Comparative Analysis of Educational Programming Languages and Environments

Group DPT906E15 - Room 5.1.50

1 September - 8 January

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

Title:

Comparative Analysis of Educational Programming Languages and Environments

Project period:

1 September - 8 January

Project group:

DPT906E15

Participants:

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

Supervisor:

Bent Thomsen

Pages: 62

Appendices: 0

Copies: 2

Finished: 8 January 2016

Abstract:

Computational thinking is becoming a skill needed more and more in everyday life. This leads to several countries introducing children to programming in primary school. However, it is questionable whether the tools and methods used are adequate for the task. We investigate the area of educational tools and environments to understand their capabilities in educating novices. We have described several languages and environments, and analysed a set of these, being Scratch, BlueJ and DrRacket. The analysis was based on a set of subjectively defined criteria, and was followed by a comparison of the tools. We concluded Scratch to be the tool best fitting for novices, as it is easily learned and gives a great understanding of the basics in computational thinking.

Preface

The following report was written by Henrik Vinther Geertsen, Jais Morten Brohus Christiansen, Svetomir Kurtev and Tommy Aagaard Christensen in accordance with the conclusion of the ninth 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. Additionally, we would like to thank the Coding Pirates and Magnus Toftdal Lund for the opportunity to observe their introductory programming course.

Contents

Chapter 1

Introduction

Traditionally, programming has been seen as a specialized skill, only relevant for people who work on developing new software. While several attempts have been made at making languages more accessible to everyone [1, 2], 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 general purpose programming languages like Java [3]. 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 [4] for their intuitiveness and simplicity. This means that different levels of education differ in what they teach. First, there is a difference in programming languages, as to between being taught an educational language and a general purpose language. While the educational language has the advantage of being intuitive, it does not have the expressiveness and robustness for large projects that general purpose 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?" [5] 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 understand the educational value of teaching novice programmers programming in primary school. To do this, we first need to understand how and what novices are taught. The report therefore aims to understand the areas where novices in primary school have problems as well as the tools used for teaching

programming.

1.1 Initial Questions

Today, more and more countries are implementing computational thinking as a part of the primary school curriculum. We want to get an understanding of educational programming. To do this, we want to analyse the literature and tools used for this. The difference in languages and environments must also be addressed. As there is a big difference in visual and text-based programming, the difference between these must be analysed. Different paradigms in programming also have different approaches to problem solving. This might lead to the fact that novices have to learn aspects that help them code in the specific paradigm. These factors must be explored as well.

Through this, we have the following initial questions:

- Which tools are being used by novices regarding their first contact with programming?
    - How can these tools be compared?
    - What do the tools lack?
    - Which programming paradigms are being taught?
        * What are the benefits of each paradigm?
- What are the difficulties in learning programming?
- What are the differences of existing educational programming languages?
- Is it difficult to go from visual programming to text-based programming?

1.2 Report Structure

In the first part of the report we analyse the area as a whole. We first present some related work in Chapter 2. Next, in Chapter 3, we discuss the various problems that novices are faced with when learning to program. We then present different educational languages and environments, and make an initial comparison between visual and textual languages in Chapter 4. In Chapter 5 we analyse what is done to implement teaching into the school curriculum in a couple of different countries.

In the second part of the report, we give a language comparison, done as a subjective evaluation through discussion by the authors. We state the criteria evaluated in the comparison in Chapter 6. We analyse the chosen languages in Chapter 7. The comparative analysis itself is provided in Chapter 8. The results of the analysis is then discussed in Chapter 9.

In the final part of the report, we have a conclusion in Chapter 10, and a discussion of the project and ideas for further works in Chapter 11.

Part I

Problem Analysis

Chapter 2

Related Work

The area of educational programming encompasses a somewhat big area of research, with subareas focusing on children (ages 8-16), CS1 courses at university, teaching programming and the transition from novice programming to "real" programming. This chapter explores some of the research in these areas.

S. Papert wrote a book in 1980 called "Mindstorms: Children, Computers, And Powerful Ideas" [6]. He stated in a note that he wanted to demonstrate the fact that computational thinking could be taught to the pupils in the earlier grades of school [7]. This shows that the thought of bringing programming, or specifically computational thinking, to children in middle school or earlier, has been around for some time. One of the more recent papers on this subject, by Meerbaum-Salant et al. 2013 [8], proposes and evaluates learning material for using Scratch specifically aimed at teachers of middle-school students. Scratch seems like a good choice for this as it is one of the proposed languages to use for the curriculum at least in regards to the UK and Denmark [9] [10]. The paper concludes that Scratch is a viable platform for teaching CS, but that it should be in combination with close and effective mentoring. Another way of learning e.g. Scratch is through online resources, but according to Lee et al. 2013 [11], many of these resources struggle to keep the users engaged. They propose a way of improving this through assessments integrated into the learning environment. Research is done in both directions of self-directed learning and learning in traditional classes, but focusing on self-directed learning might become less important over time if more and more countries start to add computational thinking to their curriculum.

Even though Scratch is recommended for teaching in middle school, it has elements which makes it cumbersome to work with if the idea is to keep it as part of e.g. math [10] as it is not really well suited for more complex formulas. This is shown by Koitz & Slany, 2014 [12], who have made a VPL using blocks for coding similar to Scratch but for phones, called PocketCode. One of the major differences they focus on is the fact that PocketCode uses text input for formula manipulation instead of constructing them entirely from blocks. Their reasoning for this is that more complex formulas can get quite big and unmanageable in Scratch, and that it has a problem with how it handles nested blocks. The problem lies in the fact that if a block containing nested blocks is deleted, then the whole thing is deleted. This makes it harder to edit a formula if a user makes a mistake compared to a text-based approach.

Meerbaum-Salant et al. 2015 [13], a follow up to their earlier mentioned work, have also researched the transition from novice programming to "real" programming. Their research is based around five classes who had a course, where some participants had no programming experience and some had spent a year with Scratch in an earlier grade. They found the participants who had learned Scratch were more motivated and could relate most of what they had experienced in Scratch to the text-based language used. Although they had some wrong impressions about how some constructs worked, they were quickly overcome. They concluded that there were not a significant difference in the grades between the participants who knew Scratch and those who did not, but that knowing Scratch improves the learning of difficult concepts.

Matsuzawa et al. [14] presents a more recent paper approaching the problem of migrating from a visual programming language to a text-based programming language. They have developed a tool called *BlockEditor*, where one can make use of both a block-based and a text-based approach to developing software. They made an empirical study, based on about 100 student and the tool, to see if the students would gradually use the text-based approach more than the block-based one as time went on. The study was conducted over the course of 15 weeks and focused on art students instead computer science students. They also made the students self-evaluate their own programming skill level through the course. They found that about 10 % of the students rarely, if never, used the block approach, where as another 10 % used it nearly for the entirety of the course. The remaining 80 % relied on a block-based approach and gradually changed to mostly using a text-based approach. They also found that most of the students who did not use the block-based approach was the students who had rated their own skill level the highest among other student. Similarly the students who rated their skill level low, was the ones that relied on a block-based approach for the longest amount of time.

In the area of creating text-based programming languages, Stefik & Siebert, 2013 [15] have developed a language called Quorum. They call it an evidence-based programming language, and the goal is to continually make the syntax and keywords fit what the evidence of current literature suggests. They do this because, claims regarding design decisions in programming languages have often not been backed by scientific evidence in the past [16]. They have conducted a couple of studies to get an insight into which words and symbols novices find intuitive/not intuitive. Additionally they have conducted empirical studies comparing Quorum to other programming languages with focus on educational purposes. In the comparison they have also used a principle from medicinal science, where they have tried to use a "placebo" language together with the other languages chosen. They call it Randomo and it is a somewhat randomly generated language where keywords and symbols are chosen at random from the ASCII table.

The widespread use of electronic devices and media makes the demand for people with digital literacy and skills grow every day. However, there is a trend of school computing lessons on the decline and not enough appropriately qualified teachers to accommodate that demand, as stated by Bundy & Scott, 2015 [17]. In order to tackle the problem, a worldwide movement has been formed in the UK by the Computing at Schools organization (CAS) [18]. It is considered not sufficient to teach children only to program, but also to think computationally and to use abstraction and modularity for understanding and

solving problems. Scotland's new Curriculum of Excellence (CfE) provides more freedom for teachers to decide on teaching materials and methods and CS is a core entitlement for students during the first three years of secondary school.

In 2010, with the help of government bodies, the Royal Society of Edinburgh (RSE) and the BCS Academy of Computing, Scotland successfully developed a program that combines programming, computational thinking (CT), and evidence-based pedagogy in order to address the computing problem. Due to its success, it was adopted by many other countries. Phase 1 of the program focused on the first three years of secondary school (11-14 in Scotland) relying on a more modernistic approach given the fact that current generations are online,social and very mobile. In terms of programming environments, Scratch and AppInventor were selected because of the way they shift the focus from figuring out the syntax to computational thinking. These environments support the creation of multimedia applications and smartphone apps. Phase 2 of the program focused more on the transition from block-based languages to more traditional text-based ones. The choice was on LiveCode, allowing cross-platform development.

Chapter 3

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

3.1 Syntax and Semantics

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 [19]. This consists of brackets, semicolons, commas, and other such symbols, representing 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 might lead to a compiler error. This error margin is not seen very often when learning other skills, and might discourage the novices from continuing to try.

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

3.2 Pragmatics

A programming language is built on syntax and semantics. To learn how this works can be effective in programming, but programmers often do not 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 programmers instead 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 of using pragmatics is hard to find and to measure, which leads to no empirical work within this area that can be used for a further evaluation. Although, it should be taken into account that teaching pragmatics might be much more useful than teaching syntax and semantics.

3.3 Programming Paradigms

Different paradigms each have their different difficulties. Some programmers first touch programming through an imperative approach. Others start out with an object oriented programming language. Some have their first experiences programming in a functional programming language.

Imperative programming has its values in its very straightforward 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, which makes imperative programming a valid learning approach.

Object oriented programming (OOP) has its 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. In relation to imperative programming OOP is often seen as not being something else, but the same, only with object oriented features [19]. This leads to a problem of both understanding the very basic concepts of programming, such as control structures (loops and selections), along with understanding the object oriented approach. The Object-first approach tries to avoid this problem, by teaching OOP before imperative programming.

Functional programming (FP) is a paradigm which uses lambda 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 [20]. Additionally, many concepts of the imperative approach can be simulated in Functional programming(e.g. looping control structures are expressed in terms of recursion), giving as much expressivity 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 [21].

It is a wide discussion to determine what paradigm is the most efficient when teaching. The necessity of learning concepts of OOP before learning to code can be argued, as well as it can be argued that the basic constructs are necessary before learning about different paradigms and advanced structures. The imperative approach is being taught in primary school in various countries [10]. 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 [22]. One teaching method that implements this is 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 through this approach, or the classic "algorithms-first approach" [23]. One drawback is the fact that the concepts of OOP might be seen by the novices as the basics of 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 4

Programming Languages and Tools

With the growing distribution of computers and mobile devices, e.g. laptops, smart phones and tablets, the need for 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.

4.1 Text-Based Programming Languages

Text-based programming languages have two distinct subcategories. One is the text-based educational programming languages for novices, specifically designed to teach novices for then to be replaced by a more feature rich language. The other is the general purpose programming languages. Even 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.

4.1.1 Text-Based Educational Programming Languages

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

Smalltalk

Smalltalk is considered an important programming language which introduced Object oriented programming and its fundamental concepts. Originally developed by Xerox Corporation's Palo Alto Research Center (PARC) in the 1970's, it was a 10 years project, culminating in 1980 as the Smalltalk-80 system [24]. One of the key members responsible for that project was Alan Kay who helped develop prototypes of networked workstations, later commercialized by Apple Computer. He also played a pivotal role in the educational scene. He was the inventor of the Dynabook concept which defined conceptually the basics for laptops and tablet computers and was conceited primarily as an educational platform [25]. Some basic programming concepts, used today by popular languages such as Java or C++, were originally conceived in Smalltalk. It is worth to mention that Smalltalk brought the world a number of innovations which have fundamentally affected today's computer landscape such as:

- bit-mapped graphics displays

- windowing systems

- mouse-driven menus

After its immediate release, the idea of object-oriented programming was perceived as a major revolution in programming languages. This lead to the consequence that many popular languages at that time adopted that idea of object orientation,such as Common LISP Object System, Object- Pascal, Objective-C and Object-COBOL. Due to some licensing restrictions however, Smalltalk did not become widely available to the programming community, but rather for the research and academic circles. This was later changed with the introduction of the Squeak implementation of Smalltalk which spread quickly and was completely free.

