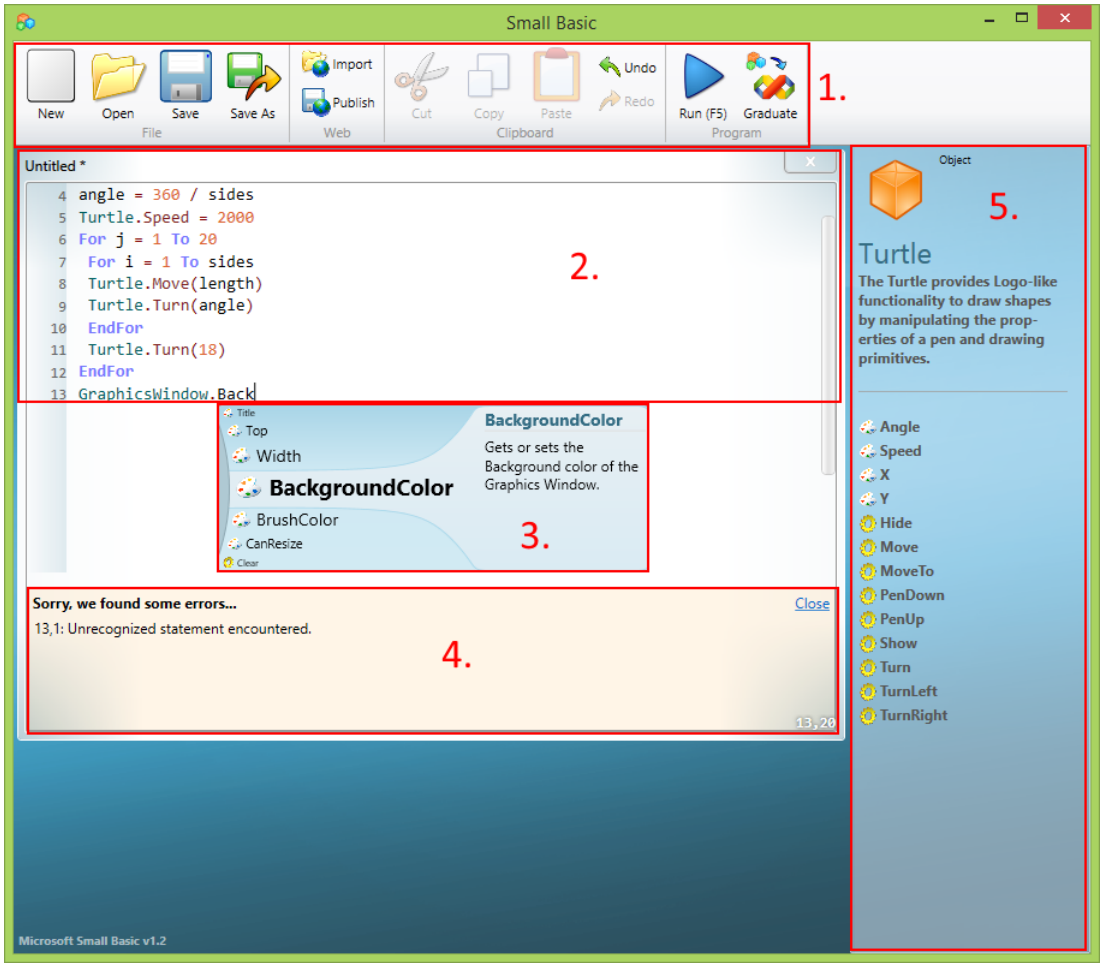

Figure 3.1: Small Basic IDE

3.1.2 General Purpose Programming Languages

This section will provide an overview of some of the general purpose programming languages that has been designed with novices in mind, but still aims to be used as a end-user language. It will also provide an overview of some of the tools used to teach novices to program in languages not specifically designed for novices but as a professional programming language. TODO: We need to discuss terminology further.

