



TECHNISCHE UNIVERSITÄT
CHEMNITZ

**Exploring Children's Independent Programming
and Debugging Skills Through an Unplugged and
ScratchJr Programming Course**

Master Thesis

Submitted in Fulfilment of the
Requirements for the Academic Degree
M.Sc.

Dept. of Computer Science
Chair of Computer Engineering

Submitted by: Fränzi Kühn
Student ID:
Date: 07.11.2023

Supervising tutor: Prof. Dr. Janet Siegmund
Supervisors: Elisa Madeleine Hartmann

Abstract

Technology is now part of our daily lives. However, unlike other STEM courses, programming courses are not yet part of the early school curriculum. Early learning can benefit later skills, and learning to program is not excluded. However, while programming tools for children are available, courses that focus on learning and exploring *unplugged* and *block-based* programming and debugging are not yet widely available. This thesis addresses this gap with a programming course for children focusing on programming and debugging with *ScratchJr* and inspired by it, *unplugged*. The study investigated what knowledge is transferred between the two types of programming and how behavior in the two types of programming compares. Debugging is also analyzed in terms of how children react to errors, what debugging strategies they develop, and how their age plays a role in debugging.

Through qualitative research it was found that *unplugged* programming and debugging were suitable for the course, but the method used to teach *ScratchJr* needed more polishing. Furthermore, the knowledge transfer was successful and the children were eager to explore programming and debugging on their own. In addition, four debugging strategies were identified.

Keywords: Children's Programming, Children's Debugging, ScratchJr, Unplugged Programming

Contents

Contents	3
List of Figures	6
List of Tables	7
List of Abbreviations	8
1. Introduction	9
2. Theoretical Background	11
2.1. Programming	11
2.1.1. Unplugged Programming	12
2.1.2. Block-based Programming	14
2.2. Debugging	15
3. Related Work	19
3.1. Research Questions	21
4. Experimental Planning	22
4.1. Pre-Course Procedure	22
4.1.1. Pilot Studies	22
4.1.2. Participants	22
4.1.3. Experimental Design	23
4.2. Programming Course and Material	23
4.2.1. Overview	23
4.2.2. Material	23
4.2.3. Procedure	30
4.3. Transcription and Analysis Procedure	32
5. Results and Discussion	34
5.1. Answer to How do children react to programming errors, and are there differences between their own and implemented errors?	34
5.1.1. Aha! Moment	35
5.1.2. Describe Behavior	35
5.1.3. Questioning	36
5.1.4. Logical Reasoning	37

CONTENTS

5.1.5.	Reaction to Given Bugged Program	38
5.1.6.	Problems Due to the Tablet	39
5.1.7.	Conclusion to How do children react to programming errors, and are there differences between their own and implemented errors?	40
5.2.	Answer to How does programming and debugging compare in <i>un-</i> <i>plugged</i> and <i>block-based</i> programming?	41
5.2.1.	Pair Programming	41
5.2.2.	Thinking Aloud	43
5.2.3.	Testing	45
5.2.4.	Debugging	46
5.2.5.	Mental Simulation and Embodiment	49
5.2.6.	Backseat-Programming	50
5.2.7.	Distraction by the Tablet	50
5.2.8.	Other Interesting Things	52
5.2.9.	Conclusion to How does programming and debugging compare in <i>unplugged</i> and <i>block-based</i> programming?	54
5.3.	Answer to Is there a transfer of knowledge from <i>unplugged</i> to <i>block-</i> <i>based</i> programming?	56
5.3.1.	Conclusion to Is there a transfer of knowledge from <i>unplugged</i> to <i>block-based</i> programming?	58
5.4.	Answer to What debugging strategies do children use when none have been taught?	58
5.4.1.	<i>Unplugged</i> Programming	58
5.4.2.	<i>ScratchJr</i> Programming	59
5.4.3.	Conclusion to What debugging strategies do children use when none have been taught?	60
5.5.	Answer to Are there differences in debugging between different age groups?	60
5.5.1.	Conclusion to Are there differences in debugging between dif- ferent age groups?	62
5.6.	Further Exploration	62
5.7.	Limitations	65
5.8.	Lessons learned	66
6.	Conclusion and Future Work	68
7.	Acknowledgments	69
	Bibliography	70
A.	File Share Code	78
B.	ScratchJr Programmed Stories	79