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 relatively easy programming language for novices to learn, 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 [7].

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

---

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

- 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 colour, 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 [1].

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 has 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 [26].

Small Basic

Small Basic is a text-based programming language with its whole purpose being teaching novices to program. 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 [2]. It is perhaps one of the last text-based 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 parts: 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 [2], 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, as 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.

---

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

To learn programming with Small Basic, its website provides a tutorial for getting familiar with the language and the IDE [27], and a curriculum. The curriculum can be downloaded for offline use, and teaches general programming topics using Small Basic [28]. 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. It is therefore 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 4.1. 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 works the same as in Visual Studio, as it shows what is possible to write with a description of the functionality. The fourth part is where the error messages are shown and the fifth part shows the properties of a selected element.
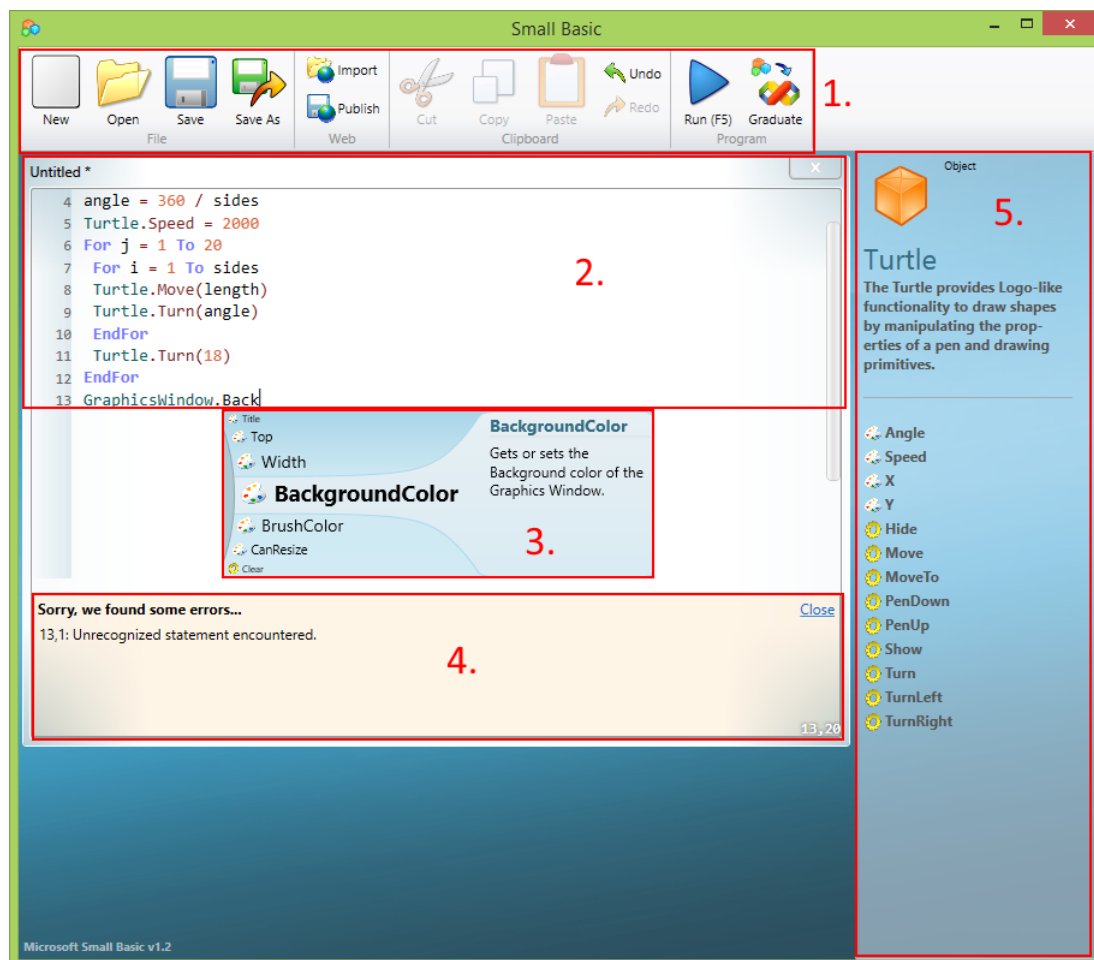

Figure 4.1: Small Basic IDE

13

4.1.2 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 general purpose programming language.

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. The language 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 [3].

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" [29].

BlueJ

BlueJ is a development environment which 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 [30]. 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, BlueJ is 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 avoiding novices being overwhelmed.

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, double clicking classes to explore their contents etc. The general interface of BlueJ can be seen on Figure Figure 4.2 where the left pane allows the creation of classes and connections between them. The Main Panel takes the majority of the space on the interface and it serves as an overview of all the classes and the established hierarchy between them. The bottom pane serves the purpose of storing the instances of objects.

Given that BlueJ is fifteen years old with a solid foundation and full-time team working on it, novices 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.
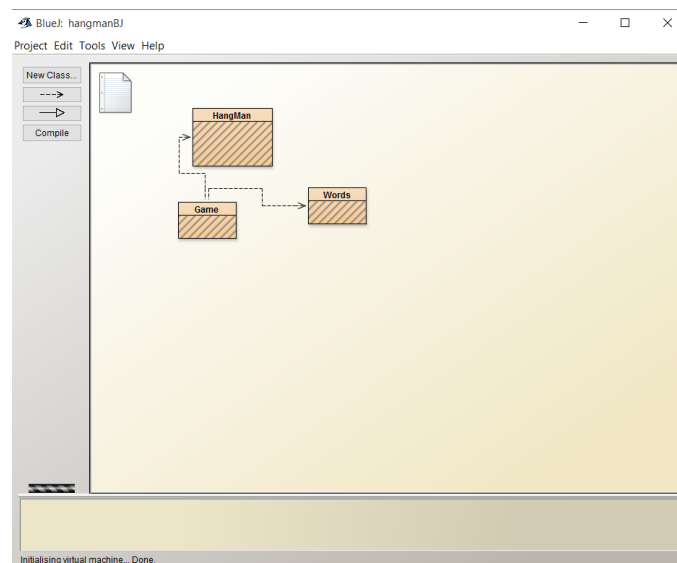

Figure 4.2: BlueJ IDE

In order to better address 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 [31]. 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, such 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, gave fruition to the Blackbox project.

The Blackbox data collection project was announced at the 2012 SIGSE conference in Raleigh, North Carolina, USA [31]. Initially, with the feedback from attendees, the data collection method was finalised along with some technical details for the whole process. 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 [32].

15

Dr. Racket

Dr. Racket is and IDE for the language Racket, which is a functional programming language and has its roots in Lisp and Scheme, but with added features. The IDE is created for novices that are learning the Racket language, and programming in general, in combination with the book "How to Design Programs"[3]. There are also tutorials for getting started on their website and the language is fully documented, which can also be found on their website[4].

Even though the IDE is for novices, it still has features similar to a fully fledged IDE, and the assumption for having those are so that the novices can stay with the IDE even when they have finished learning the basic elements. Most of these features are hidden away in the menu and a novice is only required to start the IDE to get started with the programming. Some of the features for novices can be seen in Figure 4.3, where one can see the editor window (upper part of the window) and the interactive window (lower part of the window). Code can be written in the editor and then run using the "Run" button in the upper right corner. Code can also be written in the interactive window, where it will be executed immediately after the completion of an expression. After the execution the value that the expression has been evaluated to is written as well, so a novice can always follow what they are doing. The two windows work together, so if a user has written a definition in the editor window and presses run, that definition is available in the interactive window.

Another feature for novices can be seen in Figure 4.4, in which the "Choose Language" window is shown. In this window the user can select which language should be supported by the IDE, e.g. subsets of the Racket language. This feature is there so novices can choose e.g. a subset of racket which removes access to advanced features, giving better focus on what is relevant. An example of this is if the "Beginning Student" is chosen, procedures must take at least one argument. This limitation is there because procedures in the languages have no side-effects meaning that procedures without arguments are not useful, so it can help avoiding confusing syntactic mistakes [33][5]. So in a sense it makes sure that the novice follows the programming guide lines for the language and does not end up with a work-around more appropriate for an advanced user.

---

[3]It is a free book and can be found here: http://www.htdp.org/

[4]http://docs.racket-lang.org/

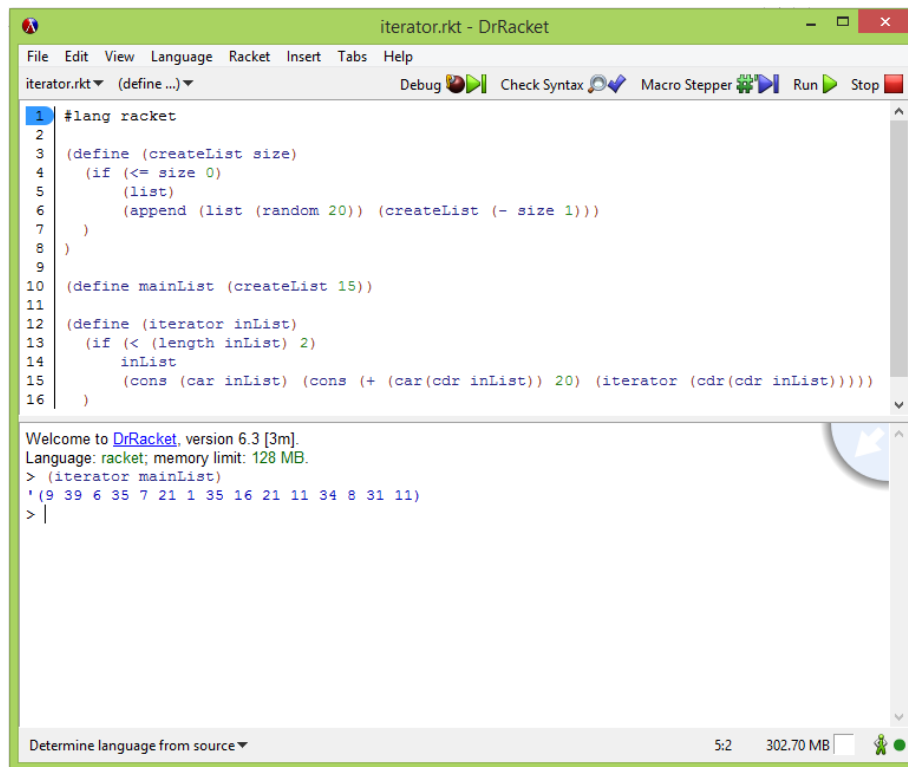[5]A list of the features modified by choosing the different languages can be found here: [33]
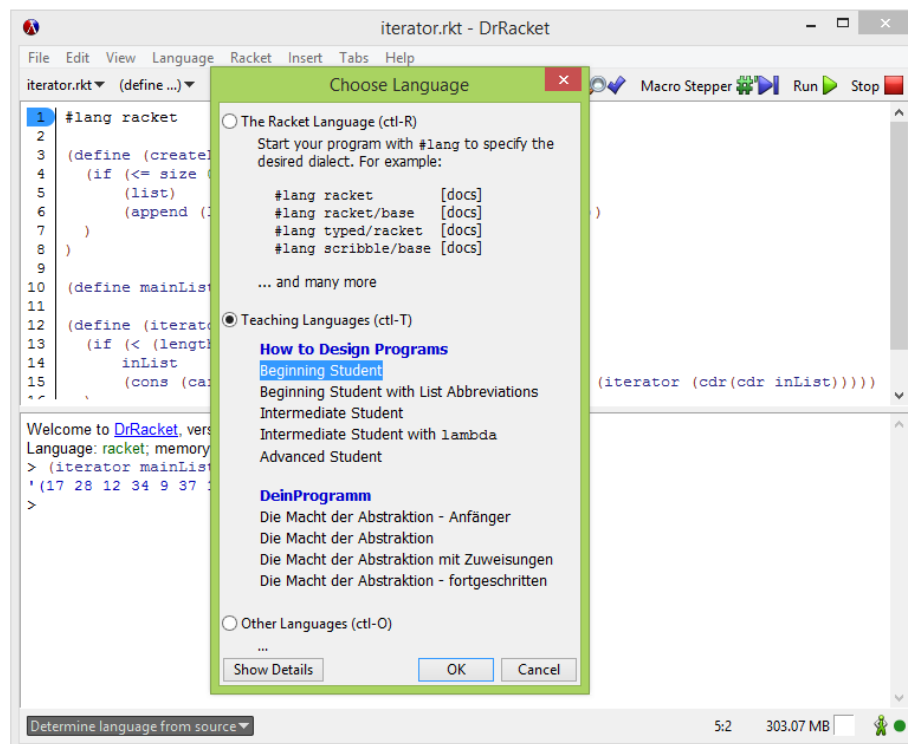
Figure 4.3: Dr. Racket IDE

Figure 4.4: Dr. Racket, Choose Language Window

4.2   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.

