From Educational Programming to Professional Programming

Group DPT906E15 - Room X.X.XX

4 February - 1 June

_____   _____
Date   Jais Morten Brohus Christiansen

_____   _____
Date   Henrik Vinther Geertsen

_____   _____
Date   Svetomir Kurtev

_____   _____
Date   Tommy Aagaard Christensen

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Whist on Mobile Devices

Project period:
4 February - 1 June

Project group:
DPT906E15

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

Supervisor:
Bent Thomsen

Abstract:

Pages: 22

Appendices: 0

Copies: 2

Finished: 1 June 2015

Contents

Chapter 1

Introduction

TODO: This was initially a chapter in the report, but as it came to be somewhat a presentation of the motivation for the project, we moved it to the introduction. It is not finished as it is, but the general idea is there. More to come! Traditionally programming have been seen as a specialized skill, only relevant for people who work on developing new software. While several attempts at making languages more accessible to everyone TODO: reference logo and probably some more they ultimately failed to get enough widespread use to change this perception. As a result programming education has been targeted towards college or university students and often had a focus on teaching the student to use professional programming languages like JavaTODO: can probably use source about the widespread use of Java here. The goal here is to create competent software engineers with the ability to work with powerful tools on complicated software.

However 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 like problem solving and collaborative communication. This leads to this education usually being given on visual educational languages like Scratch for their intuitiveness and simplicity.

Of course this means that the two educations differ in what they teach. First the difference in programming languages between being taught an educational language and a professional language. While the educational language has the advantage of being intuitive, it does not have the expressiveness and robustness for large projects that professional languages have. Second the computer science education needs to cover a lot of topics to give a sufficient understanding to work professionally with software, where the kids education leaves out a lot of the topics, though which ones may vary between teachers. Using the list of topics from "What do Teachers Teach in Introductory Programming?"[bottom link] as a reference, we can for example say that topics like algorithm design and debugging are likely to be taught in the kids education. Meanwhile topics like algorithm efficiency, pointers and object oriented programming are usually left out.

Kids now being taught programming in primary school is great for the general digital literacy of people, but the education can not teach everything necessary to do professional programming. We want to work with teaching the skills for professional programming using the already taught knowledge from primary school. To do this we first need to establish what level people have after working in the educational language and what knowledge is needed to work professionally with programming. This paper aims to do this as well as suggesting some ideas for ways to bridge that gap.

Part I

Problem Analysis

Chapter 2

Error-Prone Areas for Novices

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

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

2.1 Syntax and Semantics

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

One of the most error prone areas for novice programmers is the basic syntax [1]. This consists of brackets, semicolons, commas, and other such symbols, symbolizing control for the program. This problem might relate to an even greater problem in understanding the strict control that is needed when writing code in general. When writing code, even the smallest mistake or forgotten symbol leads to a compiler error. This error margin isn't seen very often in other lines of work, and might discourage the novices from keep trying.

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

2.2 Pragmas

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

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

2.3 Programming Paradigms

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

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

Object oriented programming (OOP) has it's values in representing real world problems, and how a solution can be modelled. As OOP is mostly based on classes, being the static description of an object, and objects, being the dynamic model of a real world phenomenon, the concepts of the paradigm can be easily grasped. Of course, this fact demands a teaching method suitable for the novice programmers being taught. On the other hand, OOP is often in relation to procedural programming seen as not being something else, but the same, only with OO features [1]. This leads to a problem of both understanding the very basic concepts of programming, such as control structures (loops and selections), and understanding the OO approach.

It is discussed widely what approach is the most efficient teaching method (TODO: need some refs here). For instance, some say it is necessary to learn the concepts of OOP before learning to code, and some say the basic constructs are necessary before learning about different paradigms and advanced structures. The procedural approach is being taught in elementary school in various languages (TODO: assumption, need refs). In OOP, the question is often what teaching methods are used to make students

understand the concepts of the paradigm. Studies have shown a better effect when teaching about the concepts before actually coding [2]. Another approach to pursue could be the "object-first approach". As the name implies, this approach searches to understand the logic behind objects before anything else. This method can both imply teaching concepts or coding before anything else, and can both be used by novices or by students that have already learned the basics of programming. Although, there is still discussion on whether novices should be taught though this approach, or the classic "algorithms-first approach" [?]. One drawback is the fact that the concepts of OOP might be seen by the novices as the basics for programming itself, instead of the basic concepts, which could lead to a block when exploring other paradigms.