Programming Languages

One of the first programming languages designed for novices and to be user friendly is BASIC (Beginner's All-purpose Symbolic Instruction Code). It first appeared in 1964 as a language that would enable students in fields other than science and mathematics to use computers. Since then it became a widely used language and today there are more than 230 different documented dialects of BASIC, among those is Microsoft's Visual Basic.

Another language that was originally designed largely with students in mind was Pascal. It was released in 1970 and aimed to teach students structured programming. However, some early adopters used it far beyond the original intent. This resulted in a lot of work on Pascal and it evolved, both the language itself but also into different dialects, which ended up being designed more towards end-user use rather than an educational tool. Although this meant that the language and dialects became very popular and widely used. The language itself and the dialect called Delphi/Object Pascal is still used today [11].

A third language that was designed and is still maintained as an easy-to-use general purpose programming language is Quorum. Quorum is an evidence-based programming language. This means that it is updated and changed according to current research on how an easy-to-use programming language should be designed. Some of the team members behind the language host an annual workshop called The Experience Programming in Quorum (EPIQ), which is "an international professional development workshop for educators to learn the foundational skills necessary to teach students computer science using the Quorum programming language" [12].

BlueJ

BlueJ is a development environment will allows the creation of Java programs quickly and easily. BlueJ has its roots back in the nineties when Michael Kölling developed a pedagogical language and environment called Blue [13]. Essentially, BlueJ was initially released as a port of Blue to Java in 1999 and its support continues to this day, thanks to Sun Microsystems and Oracle.

First and foremost, the focus of BlueJ is that its primarily designed for teaching, with good pedagogy in mind. Therefore, many of its characteristics are centered around that notion such as simplicity,interactivity, maturity and innovation. BlueJ has a smaller and simpler interface compared to professional programming environments such as NetBeans or Eclipse with the deliberate intention of not to overwhelm beginners. Additionally, it allows a great deal of interaction with objects e.g. inspecting their values, calling methods on them, invocation of Java expressions without compilation etc. Given that BlueJ is fifteen years old with a solid foundation and full-time team working on it, beginners in programming can easily make use of its technical support. Being a well established environment, BlueJ also has several original features not present in other IDEs such as object bench code, code pad and scope colouring.

In order to address better the pedagogical side of BlueJ, the BlueJ development team is constantly looking for ways to improve the learning process of programming and make it easier,simpler and more enjoyable. However, this is often not an easy thing to do since there is no real way to measure how good a design decision is [14]. Having a way to obtain data from its usage will give the team more control over the environment and will benefit the community of BlueJ users as well. Even more, the collected data could be of interest to the wider researcher community and generally people who might be interested in how BlueJ is being used, to much greater benefit. This idea of expanding the amount and type of data collected, while keeping it anonymous, gave fruition to the Blackbox project.

The Blackbox data collection project was announced at the 2012 SIGSE conference in Raleigh,North

Carolina,USA. [14] Initially, with the feedback form attendees, the data collection method was finalised along with some technical details for the whole process. As already mentioned, one of the key features of the project is keeping the data anonymous in order to avoid any ethical and legal complications. The data collection continues to date, as the only condition is to have BlueJ 3.1.0 or newer in order to participate in this research [15].

TODO: Tools for learning
TODO: Add missing sources!

TODO: Remember to summarize text compared to visual programming.
TODO: transition to visual programming
TODO: Why is it the goal to learn these languages? (Look into Intentional Programming)
TODO: Why can't we stick with the educational ones?
TODO: Why don't we begin with these?
TODO: Good thing about text based, is terminology, one learns to talk about programming, e.g. what is a statement?

3.2 Visual-based Programming Languages

Traditionally, most programming languages are categorized as text-based because of the way the program logic is written, by making use of a syntax, specific to every language. Therefore, it is often difficult to learn and use a programming language since it requires one to familiarize oneself with the syntax and available constructs first in order to use the language effectively and that takes skill many people lack.