In order to address the difficulties in learning programming, for the past 50 years [34], research has been done on "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. This essentially replaces 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 and Slany [12], 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, Alice and Greenfoot will be described further.

4.2.1   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 [35]. 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 [4].

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. Figure 4.5 depicts the overall visual interface of Scratch.

Figure 4.5: Scratch's visual interface[6]

The game window is seen to the left, where the sprites are shown and the room they are in is visualised. In the middle the categories of code blocks are shown. These code blocks are the dragged into the right side of the screen, where the code is represented, to create the code.

There exist other similar programming environments to Scratch, namely Greenfoot and Alice which also have the same purpose of introducing programming to people without prior experience. Therefore, all three systems share the same design goals [36]. They tend to support various graphics and sounds which facilitate the building of programs. One key difference lies in the target group each platform is addressing. Both Alice and Greenfoot target older students than Scratch, while they emphasize on Java and its concepts by introducing class-based and object-oriented programming. Given this fact, Scratch is often used as an introduction to both Alice and Greenfoot. There is an extended version of Scratch, called Snap, that also adds object-oriented programming features, but it is not as commonly used.

4.2.2 Alice

Alice is a 3D interactive animation environment which allows novice programmers to create animated 3D movies in order to get a better understanding of object-oriented programming concepts. Similarly to Scratch, it provides a drag-and-drop functionality in order to prevent syntactical errors which are prevalent for beginners. Working with such a 3D graphics environment is also highly motivating given more powerful graphical capabilities. Additionally, The 3D modeled classes and objects provide a concrete notion of the concept of objects and support an "object-first" approach. Alice was designed

---

[6]https://llk.media.mit.edu/files/content/projects/scratch-editor.png

originally as a tool to improve undergraduate student's ability to succeed in CS1, while currently it is being adopted by hundreds of secondary schools and colleges for pre-CS1, CS1 and non-majors courses [37]. Alice's visual interface can be seen in Figure 4.6
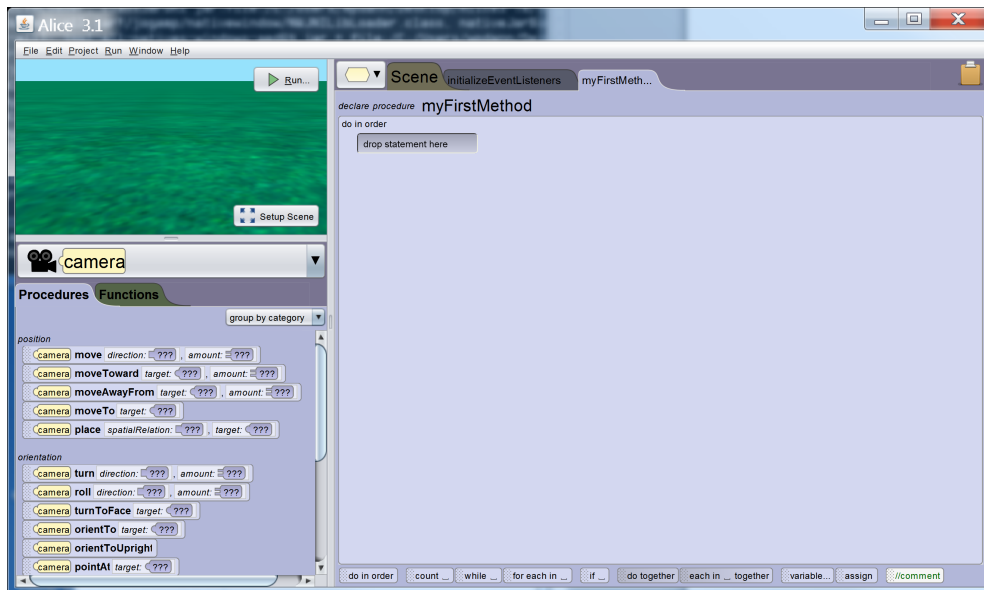

Figure 4.6: Alice's visual interface[7]

The 3D world view is seen in the top left corner. Under it is the blocks that can be dragged over to the canvas on the right to create code.

4.2.3 Greenfoot

As already mentioned, Greenfoot is a graphical environment which emphasises on the educational aspect of programming with Java. It gives a toolkit for developing engaging and interesting programs while using and learning more about object-orientation concepts. Transitioning from Scratch to Greenfoot is very smooth since the two platforms share similar features, one of which is the direct mapping from Scratch's sprites and stage map to Greenfoot's actors and world. [4]. The visual interface of Greenfoot can be seen on Figure 4.7.

---

[7]http://alice3.pbworks.com/f/1381411640/Alice3_Display_Spanish-image-4.png
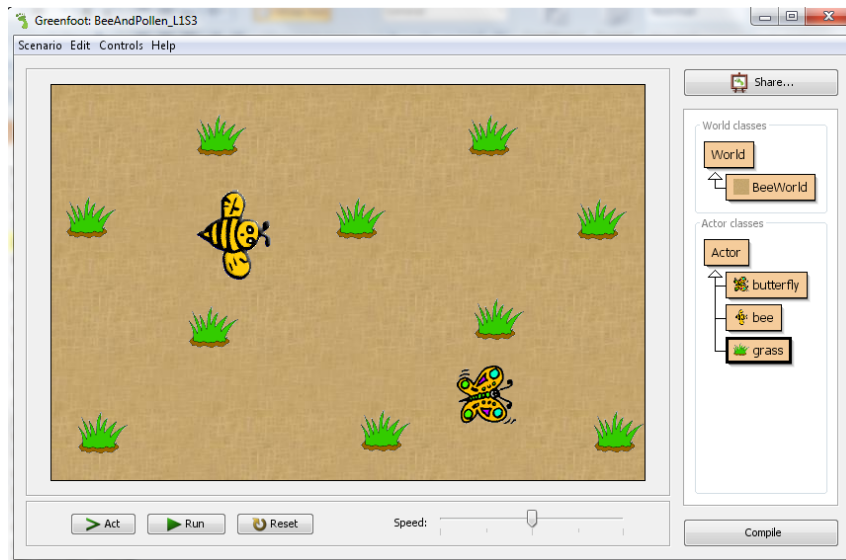
Figure 4.7: Greenfoot's visual interface[8]

The big window shows the game window where the objects are represented, and to the right of it is the class hierarchy for the objects.

4.3 Comparison

As shown in this chapter both text- and visual-based programming languages have parts of their domain dedicated to learning programming. This is done mostly through tools, be it environments for general purpose programming languages, or visual programming languages where the environment can be seen as a part of the language. Both approaches has pros and cons, and these will be explored in this section.

Interface Layout

One of the apparent problems with visual programming environments is the fact that a lot of them has multiple elements, e.g. code, available blocks, etc., where each of them requires a slice of the available screen space. This means that a lot more thought should be put into the interface design, where as environments for text-based programming languages can put less focus on it.

Statement Categories

One of the pros of visual programming environments is the fact that they often require a way of dragging e.g. blocks to create functionality. This means that all of the available blocks, or commands in a text-based programming language, is always shown and available to the user (it might be split into categories), which gives the user an overview of the possibilities in the language. Text based programming languages lacks in this aspect, as the user often is presented with an empty editor or some tutorial code, which still does not tell the user much about the possibilities. Small

---

[8]http://www.oracle.com/webfolder/technetwork/tutorials/OracleAcademy/GreenfootSelfStudyV1/images/L1S3_GreenfootIDE.png

Basic is an example where the possibilities are shown through its auto completion, but the user is just presented with a long list and not all of the commands are descriptive in their naming, possibly making it hard to get an overview for a novice.

Writing Speed

One of the pros of text-based programming languages is the fact that it is possible to produce code faster compared to visual programming environments. Visual programming environments are often dependent on moving code blocks using the mouse. If a user wants to make changes to e.g. a function made up of multiple blocks, they have to pull it apart to get to the block they wish to exchange. In text-based environments the user has the possibility of using both the keyboard and the mouse when they want to select something, giving the possibility of using the preferred peripheral of the user.

From this it can be concluded that visual based programming is better for novices, since the statement categories help overcome the initial writing block, which novices will run into, while the strength of writing speed is not important for them. Similarly text-based programming is better for experts as the writing speed can improve their productivity, while they do not have any problems knowing what they can do.

Chapter 5

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 teaching situation and discuss possible problems regarding that state. The chapter will focus mainly on Denmark and the United Kingdom.

5.1   Current State

As of 2015 the danish government has added programming to the school curriculum. However, 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 by EMU on how to incorporate "programming" into the mathematics course[2]:

> Programming activities can support the students working 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... [10]

The problem with this addition to the curriculum lies in the Danish government expecting the individual teacher to teach themselves the subject, to then teach the students. The government has allocated one billion 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 [38].

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 [39]. To prepare the teachers for this they allocated 1.1 million British Pounds in funding specifically to train school teachers who are new to teaching computing [40]. 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 [41].

In the USA they also recognized the need for programming in the schools in 2013 [42], but were not willing to spend the money on educating the teachers like in the U.K. Unlike Denmark however they simply chose not to make programming mandatory yet, but instead focus on preparing the educational system in smaller steps. They have first focused on making the optional courses in high school programming more attractive by making them count toward the graduation requirements [43]. Also recently there has been research done in facilitating the principles of computational thinking in non-programming courses [44].

5.2   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. If the teachers are not experienced enough or not enthusiastic about the subject, it can possibly end up demotivating students from programming. In Denmark, it is the physics/chemistry teacher who mainly has to teach programming. Luckily, governments suggest 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 6

Criteria

Before a language analysis can take place, a set of criteria are needed. These criteria represent the foundation for the evaluation, and are chosen from a mixture of subjective viewpoints.

This chapter will present the criteria for the evaluation. Several criteria have been considered, where most are taken from known criteria for comparison [45] [46], and others are found through discussion of the research questions.

Readability

Readability is how easy code is to read and understand. Understanding can both be in sense of understanding what a specific procedure does but also how the procedure fits in with the rest of the code. Readable code gives a greater understanding of the semantics as well as the nature of the code. For this project, readability is a vital criteria, as interest often follows understanding. Readability is hard to measure, as it is a subjective matter, mostly depending on the person writing code.

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 is often found in the level of abstraction, and in the simplicity of the syntax.

Observability

Observability is the level of feedback gained for a better understanding of how the code affects the project. It is to what extend one can observe how changes affect the result. It is found in the level of feedback, and the informative value it provides.

Trialability

Trialability is the level of possibility for trial and error through coding. This feature is seen in the level of feedback when an error occurs, how often feedback is given, and how easily a programmer can recover from a mistake.

Learnability

Learnability is the ease of learning the language. It defines how long it takes to learn the language and be proficient in its use.

Reusability

The level of possibility for reusing code, through abstraction. Examples can be found in the use of functions, procedures and classes.

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.

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.

Uniformity

The consistency of appearance and behavior of language constructs. If the code does not look like any conventional language, the programmer will possibly have trouble moving on from this language. Constructs should be similar to commonly known syntaxes in the given paradigm, or preferably written in the exact same way.

Miscellaneous

There is a possibility that other points of interest will be discovered during the comparison. These will be described in this category.

6.1   Criteria Evaluation

As the criteria are hard to measure in an objective way, the evaluation will be conducted through a subjective discussion. Each of the environments are meant for different purposes, as they are based on different paradigms. Taking this into account, we will create solutions to problems that are fit for each of the environments, and then compare these solutions in how all the environments handle and solve this problem, based on the criteria.