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

Chapter 3

Programming Languages and Tools

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

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

3.1 Visual-based Programming Languages

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

3.1.1 Visual Programming

In order to address these difficulties, for the past 25 years, research has been done on the so called "Visual Programming" or "Graphical Programming", and dozens of visual-based programming languages have been created. This approach, reserved and used in the past primarily for systems design, allows the use of spatial representations in two or more dimensions in the form of blocks and different structures and shapes. Compared to text-based programming where lines of code are used, graphical programming re-

places these with visual objects, essentially replacing the textual representation of language components with a graphical one, more suitable for visual learners and intuitive for people with no prior knowledge in programming. The creation of programs in such languages is defined by placement and connection between visual objects where the syntax is encoded within the objects' shapes. The main aim of visual programming languages (VPL) and environments, as stated by Koitz ans Slany [3], 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.

3.1.2 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 for them a soft introduction to the world of programming. 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 [4]. 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. Given this fact and the steady increase of its popularity have prompted hundreds of schools and educational organizations to adopt and integrate it into their curriculum [5].

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

3.2 Text Based Programming Languages

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

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

3.2.1   Text Based Educational Programming Languages

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

Turtle Programming

One of the first programming languages that added constructs for learning programming was LOGO. It did not have the constructs from the start, but after 12 seventh-grade students[1] worked with LOGO for a year (1968-1969), Seymour Papert, one of the developers of LOGO, proposed the Turtle as a programming domain that could be interesting to people at all ages. He proposed it since the demonstration had confirmed that LOGO was a learn able programming language for novices, but he wanted the demonstration extended to lower grades, ultimately preschool children. Constructs for Turtles was then added to LOGO and has since been widely adopted in other programming languages such as SmallTalk and Pascal, and more recently Scratch [6].

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

- FOWARD X, moves the Turtle X number of Turtle steps in a straight line

- RIGHT X, turns the Turtle X number of degrees in a clockwise direction

- LEFT X, turns the Turtle X number of degrees in a counter clockwise direction

- PENDOWN, makes the Turtle draw

- PENUP, makes the Turtle stop drawing

These commands make up the essence of Turtle programming and the functionality is also present in the other languages which has implemented Turtle programming, maybe using different keywords. Some languages has expanded on these commands e.g. in Scratch, one can change the color, size and shade of the pen. The commands can be part of user defined functions. Examples could be functions called *SQUARE* or *TRIANGLE* which would draw a square and a triangle respectively using the commands shown. These functions can be part of other functions, e.g. a function called *HOUSE* would use a mix of the commands for correct positioning and then the *SQUARE* and *TRIANGLE* functions to draw the house itself [7].

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

---

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

that for each iteration makes a circle showing where the car have been and then moves the car forwards to a new position. In each iteration he also increases the distance that the car moves forwards, meaning that the distance between the circles becomes greater and greater, thus visualizing the car accelerating [8].

Small Basic

Small Basic is a text based programming language with its whole purpose being teaching novices to program. It is not meant as a language one should keep using, but as a tool for learning programming principles and then "graduating" to learn more advanced languages. In the developers internal trials they have had success with teaching programming to kids in the ages of 10 and 16, but it is intended for novices in general, so it is not specifically made for that age group [9]. It is perhaps one of the last languages with this purpose that is still being updated[2]. It seems as if the focus of teaching programming to novices has changed to visual programming languages, tools for learning existing languages that is meant to be used professionally or languages that are novice friendly, but still meant to be used professionally.