CONTENTS

C. Feedback Sheet	81
D. Percentage Calculation	82
E. Original Quotes	83
D. Declaration of Authorship	98

List of Figures

2.1.	Examples of different types of <i>unplugged</i> programming	13
2.2.	<i>Scratch</i> and <i>ScratchJr</i> side-by-side comparison	15
2.3.	Introductory <i>block-based</i> and <i>text-based</i> programming program on the Blockly website [29]	16
2.4.	Error locating technique model by Katz and Anderson [38] derived from Morris and Rouse [54]	17
4.1.	Example of <i>ScratchJr</i> commands [22]	23
4.2.	All cardboard command cards sorted by category: a) program start, b) control and state commands, c) action commands, d) motion com- mands, e) sound commands, f) program end	25
4.3.	A <i>step-by-step guide</i> card and an identical task card with a program that has intentional errors in it	26
4.4.	Four examples of <i>activity card</i> tasks	27
4.5.	Left: <i>ScratchJr</i> introduction, Right: Example of a <i>ScratchJr</i> Story Guide	28
4.6.	Three roles as cards with their tasks: robot, compiler, programmer . .	29
4.7.	Outside programming environment for <i>unit 1</i>	30
4.8.	Inside programming environment for <i>unit 2</i>	31

List of Tables

4.1. Overview of the <i>units</i>	24
5.1. Quotes from types of Pair Programming Behavior	42
5.2. Overview comparing programming and debugging between <i>unplugged</i> and <i>ScratchJr</i> programming	55

List of Abbreviations

CS Computer Science

CT Computational Thinking

STEM Science, Technology,
Engineering and Mathematics

UI User Interface

1. Introduction

In today's environment, programmed technology has become an essential part of our daily lives. If one just thinks about it now, a few things certainly come to mind, such as a ticket vending machine, a navigation system, or games and applications on the phone.

It is impossible to ignore the impact of Computer Science (CS) and Computational Thinking (CT) flowing into day-to-day life [76, 65]. This knowledge is no longer limited to professional programmers, as it has become more widespread in other fields. As a result, it is important to recognize the importance of CS and CT in our lives and strive to make it more common knowledge [81].

This led to the rise of the need for more qualified people with knowledge about programming concepts and CT, and encouraged some CS-unrelated degrees to incorporate beginner programming courses to teach a better understanding. Though with programming knowledge, experience is much more important and that takes time. Long before that, students have the opportunity to participate in school programming courses. In Germany, however, these are integrated into the school curriculum only in higher grades [73]. Unlike mathematics, as another Science, Technology, Engineering and Mathematics (STEM), which is already introduced in elementary school to build a knowledge base to be refined over the years in school.

The introduction of beginner STEM courses in elementary schools brings benefits for children and there is no argument against it [20]. Children can already experience more advanced topics such as technology, physics, history, etc. through interactive exhibits in museums or workshops. The same can be done with programming. Being exposed to programming projects at a young age could lead to interest and positive attitude towards CS fields later in life [84]. Furthermore, the acquisition of a programming language can be similar to learning a new language [12, 13] and the elementary school age could be advantageous for it [69]. Some other benefits that children could benefit from early learning programming include problem-solving skills, planning skills, communication skills, and learning to work together [76, 61, 7, 14, 25, 27, 17, 18].

The classic programming languages are text-based. Due to children's development and the fact that they are still learning, reading and writing programs can be difficult. However, over the years, some age-appropriate programming methods have emerged: *block-based* programming languages and *unplugged* programming, which are discussed in chapter 2. This makes it possible for children as young as four to experience and apply programming [61].

While programming can be done in different ways, one of the most important skills during programming remains the same: debugging. Although debugging is

1. Introduction

often associated only with CS, it can be argued that the problem-solving aspect of it is beneficial in other fields [41]. Even the most knowledgeable expert programmers still make errors [64] and need to debug them. For novices it is more challenging because they are still learning programming and have limited experience. However, learning to program, will not provide a good approach how to debug [68]. Debugging is something to encounter and learn by yourself.