In order to address the difficulties in learning programming, for the past 25 years TODO: JAIS: Isn't it longer than that? Either way, a cite is needed, research has been done on the so called "Visual Programming" or "Graphical Programming", and dozens of visual-based programming languages have been created. This approach, reserved and used in the past primarily for systems design, allows the use of spatial representations in two or more dimensions in the form of blocks and different structures and shapes. Compared to text-based programming where lines of code are used, graphical programming replaces these with visual objects, essentially replacing the textual representation of language components with a graphical one, more suitable for visual learners and intuitive for people with no prior knowledge in programming. The creation of programs in such languages is defined by placement and connection between visual objects where the syntax is encoded within the objects' shapes. The main aim of visual programming languages (VPL) and environments, as stated by Koitz ans Slany [16], is "*diminishing the syntactical burden and enabling a focus on the semantic aspects of coding*." VPL try to facilitate end-user programming, both kids and adult novice programming, empowering the creation of new programs, not just their consumption, effectively minimising the distance between the cognitive and computational model.

Currently, there is a wide variety of visual programming languages with varying popularity such as Alice, Greenfoot, Tynker, Scratch, Raptor and many more. From these, Scratch and Alice will be described and analyzed further.

3.2.1 Scratch

Scratch is a visual-based programming environment which allows users to create visually-rich, interactive projects. Since its inception in 2003, the main goal of its creators has been to address the needs and interests of young people (primarily ages 8 to 16) and make a soft introduction to the world of programming for them. Publicly released in 2007, the project has grown in size and scope, with a dedicated site hosting all its 11 million projects and with a user base of 8 million [17]. Given its targeted audience, one of the main design goals of Scratch is the focus on self-directed learning and exploration through tinkering with the different constructs of the language and environment. This combined with the steady increase of its popularity has prompted hundreds of schools and educational organizations to adopt and integrate it into their curriculum [1].

What makes Scratch a sensible choice for people with no prior programming experience is the fact that it has less emphasis on direct instruction than other programming languages. Instead, it focuses on the aspect of learning through self exploration and peer sharing, which breaks the norm of a traditional educational approach.

3.3 Comparison

TODO: A general comparison on text-based and visual programming languages. Not on the specific languages chosen, but on the concepts and their fit for novices and teaching.

Chapter 4

Teachers and Programming

Around the world, learning computational thinking is starting to appear in the school curriculum for students in the age range of 5-16 years old, depending on the country. Because of this it is relevant to study which resources are available to the teachers who are going to teach the children. In this chapter we will describe the current state of the teachers and discuss possible problems regarding that state. The chapter will focus mainly on Denmark and the United Kingdom.

4.1 Current State

As of 2015 the danish government has added programming to the school curriculum, although this might be a wrong way of phrasing it as, according to the Danish Learning Portal[1] it is not only programming that should be taught but computational thinking in general. An example of this can be seen in this quote on how to incorporate "programming" into the mathematics course[2]:

> Programming activities can support that the students work with algorithms, meant as systematic descriptions of issues, solution strategies and events. A recipe is a good example of an algorithm. (1) Mix the dry ingredients together, (2) stir. (3) Add 2/3 of the water and stir. (4) If the dough is smooth, stir for 2 minutes. Else go to step (3) and add more water. Algorithmic thinking is about setting up and making machines execute such algorithms... [**?**]

The problem with this addition to the curriculum is that the danish government expect the individual teacher to teach themselves the subject, to then teach the students. The government has allocated $1 * 10^9$ Danish Crowns to educate current teachers in the new subjects added to the curriculum for 2015, but programming is only one subject among many. This means that it is uncertain how much of it goes to educating teachers in the ability to teach programming. One of the things the government has suggested is the programming environments which can be used, here among "Scratch", "Tynker" and websites like

[1] http://www.emu.dk

[2] This is translated from danish, so the wording might be different in the original text, but the meaning is the same

Code.org.

Another part that is uncertain is how the educations regarding teaching will implement the new curriculum. It is the individual educational institution's job to make certain that the teachers who graduate are equipped to the goals of the curriculum [**?**].