Chapter 7

Language Analysis

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 popular learning environments for languages of each their paradigm. Scratch represents an imperative paradigm, BlueJ an object-oriented paradigm and DrRacket a functional paradigm.

In this chapter, a subjective analysis is made for each of the mentioned languages individually. The procedure used for the analysis is as follows: First, some tasks have been made which are implemented in the different environments/languages. Then each language is analysed individually using the implementations of the tasks, where the analysis is based on the criteria presented in Chapter 6. Finally, the analyses are used to compare the languages and environments. The goal of this analysis is to get an insight into what each paradigm has to offer regarding educational programming and their environment and how they compare. Furthermore, we wish to compare the most common of the educational tools, to find out what their strengths and weaknesses are. This is done for inspirational purposes, giving the possibility of furthering this analysis.

The tasks are made to get as good an overview of the languages as possible. They will sometimes hypothetically favour a certain paradigm over the other, but this is done on purpose to try and find possible problems with the languages. The tasks that are implemented in each of the languages are:

Task 1: Iterator

> This task is to create a list of a certain size, which is filled with random numbers between 0 and 19, and to create a function which adds a number to every other element in the list. The purpose of this task is to get an example of how lists are made and iterated through in the different languages, which also will give an idea of how the languages are structured in general.

Task 2: Fibonacci Sequence

> This task is to implement an algorithm which calculates numbers of the Fibonacci sequence limited by an input by a user. The purpose of this task is to see how an "introductory" algorithm can be implemented in each language. It is also to have a task that somewhat favours the functional

paradigm, as the functional paradigm is meant to be a way to express math in a programmatic way.

Task 3: Cups and Ball

This task is to create a game where a ball is hidden under one of 15 cups, and the goal is to locate the ball. The purpose is to have a task that hypothetically favours object oriented languages, and to see how this is coped with in a non-object oriented language. Two of the three environments/languages does not support graphics as such, so there will be a difference in how this task is handled in between the languages.

Task 4: Hangman

This task is to create a hangman game, where the user is supposed to guess a word with only a set amount of wrong guesses. The purpose of this task is to have a larger piece of software as the previous tasks are quite small in a sense. This will hypothetically favour Scratch and BlueJ over DrRacket, as the functional paradigm is not really meant for this sort of task.
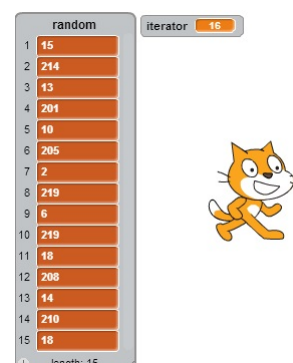
7.1 Scratch

Scratch is chosen as the imperative language, and as the representative of the visual programming languages. As this causes the language to differ in both paradigm and writing style, it also affects the discussion on the criteria. Hence, there will be a focus on both aspects that differs from the other chosen languages.

7.1.1 Iterator

The iterator is made in a standard iterative way, through a simple loop. The debugging tool when clicking a piece of code is used to show a visual representation of the output in the game window. The control structure is a *repeat* loop, which is the same as the well known *for* loop, only with another name. The code for the program can be seen in Figure 7.1a, and the output is seen in Figure 7.1b.
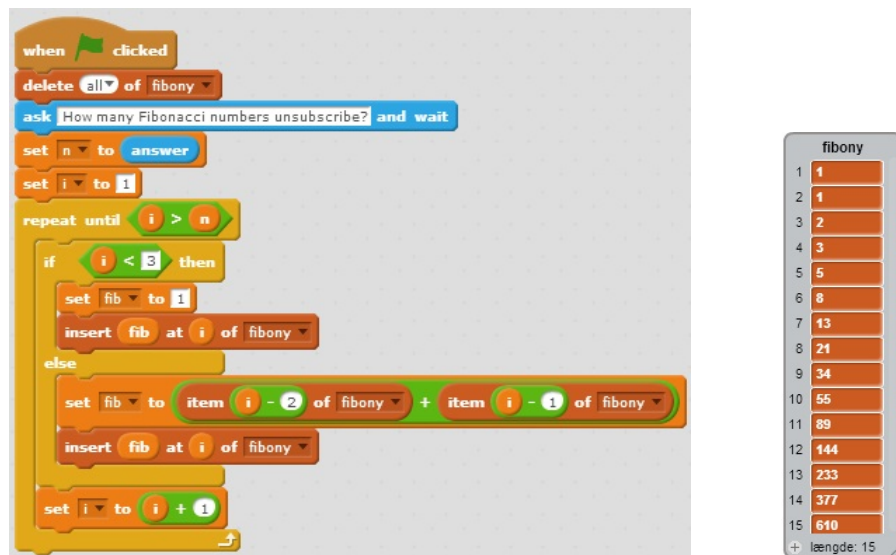

(a) Scratch Iterator code.

(b) Scratch Iterator output.

Figure 7.1: Code and output for Hangman.

29

7.1.2 Fibonacci

The fibonacci sequence is done through simple iteration in Scratch. The implementation can be seen in Figure 7.2a. The user is asked to input how many numbers of the sequence are wanted. With a single loop and a selection, a list is presented, as seen in Figure 7.2b.

(a) Scratch fibonacci code.

(b) Scratch fibonacci output.

Figure 7.2: Code and output for fibonacci numbers.

7.1.3 Cups and Ball

The game of guessing the position of the ball amongst the cups is made with events, as Scratch is able to handle these with blocks. Events happen e.g. when a cup is clicked, cups are cloned, and when the ball is clicked. Code is also attached to different sprites, as these work individually to the events. The code blocks for the cups can be seen in Figure 7.3a, and the code blocks for the ball can be seen in Figure 7.3b. A screenshot of the game screen while in a game can be seen in Figure 7.3c.

7.1.4 Hangman

The Hangman game is made in an imperative manner. As there are many conditions to take into account, the code is rather long, and there is a lot of control structures. The guessing part itself is a big loop, which can be seen in Figure 7.4a. On the game screen, a list holds the letters for the word to guess, a list holds all the wrong guesses, and a sprite changes for each wrong guess. An input field is provided for guessing. The game screen can be seen in Figure 7.4b.

(a) Scratch cup code.

(b) Scratch ball code.
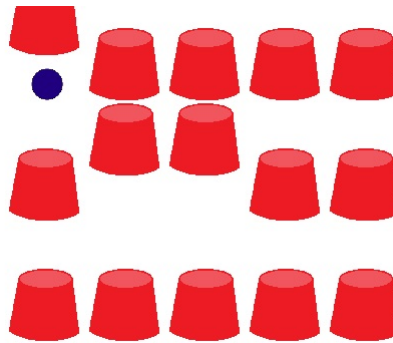
(c) Scratch Cups and Ball output.

Figure 7.3: Code and output for Cups and Ball.

7.1.5 Criteria Evaluation

Readability

Scratch is known for its great readability, and this shows when reading it. The coloured structures clearly show what the different building blocks are doing. The feature is unfortunately lost for colour blind people, as it is not possible to change the colours. The language is very verbose in its statements and declarations, leading to a better understanding of what happens in a block. A problem lies in the fact that a project becomes very big very fast. An example of this can be seen in Figure 7.4a, where the collection of blocks seems hard to read at first glimpse, due to its sheer size.
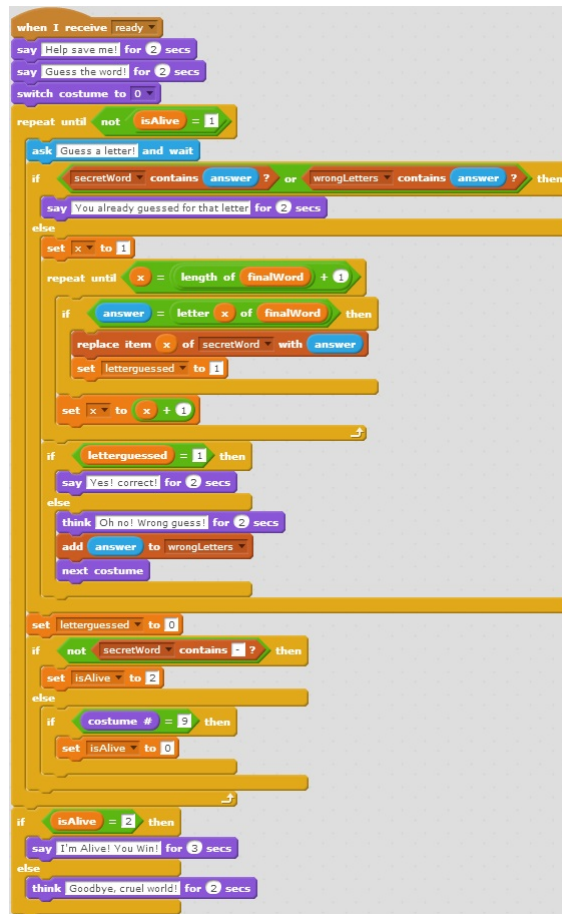
(a) Scratch Hangman code. (b) Scratch Hangman output.
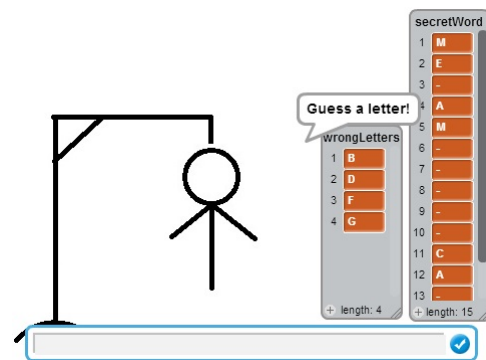
Figure 7.4: Code and output for Hangman.

Writability

Scratch adds to better writability by providing categories with all its possible operations. It is easy to create simple structures, but it can get cumbersome if one has to write a lot of code. It is extremely easy to understand how to use Scratch, due to the fact that it uses building blocks as a way of constructing code.

Observability

Scratch has a live game window, where output is shown when compiling code. Combined with the possibility of double-clicking sets of blocks to compile that specific piece of code, the programmer can see the effect whenever wanted.

Trialability

The visual environment in Scratch allows no syntax errors. Combined with the level of observability, Scratch has great possibility of recovering from mistakes, as the mistake is easily found in the game window. In bigger programs, however, it can be hard to find the mistake, as one cannot follow the stack. Closing in on a small mistake that affects the whole program can be hard.

Learnability

Making a game is a good way to capture the attention of children. But to keep them occupied, the process of coding should be intensive in a playful way. Building blocks from the visual style is a way to do it, as building blocks has been proven successful in the context of playing (Lego is an example of this).

Reusability

As Scratch is also a minor game engine, it uses 2D sprites for visualization. Each sprite can contain code local to that sprite. The code shown in Figure 7.3a is the code local to the Cup sprite, whereas the code shown in Figure 7.3b is local to the Ball sprite. As these can be cloned, the code written serves as a blueprint for all instances of the sprite to use. However, the individual clone of a sprite cannot be accessed directly. Furthermore, Scratch has the possibility of defining custom blocks. These blocks can bring a collection of blocks down to the size of one, and they can easily be reused. These custom blocks work as functional procedures for the language.

Pedagogic Value

Scratch makes use of the most basic of concepts, such as variables and control structures. That said, there is a huge leap when moving from Scratch to a non-visual programming language. The lack of conventional syntax, however, can be hindering further in the programmer's career. That means the language is great for novice programming, but it comes to a stop.

Environment

Scratch has a very friendly environment, with great visibility in the colour coding. As novices are not familiar with anything code-related, the naming of structures is meaningful and easy to understand. Navigation can be a bit confusing at first, as it can be hard to know what category a needed block is found in. The control of the blocks can be a bit annoying, as disconnecting interlocked blocks does not always go as expected. The game window combined with the block compilation, as mentioned, is a great way of obtaining feedback for trial-and-error.

Documentation

Scratch has an integrated introduction tutorial, which is a great way for novices to learn the language. Furthermore, each building block can be right-clicked to find a documentation page for that specific block. This documentation is easy to read, but is unfortunately only available in English. Furthermore, it is possible to find and share projects via Scratch's homepage.