The Small Basic project consist of three pieces, the language itself, an IDE and libraries. This means that one has to use the bundled IDE to program in Small Basic. According to their FAQ [9], the language takes inspiration from an early variant of BASIC but is based on the modern .NET Framework Platform. It is much smaller than Visual Basic, it consists of 14 keywords, and supports a subset of the functionality that Visual Basic .NET supports. It does not have a type system and all variables are global and always initialized as to avoid confusion regarding scopes. It is also imperative and do not use or expose beginners to the concept of object orientation.

To learn programming with Small Basic its website provide a tutorial for getting familiar with the language and the IDE [10], and a curriculum, which can be downloaded for offline use, which teaches general programming topics using Small Basic [11]. Through the introduction one will learn that there is to different window types ones program can be run in, either the *TextWindow* which is a regular console, or in the *GraphicsWindow* where graphics can be drawn and so on. So it is possible to create e.g. games in Small Basic. Turtle programming is also supported.

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

---

[2]With that being said there was a four year hiatus between version 1.0 and 1.1
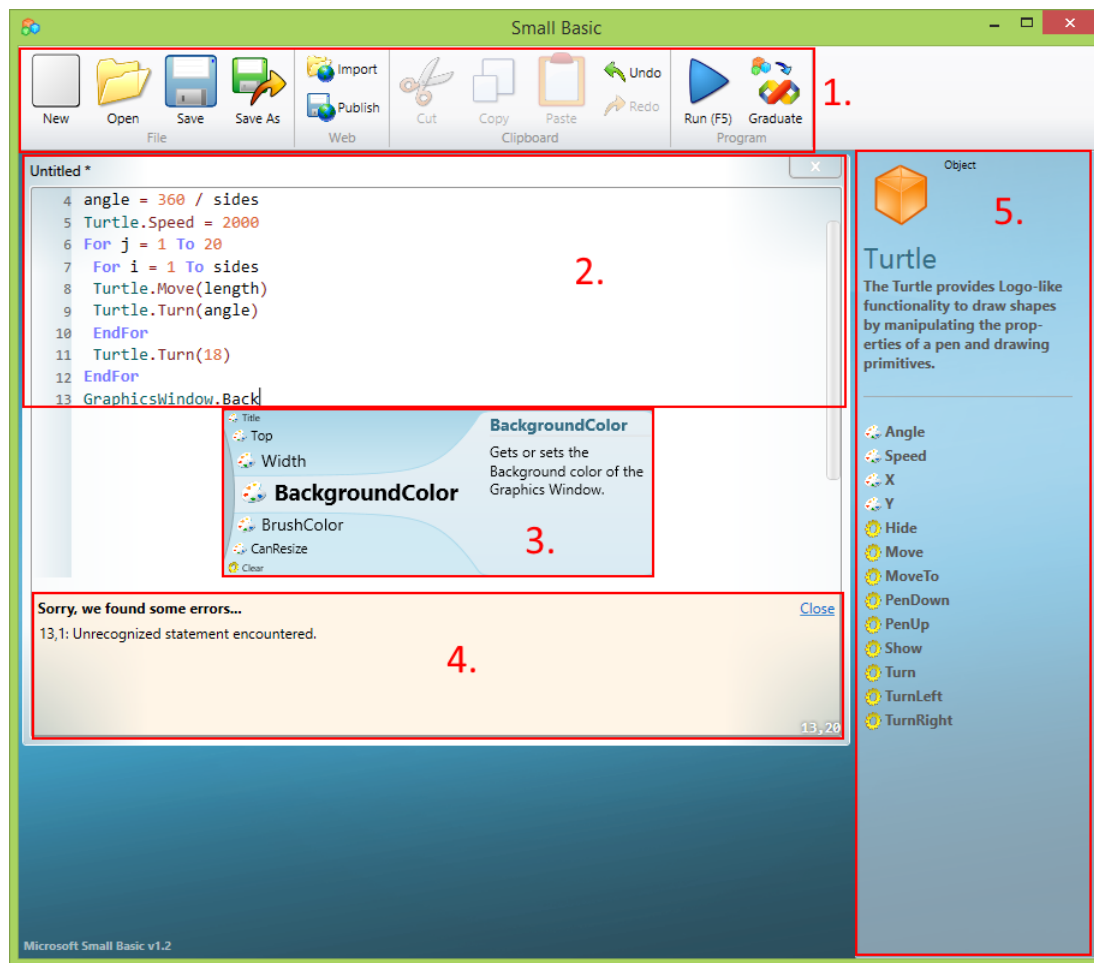
Figure 3.1: Small Basic IDE