It is a different story regarding the U.K. They implemented programming and computational thinking in the curriculum in 2013, which took effect September 2014 [**?**]. To prepare the teachers for this they allocated $1.1 * 10^6$ British Pounds in funding specifically to train school teachers who are new to teaching computing [**?**]. This was announced in December 2013 and in February 2014 another $500,000$ British Pounds was allocated in funding to attract businesses to help train teachers [**?**].

4.2 Discussion

The United Kingdoms seem to have a good grasp on how to educate teachers with no knowledge about programming. Denmark on the other hand seems to struggle which can possibly lead to problems in the future. To start with, if the teachers are not experienced enough or not enthusiastic about the subject, as it, in the case of Denmark, is the physics/chemistry teacher who mainly has to teach programming, can possibly end up demotivating students from programming. Luckily governments suggests languages such as Scratch to use in lessons, which has a good range of online tutorials and self learning material.

Another possible problem is that if the students are taught in a way that gives them a wrong understanding of aspects in programming or makes them develop bad habits, they can struggle with those problems later in their education, given that they choose an education that is programming related[3]. Given these possible problems, it might be interesting to make research in this area to determine if these end up being actual problems.

[3]These problems are speculative and anecdotal and have no roots in the literature.

Part II

Language Comparison

Chapter 5

Preliminaries

To get a better idea of the landscape of educational programming languages, we are going to compare three popular educational languages. The three languages are: Scratch, BlueJ and Dr. Racket. These three languages are chosen as the most popular languages of each their paradigms TODO: probably need to find a source for such a claim. Scratch represents an imperative paradigm, BlueJ an object-oriented paradigm and Dr. Racket a functional paradigm. To do this we will first define some criteria to compare the languages on, after which we will subjectively analyse the languages to get a comparison.

Chapter 6

Criteria

TODO: new introduction to this chapter To measure the differences between languages with focus on novice learning, a set of criteria have been set up. These criteria are the base from the evaluation, and are a mixture of measurable and subjective criteria.

This chapter will present the criteria for the evaluation, the reason for the criteria being chosen, and how to measure them. Several criteria have been considered, where most are taken from known criteria for comparison [18] [19], and others are found through discussion of the research questions.

Readability

Readability is the expressibility of the language. Readable code gives a greater understanding of the semantics as well of the nature of the code. For this project, readability is a vital criteria, as interest often follows understanding. Since high readability produces greater understanding, readability is chosen as a criteria for comparison. Readability is hard to measure, as it is a subjective matter, mostly depending on the person writing code. The measurement of this quality will therefore based on using code conventions and skilful coding.

Writability

Writability is the ability to translate thoughts into code. It describes the expressivity of the code and the ease of writing, in a combination of quality and quantity.Writability can be measured in the level of abstraction. This can be done through lines of code as well as looking into the different language constructs that support abstraction.

Observability

Observability is the level of feedback gained for a better understanding of the input. It is to what extend you can observe reactions to what you make. To measure the observability is lo look at this level of feedback.

Trialability

The level of possibility of trial and error through coding is measured through trialability. This

feature is measured by the level of feedback when an error occurs, how often feedback is given, and a discussion on how easily a novice programmer can recover from a mistake.

Learnability

As a language is learned, there are helping and hindering factors. These are measured through learnability, which is done by measuring the cost of learning the language and its environment.

Reusability

The level of possibility for reusing code, through abstraction. This is measured both in quantity and quality, depending on the abstraction level and layer depth.

Pedagogic Value

A programming language in itself can be easy to learn, but if it doesn't help the programmer in learning the basic concepts of programming, then there is no pedagogic value. This means the programming language should support the general programming concepts that are typical for common languages and coding conventions.

Environment

The development environment plays an important part for novices, and it should provide help for the programmer in a simple, clear and manageable way. The evaluation of the environment will be done according to how interactive systems are evaluated by David Benyon [20, p. 225-250].

Documentation

The amount of documentation, as well as the informative value of this, is important for a novice, as a help for solving problems they cannot solve themselves.

Security

The level at which violations of definitions are caught by the compiler. TODO: do we need this one?