Uniformity

The basic concepts used in Scratch do the same as in other languages. However, the naming of the expressions and structures are often different, e.g. Scratch has a *repeat* structure, instead of a *for* structure.

Miscellaneous

Scratch uses sprites, but it is very hard to make different sprites work together, as the language itself is not object-oriented. As an example, you cannot reference to a specific sprite or one of

its clones. Furthermore, the language has great accessibility in the fact that it is programmed in a browser.

7.2 BlueJ

Since BlueJ uses Java as its underlying language, all the selected tasks will be implemented in an object-oriented manner and there might also be some similarities to their counterparts in Scratch in terms of code structure and use of constructs.

7.2.1 Iterator

The Iterator in BlueJ is implemented using simple iteration through a list with a certain size, specified by the user. Then a *for* loop is used to add a number, in that case 20, to every other element from that list and then the result is printed. The code and the result can be seen in Figure 7.5b.

```java
public class iterator
{
    private int[] list;
    private Random rand = new Random();

    // Konstruktør for objekter af klassen iterator
    public iterator(int size)
    {
        list = new int[size];

        for(int i = 0; i < list.length; i++) {
            list[i] = rand.nextInt(20);
        }
    }

    public void printList()
    {
        System.out.print("[");
        for(int i = 0; i < list.length; i++) {
            System.out.print(list[i] + ",");
        }
        System.out.println("]");
    }

    public void iterate() {
        printList();

        for(int i = 1; i < list.length; i += 2) {
            list[i] += 20;
        }

        printList();
    }
}
```

(a) Code for Iterator
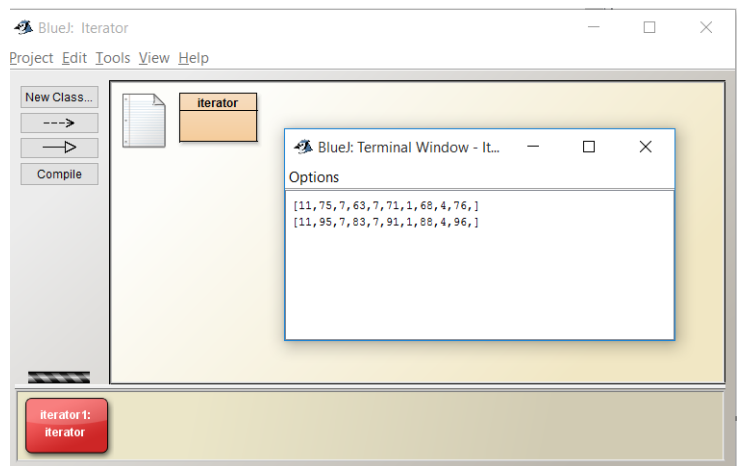
(b) Output for Iterator.

Figure 7.5: Code and output for Iterator.

7.2.2 Fibonacci

The Fibonacci sequence in BlueJ can be expressed both by using an iterative and recursive approaches. An example can be seen in Figure 7.6b, where the user is prompted to give a number and the respective result is shown.

<div align="center">(a) BlueJ Fibonacci code.   (b) BlueJ Fibonacci output.</div>
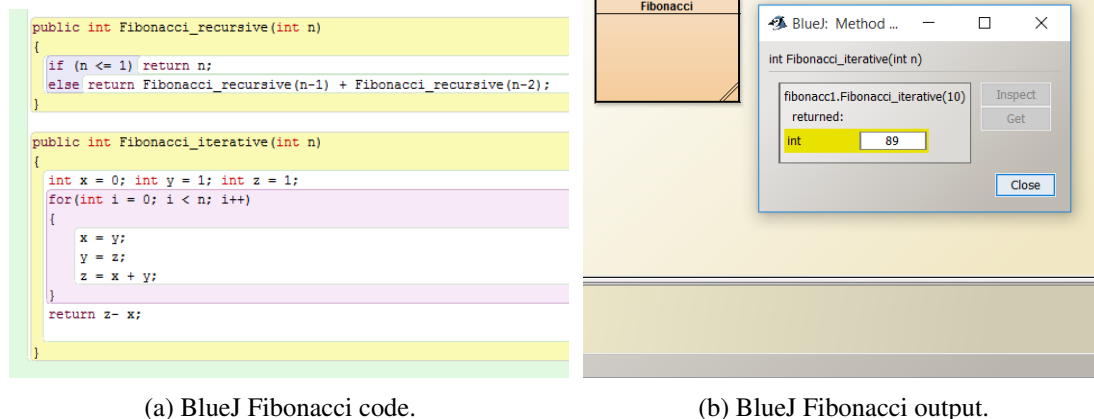
<div align="center">Figure 7.6: Code and output for Fibonacci numbers.</div>

7.2.3 Cups and Ball

Similarly to how this game was implemented in Scratch, it gives the player the chance to guess where a ball might be among 15 identical cups, with the difference that it feels less intuitive since there is no visual feedback given but rather textual one - that the player has either successfully guessed the position of the ball or not. The code for TODO: watever chosen, the class hierarchy and the final output can be seen in Figure 7.7b.

7.2.4 Hangman

The Hangman is also a guessing game where the player has to guess a particular word by providing one letter at the time. In the BlueJ implementation, if the letter is part of the word, then it is added to a list of correct guesses. If not, it is added to a list of wrong guesses instead. Every time the player guesses a wrong letter, one of their lives is lost, thus the `lives` counter decrements by one. If the player is to lose all of them eventually, the game is lost. The code follows an imperative style of programming with conditional statements and loops, some of which are shown on Figure 7.8

```
boolean guessCorrect = false;
Scanner input = new Scanner(System.in);
int guess = 0;

System.out.print("Pick a cup between 1 and 15: ");

while(guessCorrect == false)
{
    guess = input.nextInt();
    if(guess >= 1 && guess <= 15)
    {
        if (cups.get(guess-1).HasBall())
        {
            guessCorrect = true;
            System.out.print("Congratulation, you found it!");
        }
        else
        {
            System.out.print("Miss! Try again, pick a cup between 1 and 15: ");
        }
    }
    else
    {
        System.out.print("Not between 1 and 15, try again: ");
    }
    System.out.println();
}
System.out.println();
```
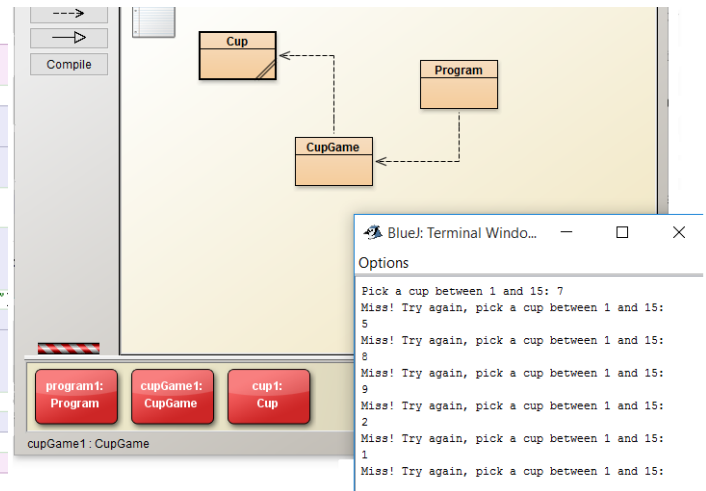
(a) Code for Cups and Ball.

(b) Output for Cups and Ball

Figure 7.7: Code and output for Cups and Ball.


```
boolean isDone = false;
boolean isCorrect = false;

while(!isDone)
{
    System.out.println("Guess a letter:");
    String guess = key.nextLine();
    if(guess.length() > 1)
    {
        System.out.println("Please write only a single letter!");
    }
    else
    {
        isCorrect = words.Guess(guess);
    }

    if(!isCorrect)
    {
        hangman.LoseLife();
        System.out.println("You now have " + Integer.toString(hangman.GetLives()) + " left");
    }

    if(!words.GetGuessed().contains(" _ "))
    {
        System.out.println("You Win! The word was " + words.GetGuessedString());
        isDone = true;
        break;
    }
    else if(hangman.GetLives() <= 0)
    {
        System.out.println("You Lose!");
        isDone = true;
        break;
    }
}
```

Figure 7.8: Code for Hangman game

7.2.5 Criteria Evaluation

Readability

BlueJ has reasonably good readability both by its textual and visual representations. The Main panel provides an overview of the hierarchy and interactions between different classes and objects. This gives novice programmers a greater understanding of how different pieces of the code are connected. Additionally, the textual representation provides good indentation and different colour

schemes for every method so it is easier to understand the structure of the code.

Writability

As already mentioned in Section 7.2, BlueJ uses Java as its underlying language. Writing code follows an object-oriented fashion and additionally keywords are highlighted in order to be differentiated by the programmer more easily.

Observability

BlueJ has fairly good observability as it usually manages to quickly and efficiently compile code. Additionally, single classes can be compiled separately in order to check their correctness. BlueJ's visual representation provides the option to create instances of objects as visual elements and call different methods specific to these objects in an easy, fast and understandable way. Problematic and erroneous sections are highlighted in red with an appropriate message for specific errors.

Trialability

In terms of trialability, BlueJ has a good handling of syntactical and semantic errors. Where need be, exceptions are thrown and the code cannot be compiled before those are fixed. When a piece of code is compiled, if there is an error, the screen jumps to the specific location for the error, with an appropriate error message.

Learnability

BlueJ is first and foremost an educational platform, and it provides an intuitive interface, with a graphical representation of classes and objects. The intuitiveness is found in the possibility of creating and manipulating these through the Main panel.

Reusability

BlueJ uses the established practices for reusing code from Java such as class hierarchy, polymorphism etc.

Pedagogic Value

BlueJ is a powerful educational tool which provides a gentle introduction for novice programmers to the world of object-oriented programming. BlueJ gives the option to visually create instances of classes and objects and playing around with their properties, giving the user a better understanding of how their hierarchical structure works. This is supplemented by its thorough documentation and exercises with steadily increasing difficulty.

Environment

Usually, the Integrated Development Environment (IDE) of a language is the only direct way of communication with that language. In the case of BlueJ, it provides the essentials needed for learning object orientation, specifically targeting novice programmers without the unnecessary clutter present in more advanced IDEs such as Eclipse or NetBeans. Furthermore, it does not require much effort to install it and create projects in it.

Documentation

As already mentioned, BlueJ has a solid documentation with a lot of reference materials and

additional exercises for those who are interested to delve deeper in its features. Also it is fairly easy to get access to the documentation on the BlueJ homepage without the need for any additional books. Furthermore, all of the documentation meant for Java is also applicable for BlueJ.

Uniformity

BlueJ has a high uniformity compared to other languages since Java is one of the mainstream general purpose languages, to which other languages compare their uniformity.

7.3 DrRacket

DrRacket is an environment used to learn to write Racket code. Racket is a functional programming language, and therefore DrRacket is our representative for an educational programming environment for the functional paradigm. Worth noting is that all of of the authors have learned to program in an imperative paradigm first and do not have much experience working with functional languages. This will likely impact the code examples as well as the opinions in the criteria evaluation.

7.3.1 Iterator

The first code example is the iterator example. In functional programming, when one wants to apply a function through a list, one would usually use the map function. However, since that applies the function to every element in the list, and we want to apply it to every other element the code looks like shown in Listing 7.1 instead.

```
1  (define (iterator inList)
2    (if (< (length inList) 2)
3        inList
4        (cons (car inList) (cons (+ (car(cdr inList)) 20) (iterator (cdr(cdr
              inList)))))))))
```

Listing 7.1: The iterator function in DrRacket

The function takes a list `inLinst` in and returns a list. In the trivial case where the list is shorter than two, the function does not need to be applied to anything and the list is simply returned. Otherwise the second element in the list should be modified and the function recursively called on the remainder. To do this three functions are used: `car` which returns the first element of the list called the head, `cdr` which returns the list minus the first element called the tail, and `cons` which combines a head and a tail to create the bigger list. Firstly we want to combine the unmodified head of the list with the modified tail to construct the new list. We then construct the modified tail by modifying the head of the tail, which modifies the second element in the list, and calling the function recursively on the tail of the tail to go through the rest of the list, and combining these.