3.2.2 General Purpose Programming Languages

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

Programming Languages

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

Another language that was originally designed largely with students in mind was Pascal. It was released in 1970 and aimed to teach students structured programming. It was all that, but some early adopters used it far beyond the original intent. This resulted in a lot of work on Pascal and it evolved, both the language itself but also into different dialects, which ended up being designed more towards end-user use rather than an educational tool. Although this meant that the language and dialects became very popular and widely used. The language itself and the dialect called Delphi/Object Pascal is still used today according to [12].

A third language that was designed and is still maintained as an easy-to-use general purpose programming language is Quorum. Quorum is what the developers of it call 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 hosts 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" [13]. TODO: Tools for learning e.g. BlueJ

TODO: Add missing sources!

TODO: Remember to summarize text compared to visual programming.

TODO: transition to visual programming

TODO: Why is it the goal to learn these languages? (Look into Intentional Programming)

TODO: Why can't we stick with the educational ones?

TODO: Why don't we begin with these?

TODO: Good thing about text based, is terminology, one learns to talk about programming, e.g. what is a statement?

Chapter 4

Teaching Programming

TODO: Chapter about the problems in teaching programming - analysis and discussion

Part II

Language Comparison

Chapter 5

Preliminaries

TODO: Introduction to the experiments. Perhaps another name for this chapter.

Chapter 6

Criteria

TODO: Criteria for the comparative analysis. How the analysis should be made, how to compare, why to do it like so etc

Chapter 7

Comparative Analysis

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

Chapter 8

Results

TODO: Results and discussion.

Part III

Bibliography

Bibliography

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

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

[3] S. W. Koitz Roxane, "Empirical comparison of visual to hybrid formula manipulation in educational programming languages for teenagers," *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, pp. 21–30, 2014. 3.1.1

[4] MIT Media lab, "Scratch statistics." `https://scratch.mit.edu/statistics/`. Used: 28/09/15. 3.1.2

[5] R. N. S. B. E. E. Maloney John, Resnick Mitchel, "The scratch programming language and environment," *Journal Name*, vol. 10, no. 16, 2010. 3.1.2

[6] S. Papert, *Mindstorms: Children, Computers, And Powerful Ideas*, p. 218. Basic Books, Inc., 1 ed., 1980. 3.2.1

[7] S. Papert, *Mindstorms: Children, Computers, And Powerful Ideas*, pp. 11–15. Basic Books, Inc., 1 ed., 1980. 3.2.1

[8] A. Kay, "Alan kay shares a powerful idea about teaching ideas." `https://www.youtube.com/watch?v=JDpsXWuedVc`, 2014. Used: 26/09/15. 3.2.1

[9] Microsoft, "Small basic faq." `http://smallbasic.com/faq.aspx`. Used: 26/09/15. 3.2.1

[10] Microsoft, "Microsoft small basic, an introduction to programming." `http://download.microsoft.com/download/9/0/6/90616372-C4BF-4628-BC82-BD709635220D/Introducing%20Small%20Basic.pdf`. Used: 26/09/15. 3.2.1

[11] Microsoft, "Small basic curriculum: Online." `http://social.technet.microsoft.com/wiki/contents/articles/16982.small-basic-curriculum-online.aspx`. Used: 26/09/15. 3.2.1

[12] T. Software, "Tiobe index for october 2015." `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`. Used: 26/09/15. 3.2.2

[13] Qourum, "Epiq 2016." `http://quorumlanguage.com/epiq.php`. Used: 26/09/15. 3.2.2