Uniformity

The consistency of appearance and behavior of language constructs. If the code doesn't look like any conventional language, the programmer will not learn the general approach to programming, and will have trouble moving on from this language. This is measured in terms of constructs being similar to known syntaxes, or preferably written in the exact same way.

Chapter 7

Comparative Analysis

TODO: The setup for the comparison as well as the actual comparison.

Chapter 8

Usability Evaluation

TODO: Do the experiment, then write setup and results here.

Chapter 9

Results

TODO: Results and discussion.

Part III

Bibliography

Bibliography

[1] R. N. S. B. E. E. Maloney John, Resnick Mitchel, "The scratch programming language and environment," *Journal Name*, vol. 10, no. 16, 2010. 1, 3.2.1

[2] C. Schulte and J. Bennedsen, "What do teachers teach in introductory programming," *ICER '06 Proceedings of the second international workshop on Computing education research*, pp. 17–28, 2006. 1

[3] S. Garner, P. Haden, and A. Robins, "My program is correct but it doesn't run: A preliminary investigation of novice programmers' problems," *ACE '05 Proceedings of the 7th Australasian conference on Computing*, vol. 42, 2005. 2.1, 2.3

[4] S. Xinogalos, "Object-oriented design and programming: An investigation of novices' conceptions on objects and classes," *ACM Transactions on Computing Education*, vol. 15, no. 3, 2015. 2.3

[5] S. Papert, *Mindstorms: Children, Computers, And Powerful Ideas*, p. 218. Basic Books, Inc., 1 ed., 1980. 3.1.1

[6] S. Papert, *Mindstorms: Children, Computers, And Powerful Ideas*, pp. 11–15. Basic Books, Inc., 1 ed., 1980. 3.1.1

[7] A. Kay, "Alan kay shares a powerful idea about teaching ideas." `https://www.youtube.com/watch?v=JDpsXWuedVc`, 2014. Used: 26/09/15. 3.1.1

[8] Microsoft, "Small basic faq." `http://smallbasic.com/faq.aspx`. Used: 26/09/15. 3.1.1

[9] Microsoft, "Microsoft small basic, an introduction to programming." `http://download.microsoft.com/download/9/0/6/90616372-C4BF-4628-BC82-BD709635220D/Introducing%20Small%20Basic.pdf`. Used: 26/09/15. 3.1.1

[10] Microsoft, "Small basic curriculum: Online." `http://social.technet.microsoft.com/wiki/contents/articles/16982.small-basic-curriculum-online.aspx`. Used: 26/09/15. 3.1.1

[11] T. Software, "Tiobe index for october 2015." `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`. Used: 26/09/15. 3.1.2

[12] Qourum, "Epiq 2016." `http://quorumlanguage.com/epiq.php`. Used: 26/09/15. 3.1.2

[13] M. Kölling, "Bluej features overview." http://www.bluej.org/about.html. Used: 01/11/15. 3.1.2

[14] M. Kölling, "Bluej blackbox data collection project." http://www.bluej.org/blackbox.html. Used: 01/11/15. 3.1.2

[15] B. team, "Bluej blackbox data collection for researchers." https://blackboxdc.wordpress.com/. Used: 01/11/15. 3.1.2

[16] S. W. Koitz Roxane, "Empirical comparison of visual to hybrid formula manipulation in educational programming languages for teenagers," *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, pp. 21–30, 2014. 3.2

[17] MIT Media lab, "Scratch statistics." https://scratch.mit.edu/statistics/. Used: 28/09/15. 3.2.1

[18] L. Rhodes, "Design criteria for programming languages." http://jcsites.juniata.edu/faculty/rhodes/lt/plcriteria.htm, 2015. Used: 23/11/15. 6

[19] University of Washington, "Evaluating programming languages." http://courses.cs.washington.edu/courses/cse341/02sp/concepts/evaluating-languages.html, 2015. Used: 23/11/15. 6

[20] D. Benyon, *Designing Interactive Systems*. Addison Wesley, 2 ed., 2010. 6