7.3.2 Fibonacci

The second example is shown in Listing 7.2. This is a simple recursive implementation of Fibonacci with no memory optimizations. Recursion is second nature to functional programming languages, so this is an intuitive implementation. The function takes in a number to find the Fibonacci number of and then calls itself recursively on the two preceding numbers to get the two numbers it needs to sum up. Eventually a trivial case of the called number being one or zero in which case it simply returns one.

```
1  (define (fibonacci n)
2    (if (or (= n 0) (= n 1))
3        1
4        (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))
```

Listing 7.2: The Fibonacci function in DrRacket

7.3.3 Cups and Ball

The next code example is the cups and ball example. This example is intuitively solved in an object-oriented way and since Racket has objects and classes, we do it like that. The definition of the class Cup can be seen in Listing 7.3. Each object of the class has a variable holdsBall, which is used to store whether this cup has a ball, used as a boolean. They also have two functions: AddBall which sets holdsBall to 1, and HasBall which returns holdsBall. The code then creates a list of 15 balls and calls AddBall on one of them chosen randomly. The main game loop is facilitated with a recursive function AskUser, which is shown in Listing 7.4. Here the user is prompted for a number between 1 and 15. The cup on that position on the list then has its HasBall function called. If it is 1 the user is congratulated and the game ends, otherwise the user is prompted to guess again and the AskUser function is called to repeat the cycle.

```
1  (define Cup%
2    (class object%
3      (define holdsBall 0)
4
5      (super-new)
6
7      (define/public (AddBall)
8        (set! holdsBall 1)
9        )
10
11     (define/public (HasBall)
12       holdsBall)
13  ))
```

Listing 7.3: The Cup class definition in DrRacket

```
1   (define (AskUser)
2     (define guess (read))
3     (if (= (send (list-ref cups (- guess 1)) HasBall) 1)
4         (println "Congratulation, you found it!")
5         (begin
6           (println "Miss! Try again, pick a cup between 1 and 15: ")
              (AskUser))))
```

Listing 7.4: The AskUser function in DrRacket

7.3.4 Hangman

The final example is the game of hangman. Here we have a list of 11 words all 15 letters long represented by strings. The initial values are then defined:

- finalWord is assigned a random string from our list of words and represent the word that should be guessed

- wrongLetters represent the list of letters guessed on that were wrong and is assigned to the empty list

- knownLetters is the list of correctly guessed letters and their position. This is initialized to a mutable string of 15 underscores.

- hangman is the number of lives left and it is initialized to 8.

The user can guess on a letter by calling the guess function with a string. If the string is one char long the function checkLetter is called with parameters 0 and the guess string. The function can be seen in Listing 7.5.

```
1   (define (checkLetter st l)
2     (cond
3       [(> (+ st 1) (string-length finalWord))
4        (if (and (equal? hasFound 0) (not (member l wrongLetters)))
5            (loseLife l)
6            #f)]
7       [(equal? l (substring finalWord st (+ st 1)))
8        (printf "correct guess! on place ~a\n" (+ st 1))
9        (set! hasFound 1)
10       (string-set! knownLetters st (string-ref l 0))
11       (checkLetter (+ st 1) l)
12       #t]
13       [else (checkLetter (+ st 1) l)]))
```

Listing 7.5: The checkLetter function in DrRacket

The `st` parameter is the index of the char in the `finalWord` string we are looking at and `l` is the guess string. This function checks for three conditions:

- If the index is larger than the length of the string, we are done going through the string. `hasFound` is initialized in `guess` to 0 and if it hasn't changed it means the letter was not found in `finalWord`. If `l` also was not in the `wrongLetters` list, it means the guess was a new wrong guess and `loseLife` is called. `loseLife` reduces `hangman` by one and adds the letter to `wrongList`.

- If `l` is equal to the substring of `finalWord` at index `st` then the guess is correct. The user is notified, `hasFound` is set to 1, the letter at index `st` in `knownLetters` is changed to `l` and the function is called recursively on the next index to go through the rest of the string.

- Otherwise the function is called with the next index to keep looking through the string.

In the end the function `guess` checks for `hangman` being 0 to report a loss and if underscore is still a part of `knownLetters` as otherwise a win is reported.

7.3.5   Criteria Evaluation

In this section we will discuss DrRacket in relation to our criteria. Again, since the authors are used to another programming paradigm, these opinions might be biased.

Readability

DrRacket is fairly readable, with the mandatory use of parentheses giving it a certain structure. The heavy use of parentheses however does mean that they sometimes can blend together making it harder to distinguish the operations from each other as can be seen in Listing 7.1. The environment does provide an automatic highlight of the area a pair of parentheses cover when the cursor is next to one, but it does not fix the immediate readability. The lack of infix operators hurts the readability since it means that common mathematical operators like plus, minus and greater-than need to be before the two arguments. This breaks the common mathematical notation, which makes it harder to understand intuitively. In general the functional paradigm also has some readability issues since the code usually requires a better overview of the function to understand it. In a more imperative paradigm it is easier to build up an understanding from partial understandings of the sequences.

Writability

DrRacket has good writability, often being able to express things a little more concisely and having a consistent syntax for all sorts of function calls. There is good support for many levels of abstraction with functions being an integral part of the language and it offering constructs for object-oriented programming. Its only problems are a the higher need for overview like in readability, and that the syntax especially around missing parentheses can get hard to keep track of

and often requires some debugging. The parentheses highlight tool does help with that quite well though.

Observability

DrRacket has good observability, as it offers an immediate console on runtime, where one can call all the functions and variables from your code and immediately see the return value.

Trialability

DrRacket has good trialability, as the code compiles quickly and easily and whenever an error is encountered it highlights the code it found the error as well as giving the error message. It does have the usual issues with the error message not always being helpful and sometimes reported at other places than where the actual problem is, and you still need to compile every time you want to test any changes to the code.

Learnability

DrRacket has good learnability with the possibility of using sublanguages to restrict the possibilities available to novice programmers. This allows for slowly opening up the higher level sublanguages to build towards the full language. It also has easily available tutorials to learn the language in this manner. It does however have the problem of not showing the legal constructs to a novice, which means they have to look up the documentation to find examples of how the code should look.

Reusability

DrRacket has great reusability with a strong focus on using functions on many levels and support for objects and classes.

Pedagogic Value

DrRacket has some great pedagogical values in the way that it encourages learning and using recursion, which is a powerful programming tool. Its consistency with function call conventions also help convey the ubiquitous use of function calls. Notably the choice of sticking to the function call syntax instead of allowing infix operators in mathematical expressions, help convey the fact that these are coded as functions in the code.

Environment

DrRacket is a reasonably good environment for Racket. It provides easy setup of compilation to an easily accessible console to evaluate the code with. Its tools for highlighting the paranetheses areas and and marking the definition with a line to it when hovering over a variable, both help keep track of the structure. The environment itself however does not provide any assistance for getting started.

Documentation

DrRacket has a lot of good documentation both in the form of easily accessible online tutorials and an online manual with a search function. There are also several books on the subject as well as a decent amount of online forum discussions on Racket and Scheme.

Uniformity

DrRacket has low uniformity relative to the other languages we know like Java, C# and F#. Its function calls has the parentheses around the whole call instead of only the parameters, and the lack of infix operators makes basic mathematical operations look vastly different.

Chapter 8

Comparative Analysis

In this chapter, a comparison is made for the languages and environments that have been analysed individually in Chapter 7. If one of these has a clear advantage in a criteria, it will be mentioned as being the best of the chosen to fulfil that criteria, based on our subjective opinions.

8.1 Readability

All the three environments we have selected have a varying degree of readability. This is directly tied to the target groups each one tries to address and their respective age. For this reason, we consider Scratch to have the highest readability of the three since it is considered the most verbose and it is the only environment of the three which supports a fully visual programming language. That helps immensely with how programs are structured and represented through the use of simple, modular blocks and how easy is to understand different parts of the code. Keywords in the language are named in such a way that are more descriptive of the action they convey, adding to the verbosity of the language, and more intuitive in terms of understanding compared to the traditional approach used both by BlueJ and DrRacket.

On second place we have BlueJ, which is mainly used in CS1 courses or pre-college courses, involving people of that respected age. This naturally means that object-orientation practices are used for creating programs and there is a heavy emphasis on classes and objects which are modelled after their real world counterparts and highschoolers and students have easier time understanding them. Although not anywhere near the level of Scratch, BlueJ has a visual representation of classes and instances of classes and objects which gives a better overview and understanding of the hierarchical structure of programs than traditional IDEs used with Java.

We place DrRacket last in terms of readability since its entirely textual approach with combination of unconventional use of mathematical operations, brackets and lack of infix operators makes it the hardest to understand environment for novice programmers of varying age, in terms of its underlying programming language adhering to the functional paradigm.

8.2 Writability

Programs in Scratch are not only easy to understand but also relatively easy to make. It is actually the only one from the three environments which does not require any programming knowledge but rather intuition, intended results and trial-and-error since it is specifically designed for young kids. Compared to BlueJ and DrRacket, which have a compilation step, Scratch extends its writability through its reactive nature. Furthermore, great efforts were made during the development of Scratch so syntactical errors could be avoided for the majority of the cases. In terms of semantics, there is intentionally a limited set of possible data types and blocks which could be given as arguments at any given time which reinforces the intuitiveness of the language. However, given the visual programming nature of Scratch, it is generally slower to achieve a given result compared to BlueJ and DrRacket.

BlueJ does not provide much more beyond the standard in terms of writability established by Java, with the exception of the option to colour code, where different parts of the code take whatever colours the user specifies, taking into account colour blind people as well. However, Java makes use of constructs with widely known structure and semantics, similar to many popular languages, making it easier to become accustomed with the environment.

DrRacket has a style of programming defined by the functional paradigm and seems the least intuitive to work with when novice programmers are considered. Additionally, as already mentioned in Section 7.3.5, this is compounded by the fact that the language lacks infix functions, thus common mathematical operations have different syntactical structure. However, it generally takes less number of lines to implement a given task compared to BlueJ and Scrach's more object-oriented approach, which positively affects the writability of the environment.

8.3 Observability

In terms of observability Scratch is the best of the three languages. The visual representation allows for a greater range of feedback that can still be intuitively read, for example by showing the motion of a sprite rather than just the endpoint. The ability to simply click on a code block or a set of blocks to run it and see the results also makes the code to feedback time really small. On the second place we have DrRacket. It only requires a click for compilation and sometimes a function call, where BlueJ requires instantiation of an object as well. BlueJ also requires the user to add print statements to get feedback from the functions, where DrRacket will automatically print the return value.

8.4 Trialability

In terms of trialability Scratch wins again albeit not by much. Its quick feedback-loop and the decomposable execution are great factors for trialability, but its lack of a slower runtime evaluation mode hurts it a bit. DrRacket and BlueJ are very close, with both of them highlighting the error code on encounter

amongst other features. DrRacket only wins here on account of being an instantiation of objects shorter than BlueJ in its feedback-loop.

8.5   Learnability

Scratch is the clear winner in learnability. The list of operations both serve to give some suggestions for actions, and helps give an overview of the possible actions. This is something both DrRacket and BlueJ lack, as the textual coding interface requires the user to have an idea of how the syntax and semantics work before they can make anything. This means that they both rely on some additional instruction, either through an instructor or a tutorial, before the user can start working. Neither of the two provide an in-environment tutorial like Scratch.

8.6   Reusability

Both BlueJ and DrRacket have good features supporting reusability, both supporting classes and objects. Although, BlueJ is a little ahead of DrRacket in this criteria as it is using an object-oriented programming language which possesses a great deal of features for reusing code through abstraction. This fits well with the definition of the criteria shown in Section 6. Even as DrRacket has support for classes and objects, it is not the main focus of the language which places it a bit behind BlueJ. Scratch has limited features supporting this criteria with its cloning of sprites and function definitions, lacking features for abstraction.

8.7   Pedagogic Value

Each of the languages/environments provides pedagogic values for different aspects of programming. Scratch has pedagogic value in the sense that knowledge about basic low level concepts is easily gained. It has a possible shortcoming in a user transitioning to a text-based programming language, as some of the features that they are used to are missing. BlueJ does not provide a lot regarding basic concepts of programming, but provides a lot of tools to get an understanding of how classes and objects works. DrRacket provides pedagogic value in the sense that it can provide an understanding of the underlying functionalities in programming languages. As an example, in a lot of programming languages, the user will use the expression $2 + 2$ to get the value 4, possibly assuming that it "just works", where DrRacket shows that using the $+$ operator is a function call in itself. DrRacket also encourages learning and using recursions, as it is the intuitive way of working with it compared to loops.

8.8 Environment

As scratch is a visual programming language, its environment differs greatly from both BlueJ and Dr-Racket. With this said, BlueJ also offers a visual representation of the structure. DrRacket has no graphical advantages that would help novices in learning to code, other than libraries that provide high order functions for creating shapes as an output. This results in Scratch and BlueJ being better for a younger targeting group. As an addition to this, Scratch also functions as a small game engine, with a lot of visual representations of its mechanics and output. This, along with the playful nature of the environment, makes Scratch the environment of the three best fitting children in primary school.

BlueJ is used for programming Java, and its environment is excellent in learning novices the concepts of objects and classes. It provides a lot of helping functionality, which is not found in other development environments for the language. Scratch is good for teaching younger novices the basic concepts of imperative coding. DrRacket helps in understanding declarative programming, which is much different from most well-known paradigms. They each have their merits, but as an educational tool for teaching novices, Scratch has the most helpful environment.

8.9 Documentation

Each of the language environments have different ways of providing documentation. BlueJ has an advantage in being an environment for programming Java, which leads to a lot of documentation and helping forums. Furthermore, it has documentation on how to use the environment itself. Scratch as a language is very dependent on its environment, as it is a visual language, which leads to a mixture of environment and programming documentation. DrRacket has little documentation on how to use the environment as a whole, but more about how to use it to code. This means most of the documentation is about Racket as a language.

There is a possibility of finding online tutorials for all the languages. Scratch is very educationally friendly in that area, as it provides a tutorial as a part of the programming window. DrRacket also offers a very thorough guide and reference work on their homepage [47], and BlueJ offers a paper, serving as a tutorial and guide for the environment [48]. The documentation for BlueJ is unfortunately not very novice friendly, as it assumes the programmer is known in Java.

8.10 Uniformity

The three languages have little in common overall, which means a lower rating of internal uniformity. As BlueJ is an environment of a well known programming language, its uniformity is unquestionable when it comes to object-oriented programming. Scratch is very simplistic, and its basic constructs are similar to those in widely used languages, although they are differently named. It also has an advantage for novices, as the declarations often don't require knowledge of coding conventions for that type of ex-

pression. As an example, the block setting the value of a variable shows "set [NAME] to [VAL]", instead of the traditional `[NAME] = [VAL]`. In Scratch, `[NAME]` is even a drop-down list of possible variables, and `[VAL]` can both be written directly, or be a variable block. DrRacket, being an environment for a functional programming language, has a very different syntax to other paradigms. The lack of infix operators and other convention in parentheses makes it hard to transition from this language to other paradigms. This fact therefore also applies the other way around. On the other hand, Racket is built on a well known language, Scheme, and the transition to other languages in the same paradigm is easier. As even the operators are functions, it gives a great understanding to the nature of lambda functions.

Chapter 9

Results

The three languages have been analysed through a set of criteria, and through this it shows that Scratch is the language and environment best suited for novice programming. Its level of intuition and well-structured environment makes it possible for a novice to understand the concepts without the need of external resources. Unfortunately, this fact does not persist in BlueJ and DrRacket. Scratch is extremely easy to read, as its syntax for statements resembles natural language. The feedback level gives possibility for great observability and trialability, which leads to a more steady learning curve. Many of these factors contribute to the learnability of Scratch, which therefore makes Scratch the language best suited for novices among the languages compared. BlueJ is ahead in both reusability and uniformity, which results in a language common to other languages in its paradigm, and with a great expressibility. Its graphical representation of classes offer a great overview of the class hierarchy, which is a great help for novices. BlueJ unfortunately isn't as intuitive for novices without help, such as a teacher or tutorial. DrRacket had its merits in different areas, where writability was one of those of higher value. When understood, the language converts thoughts to code in a fast paced manner, but can be extremely hard to understand without help from outside the environment.

An interesting evaluation would be to compare the languages through the criteria used to develop Grace [49], as this is meant to be an educational language. There are three main criteria used to develop Grace, being object construction, type checking and language levels.

As object construction is a feature only available to object-oriented languages, this can be evaluated through the simplicity of implementing core features for each paradigm. Scratch is an imperative language, with a hint of procedural programming, and the nature of the paradigm is to declare statements. This objective is filled by using blocks for each type of statement. This clearly defines what the statement does, and makes a clear separation in the different statements. BlueJ is object-oriented, as it is an environment for Java programming. The main feature in this paradigm is creating and using classes and objects. BlueJ offers a graphical interface, where classes can be created and related. This offers a great help for novices, to get a greater overview of the structure of the code. DrRacket is for Racket, which is a functional programming language. The main feature in functional programming is expressions. Racket in itself consists of expressions, and it is easy to create and use expressions. DrRacket provides a text

editor for coding, but not much more help is given in this criteria.

Type checking is a factor which depends on whether the language is strongly or weakly typed. Scratch has only a very small set of data types, and the only collection type is the list. This makes the language very easy to control, and the reliability of the language is therefore of high level. The language itself handles all type checking, and the user is not needed to understand types in depth. BlueJ is based on Java, and it has the possibility of choosing. As a coding convention, the type is normally strongly typed, but keywords such as *var* make it possible for weak typing, as well a more dynamic typing. Unless used correctly, though, the program may not compile, as a perfect reliability for a multi-purpose language such as Java is near impossible. DrRacket is a weakly typed language, which makes it possible to make some fairly abstract expressions, in the form of functions. For an advanced programmer, this leads to a wide list of possibilities. As a novice, however, the possibility of using the expressions incorrect is very large.

Language levels is actually an idea for Grace inspired by DrRacket [50]. DrRacket has a feature of selecting languages depending on experience, which is a great asset for the language to be fitting all experience levels. None of the other languages have this, where BlueJ only provides an easier approach to a multi-purpose language, and Scratch is almost only fit for novice programming. Although the choice of experience level provided in DrRacket helps throughout the tutorial in understanding the language, it still does not help much without external resources.

The languages and environments almost fill the requirements set by the people behind Grace. Scratch and BlueJ do not have different language levels. Scratch is focused on novice programmers, whereas BlueJ has chosen to handle the problem through an environment instead. This problem is of no greater concern to this project, as the focus is only on novice programming. With that said, the conclusion might also suggest that Scratch works better for novices than the other languages and environments, since it focuses only on this aspect. With this short comparison in mind, Scratch still comes out best of the three compared tools.

Part III

Conclusion

Chapter 10

Conclusion

Through this report, we have analysed the field of novice programming and the problems regarding learning programming. We have investigated several languages for novices, and discussed their merits and shortcomings. There is a huge difference between visual- and text-based programming, which is addressed, where it was found that visual programming has an advantage in its ease of access and getting acquainted with, where the user for example do not use the keyboard for much. Text-based programming, on the other hand, requires a better understanding of how a computer is operated, which could lead to a problem in the younger group of novices. We have analysed the field of teaching, and concluded that the teaching approach varies from country to country. The U.K. seems to have a good grasp on novice programming, where Denmark has some trouble, as the teaching is done by possibly unqualified teachers. The US have not made programming mandatory yet, but as an optional course, counting towards final grades. Furthermore, research is being made in how to incorporate programming in other courses.

We chose to continue the analysis with three novice languages and environments, being Scratch, BlueJ and DrRacket, for a language comparison. This was carried out as a subjective evaluation by the authors, with the intention of understanding the novice approach to programming. We analysed each language by a given set of criteria, which were criteria for evaluating programming languages in general, and for evaluating the interface design. Some sample pieces of code were made for comparison, which were made to match each of the paradigms representing the languages. After discussing each criteria for each language, with examples through the sample code, we compared the languages in each criteria. We concluded that a great difference lies in the targeting age. Scratch can work as a great introduction to programming down to the age of a primal school pupil. BlueJ serves as a great introduction to general purpose programming, but requires more effort and concentration, as well as external recourses for understanding Java. DrRacket can be hard, due to the different syntax. Furthermore, as with BlueJ, it needs external resources to function as a novice environment.

Chapter 11

Discussion

During the initial stage of our project, we spend a great deal of time and effort on narrowing down the specific problem we wanted to address in the research area of our choice. The field of programming technologies is quite big and varied, and it encompasses many research areas and avenues one can explore. In order to address the raising demand for programmers worldwide, educational institutions spend more and more time and resources on properly educating people for such a monumental task. However, there is not a well established way of achieving this task, especially for people in the early stages of their lives. Through our research and analysis of the existing research domain we chose to focus on identifying what the difficulties are that young people face when they start learning programming and what the most common and severe mistakes they usually make are. The reason for this focus was the fact that the evidence for the design decisions of some programming languages was quite limited. The best case we found was an article by Stefik and Siebert [51], and even then it did not tell us a lot about the design decisions for making it novice friendly. Instead they compare its usefulness to other languages for novices. We tried to cover as much as possible from the research area by analysing both text-based and visual-based programming languages.

There were several difficulties which we faced while working with our research question. First, we had difficulties identifying which might be the most severe problems pupils face when given programming tasks to solve. The research on that specific area is highly opinionated and there is not a single right and perfectly accurate answer how to achieve that. Therefore, we focused on researching the resources we considered most important for novices. Second, we spend considerable time on deciding which are the most relevant programming paradigms in the context of teaching programming and which educational languages are best fit for representing those paradigms, reflected by their usage by novices. Last but not least, we had to familiarize ourselves with each one of the three programming environments we have chosen in order to acquire a better overall understanding of their inner workings and to be able to give our subjective opinion as part of the analysis. Given that we all a background in imperative programming, we found DrRacket the hardest to understand due to the nature of functional programming.

We consider Scratch as a great choice for teaching programming to novices given its pedagogical value and learnability. It provides a fast way to implement different ideas without the need to have knowledge

53

about the entirety of the platform. However, we are still conscious about its shortcomings and how novices can quickly outgrow the platform. Although the verbosity of the language is one of its main strengths, it is also one of its weaknesses since the code built mostly by blocks tends to get very big for medium-sized projects and impractical for larger ones, which might significantly reduce its readability. We are aware that this is an intentional design decision made by the Scratch team, since the purpose of the language is to teach the basic concepts of programming rather than be a fully-fledged programming tool for building programs.

During our research we investigated several new programming languages which were mostly educational by design and tried to address the ease of use while programming. What most of these did was to make claims which they did not necessarily had the empirical data to support scientifically which also made it harder to reproduce their state results.

Digital literacy is an essential component of a modern education, which can directly affect a particular country's economy. Currently, there is a growing shortage of people skilled in ICT and there is an estimate is that there will be 900,000 vacant jobs in Europe's ICT Secton by 2020 [52]. This is one of the main reasons why the introduction of programming as a mandatory course became a reality in Denmark in 2014, following the trend established in several European countries, starting with England. Based on their age, children are getting lessons on algorithms, coding in languages such as Scratch, and debugging programs. In Denmark however, at this point there are no dedicated teachers for teaching programming, but rather that role is filled in by the physics teachers in schools. This might mean that the learning experience which is provided might not be on a satisfactory level, but there is still no sufficient data to support that claim.

11.0.1 Coding Pirates

Even though the Danish government might not have the best handle on how to incorporate programming into the school system (See Section 5), there is another organization which tries to put programming in the hands of children. This organization is called Coding Pirates[1], and have multiple departments in different cities in Denmark. We have visited one of these departments located in Aalborg, to get a hands on experience with how they approach teaching programming to children. This also resulted in a conversation with one of the board members of the organization, Magnus Toftdal Lund. We talked about the intentions of the organization, his observations about the children learning to program, and how the other departments were doing, as the one we visited was just starting up.

At the event we split up and followed two groups, one for 7-10 year olds who worked with scratch and one for the 10+ year olds who worked with Unity. Sadly, the group working with Unity did not get further than installing the software. The Scratch group got a bit further, first learning to log in on the Scratch website and then having a basic introduction to Scratch. The introduction was based on making a sprite of a cat dance, by moving it forwards and backwards infinitely, and changing the background so it was more fitting for a dance party. We observed for a bit longer where we saw that already after

---

[1]https://codingpirates.dk/

the basic introduction, some of the children had started experimenting on their own, and were asking questions about how to do certain things. This was also the intent of the instructor present, as he made it very clear that the event was not school, but about doing what they wanted. The instructor and volunteers where just there to help them get started and help them if they encountered problems, beyond the basic introduction. They would of course also help with ideas for projects if the children had none of their own.

After observing the group, we went to have a talk with Magnus. We first discussed how fast the children were learning and how long it took for them to get comfortable with Scratch. He stated that already on the first day, they try to experiment, as we also observed, and that they relatively quickly obtained a good understanding of Scratch and how to manipulate different aspects of it. He also stated that they tried to help the children to get comfortable with Scratch quickly as they felt that it was quite limiting in what could be done without having to do a lot of workarounds, but a good introduction device none the less. The aim is to give them enough knowledge about programming to move on to "Pygame"[2], and if the children wanted to continue on from Pygame to 3D games, give them further knowledge to start using Unity. We tend to agree with his opinion about Scratch after we had our own experience with the environment, as stated earlier in this chapter.

The second part of the discussion was about the litterature on what obstacles children have when learning to program. We asked him about obstacles regarding syntax when the children moved on from Scratch to a text-based programming language. He stated that some scientists would focus on obstacles which, in the big picture, produced relatively limited problems for novices, here among the syntax of text-based programming languages. In his experience, syntax was something easily overcome by a little bit of experience, but did not specify further.

The third part of the discussion was about the philosophy they imployed when trying to teach children programming. He talked about the fact that in universities, when the students start learning to program, they are often introduced to the features of the language, and first then a problem where those features were applicable. He thought that it was not an efficient way of doing it, as the students might ask why e.g. integers, floats, double and so on, was needed. The philosophy that they employ is the opposite, which focuses on presenting a problem where these features are needed to solve the problem. That way the children learn about the features knowing why, instead of "just needing it to pass an exam".

In the end, he stated that Coding Pirates was not actually for learning to be a great programmer, but making the attendants experienced in reading and understanding code and making them able to tweak code, but not creating code from scratch. The main focus of the events becomes game design rather than programming, when the attendants have a good enough knowledge base about programming. One of the observations that he mentioned was the fact that the attendants will start to split up into smaller teams and specialize in certain aspects of game creation e.g. one who draws the sprites, one who does the programming, one who comes up with the rules, etc. This makes them able to handle most aspects of game creation for a project internally in the team.

---

[2]http://pygame.org/hifi.html

11.1 Further Works

Following the conclusion of Shane Markstrum [16], the programming language community, including us, need to provide some scientific data behind any claim made on the presentation of the language. The common suggestion is to use the techniques from social sciences to gather data [53], but these techniques often require a number of participants in the thousands to make the data significant enough. This is usually beyond the resources available to the language developers, and far beyond our resources. At the same time there has been a large push to make programming more accessible to the general public, which is especially evident with Scratch, which is built to be intuitive enough to be used without external assistance. This leads us to the idea of using the usability test from human-computer interaction on programming language environments. These tests are used to see where a program has problems with the users easily interacting with the system. Since programming language environments can be seen as programs to allow the user to interact more easily with the language, this means the tests are applicable on them. The problem-finding focus of these tests also help avoid the risk of the testing bias problem noted by Magnus, where researchers might focus on testing out a problem that might not really be that important. Furthermore, usability tests only require a few participants to say a lot about the program, making it a lot more accessible to developers, who in fact are already frequently using them to improve their programs. These tests could both be used to analyse the existing languages as well as any new language that might develop.

It could be interesting to perform our tests on novices, as that would provide data without our experience as bias. These test could be performed by having the participants, maybe following a bit of introduction, try to write the four programs we wrote. Data would either be collect by recording like in a usability test, having a discussion about the criteria afterwards or a combination of the two.

Given that Scratch is such a great environment for novices, it makes sense to teach it as a first language. However, as supported by Magnus, even with its extensions it is still limited in its application, and does not provide the ability to start making programs for all sorts of applications like a general purpose language can do. Furthermore, Scratch's appearance is too different from a general purpose language to facilitate an easy transition to those. This means a potential project could be to make a programming environment that makes it easier to transition from Scratch to a general purpose language like C#. Of course this would require figuring out which parts of the transition are difficult, and trying to break those into smaller more easily learned steps.

Alternatively one could try to make a language with general purpose applications, but which tries to replicate the intuitiveness from Scratch's interface to make it more accessible. This could be a great challenge since a lot of Scratch's intuitiveness comes from its visual interface, which might not be an easy fit for a programming language designed for larger and more varied programs.

Part IV

Bibliography

Bibliography

[1] S. Papert, *Mindstorms: Children, Computers, And Powerful Ideas*, pp. 11–15. Basic Books, Inc., 1 ed., 1980. 1, 4.1.1

[2] Microsoft, "Small basic faq." `http://smallbasic.com/faq.aspx`. Used: 26/09/15. 1, 4.1.1

[3] T. Software, "Tiobe index for october 2015." `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`. Used: 26/09/15. 1, 4.1.2

[4] M. John, R. Mitchel, R. Natalie, S. Brian, and E. Evelyn, "The scratch programming language and environment," *Journal Name*, vol. 10, no. 16, 2010. 1, 4.2.1, 4.2.3

[5] 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

[6] S. Papert, *Mindstorms: Children, Computers, And Powerful Ideas*. Basic Books, Inc., 1 ed., 1980. 2

[7] S. Papert, *Mindstorms: Children, Computers, And Powerful Ideas*, p. 218. Basic Books, Inc., 1 ed., 1980. 2, 4.1.1

[8] O. Meerbaum-Salanta, M. Armonia, and M. M. Ben-Aria, "Learning computer science concepts with scratch," 2013. 2

[9] C. A. School, "Computing in the national curriculum." `http://www.computingatschool.org.uk/data/uploads/CASPrimaryComputing.pdf`. Used: 21/12/15. 2

[10] Emu.dk, "Vejledning for faget matematik." `http://www.emu.dk/modul/vejledning-faget-matematik`. Used: 21/12/15. 2, 3.3, 5.1

[11] M. J. Lee, A. J. Ko, and I. Kwan, "In-game assessments increase novice programmers engagement and level completion speed," *ICER '13 Proceedings of the ninth annual international ACM conference on International computing education research*, pp. 153–160, 2013. 2

[12] K. Roxane and S. Wolfgang, "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. 2, 4.2

[13] M. Armoni, O. Meerbaum-Salant, and M. Ben-Ari, "From scratch to "real" programming," 2015. 2

[14] Y. Matsuzawa, T. Ohara, M. Sugiura, and S. Sakai, "Language migration in non-cs introductory programming throguh mutual language translation environment," *SIGCSE '15 Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pp. 185–190, 2015. 2

[15] A. Stefik and S. Siebert, "An empirical investigation into programming language syntax," 2013. 2

[16] 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. 2, 11.1

[17] J. Scott and A. Bundy, "Creating a new generation of computational thinkers," *COMMUNICATIONS OF THE ACM*, pp. 37–40, 2015. 2

[18] T. R. Society, "Shut down or restart?." `https://royalsociety.org/topics-policy/projects/computing-in-schools/report/`, 2012. Used: 7/01/16. 2

[19] 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. 3.1, 3.3

[20] Cunningham & Cunningham, Inc., "Functional programming." `http://c2.com/cgi/wiki?FunctionalProgramming`, 2011. Used: 15/11/15. 3.3

[21] M. McMillan, "Teaching functional programming techniques in non-functioanl programming languages," *Conference Tutorial*, vol. 27, pp. 66–67, 2012. 3.3

[22] 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. 3.3

[23] K. Periyasamy, D. Riley, K. Hunt, S. Hansen, and P. Langhals, "Teaching cs 1: Object-first or algorithmic," *Journal of Computing Sciences in Colleges*, vol. 28, no. 1, pp. 111–112, 2012. 3.3

[24] Harry H. Porter III, "Smalltalk: A white paper overview." `http://web.cecs.pdx.edu/~harry/musings/SmalltalkOverview.html#Introduction`, 2003. Used: 20/12/15. 4.1.1

[25] Georgi Dalakov, "The dynabook of alan kay." `http://history-computer.com/ModernComputer/Personal/Dynabook.html`, 2010. Used: 20/12/15. 4.1.1

[26] A. Kay, "Alan kay shares a powerful idea about teaching ideas." `https://www.youtube.com/watch?v=JDpsXWuedVc`, 2014. Used: 26/09/15. 4.1.1

[27] 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. 4.1.1

[28] Microsoft, "Small basic curriculum: Online." `http://social.technet.microsoft.com/wiki/contents/articles/16982.small-basic-curriculum-online.aspx`. Used: 26/09/15. 4.1.1

[29] Qourum, "Epiq 2016." `http://quorumlanguage.com/epiq.php`. Used: 26/09/15. 4.1.2

[30] M. Kölling, "Bluej features overview." `http://www.bluej.org/about.html`. Used: 01/11/15. 4.1.2

[31] M. Kölling, "Bluej blackbox data collection project." `http://www.bluej.org/blackbox.html`. Used: 01/11/15. 4.1.2

[32] B. team, "Bluej blackbox data collection for researchers." `https://blackboxdc.wordpress.com/`. Used: 01/11/15. 4.1.2

[33] racket lang.org, "2.3 how to design programs teaching languages." `http://docs.racket-lang.org/drracket/htdp-langs.html`. Used: 16/12/15. 4.1.2, 5

[34] M. Boshernitsan and M. Downes, "Visual programming languages: A survey," pp. 1–28, 2004. 4.2

[35] MIT Media lab, "Scratch statistics." `https://scratch.mit.edu/statistics/`. Used: 28/09/15. 4.2.1

[36] S. Fincher, S. Cooper, M. Kolling, and J. Maloney, "Comparing alice, greenfoot and scratch," *SIGPLAN ACM Special Interest Group on Compouter Science Education*, pp. 192–193, 2010. 4.2.1

[37] B. Moskal, D. Lurie, and S. Cooper, "Evaluating the effectiveness of a new instructional approach," *SIGCSE ACM Special Interest Group on Compouter Science Education*, pp. 75–79, 2004. 4.2.2

[38] Version2.dk, "Programmering kommer til folkeskolen uden plan for efteruddannelse." `http://www.version2.dk/artikel/programmering-traeder-ind-i-fysiklokalet-68296`. Used: 21/12/15. 5.1

[39] Bbc.com, "A computing revolution in schools." `http://www.bbc.com/news/technology-29010511`. Used: 21/12/15. 5.1

[40] Computing.co.uk, "Bcs given £1.1m to help teachers prepare for new computing curriculum." `http://www.computing.co.uk/ctg/news/2317331/bcs-given-gbp11m-to-help-teachers-prepare-for-new-computing-curriculum`. Used: 21/12/15. 5.1

[41] Theguardian.com, "Coding at school: a parent's guide to england's new computing curriculum." `http://www.theguardian.com/technology/2014/sep/04/coding-school-computing-children-programming`. Used: 21/12/15. 5.1

[42] Code.org, "President obama asks america to learn computer science." `https://www.youtube.com/watch?v=6XvmhE1J9PY`, 2013. Used: 19/12/15. 5.1

[43] Code.org, "Where computer science counts." `https://code.org/action`. Used: 19/12/15. 5.1

[44] A. Yadav, C. Mayfield, N. Zhou, S. Hambrusch, and J. T. Korb, "Computational thinking in elementary and secondary teacher education," *ACM Transactions on Computing Education*, vol. 14, no. 5, 2014. 5.1

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

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

[47] "Racket documentation." `http://docs.racket-lang.org/index.html`. Used: 15/11/15. 8.9

[48] M. Kölling, "The bluej tutorial." `http://www.bluej.org/tutorial/tutorial-201.pdf`. 8.9

[49] A. P. Black, K. B. Bruce, M. Homer, and J. Noble, "Grace: the absence of (inessential) difficulty," *Onward! 2012 Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, pp. 85–98, 2012. 9

[50] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen, "Languages as libraries," *PLDI '11 Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 132–141, 2011. 9

[51] 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," *SIGPLAN ACM Special Interest Group on Programming Languages*, pp. 3–8, 2011. 11

[52] K. Pretz, "Computer science classes for kids becoming mandatory." 11

[53] 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*, pp. 39–54, 2012. 11.1