The objective of this thesis is to investigate what strategies elementary school children use when identifying an error in their code in different programming environments. Furthermore, their strategies when debugging their programmed code and if these strategies change between programming settings. In addition, I investigate whether there are differences between grade levels in detection and debugging strategies.

2. Theoretical Background

The purpose of this chapter is to explain relevant theories and key words that are critical to understanding the following thesis.

2.1. Programming

In modern years, technologies have become more available for younger children who are not even able to read [15], however, programming for children is not a recent development. The first programming language for children was *LOGO*, which was invented in 1966 and is still used by researchers [70]. The *LOGO* language differs notably from “classical programming languages” with its natural English verbs and nouns, and then a few years after its invention, the well-known *turtle* was developed as a digital robot controlled by simple directional words [70]. With the rise of personal computers, *LOGO* designers developed a visual programming language for younger children to discover programming without the need to read or write [70]. With newer technologies and research, other barriers to programming for children have been identified: keyboard and mouse handling [31] and lack of immediate visual feedback [25, 27, 45]. The development, ideas and design of *LOGO* programming for children started the area, and now most *block-based* programming for children is based on it [70].

For this reason they share similarities [31, 25, 85]. Instead of *text-based* languages on a computer, children’s programming tools, software and toys use minimal or no text. Depending on the target age, programming for younger children is done with less fine-motor skill-based functionalities, such as programming on touch-based screens, drag-and-drop with elements snapping together, and physical elements.

The goals of programming are also different. In “classical programming”, the programmer solves an abstract problem by thinking about how to achieve the goal by connecting programming concepts in a logical way. For children, abstract thinking is difficult [83] and, as mentioned earlier, immediate feedback from their program is important for them to understand and connect actions to commands [61]. Consequently, the start and end goal of the programming (task) is easily understandable and comprehensible for children [31, 14]. For this reason, the tasks are often to control a character or figure from a starting point to the end. The steps are made visible. The figure that does this can be different: a digital character, in physical programming a normal figure that is placed by hand, a robot that is programmed directly or someone who acts according to the program.

There are many ways for children to experience programming, but two main cate-

2. Theoretical Background

gories, or a combination of them, are most commonly used: *unplugged* (or physical) programming and *block-based* (or graphical) programming.

2.1.1. Unplugged Programming

Unplugged or physical programming describes the absence of a computer during the programming process. The programming concepts are decoupled from writing code and instead broken down into the idea and understanding behind it [10, 61]. *Unplugged* programming has two different forms: building a program or an activity to understand CS concepts through a game. To shortly discuss *unplugged* activities: The games are designed to let people passively engage with the concepts. The learning happens by following the rules of the game with an explanation afterwards. The advantage is that the concepts are less abstract through the activity [71].

However, the other variant of *unplugged* programming, and most relevant here, is programming and executing it without the need of an electronic device. Programming elements such as commands are translated into the real world objects to interact with [52]. There are different ways to make this possible: buttons [31, 49] (cf. Figure 2.1(b)), card-like objects to connect through proximity [25, 85, 48] (cf. Figure 2.1(c)), interlocking with one another [14] (cf. Figure 2.1(d)), or on a board [61] (cf. Figure 2.1(a)). These objects can be various materials such as paper, wood, blocks, etc., as long as they can form a program [52]. The physical program is then executed by a person or an inanimate object. This makes a difference with the level of *embodiment* which will be defined later in 2.1.1. This means that the person will execute every command by themselves or with a figure. Thus, the interaction does not end with pressing a button to make the program run, but by taking on the role of executing the program, the person gets a different perspective on the program.

In research *unplugged* programming is used when programming with young children as the programming is more understandable for them as it is concrete [9].

Hybrid

As an extension to *unplugged* programming, these blocks can be translated into a computer program, i.e. *hybrid* programming [71, 40, 39, 14, 36, 37]. In this way, the programming part remains concrete, but is translated into visual feedback, e.g. by the robot following the program, or by being displayed as a *block-based* program. The way the program is transferred can be different, directly like the buttons on the Beebot [75], through commands that are able to connect and transmit the program to the robot [63, 37] or by having the computer program identify the physical program through a picture [42].

Embodiment

Cognitive science's theory of *embodiment* describes how physical interaction with the environment is important for cognition. [5]. By doing something active with

2. Theoretical Background

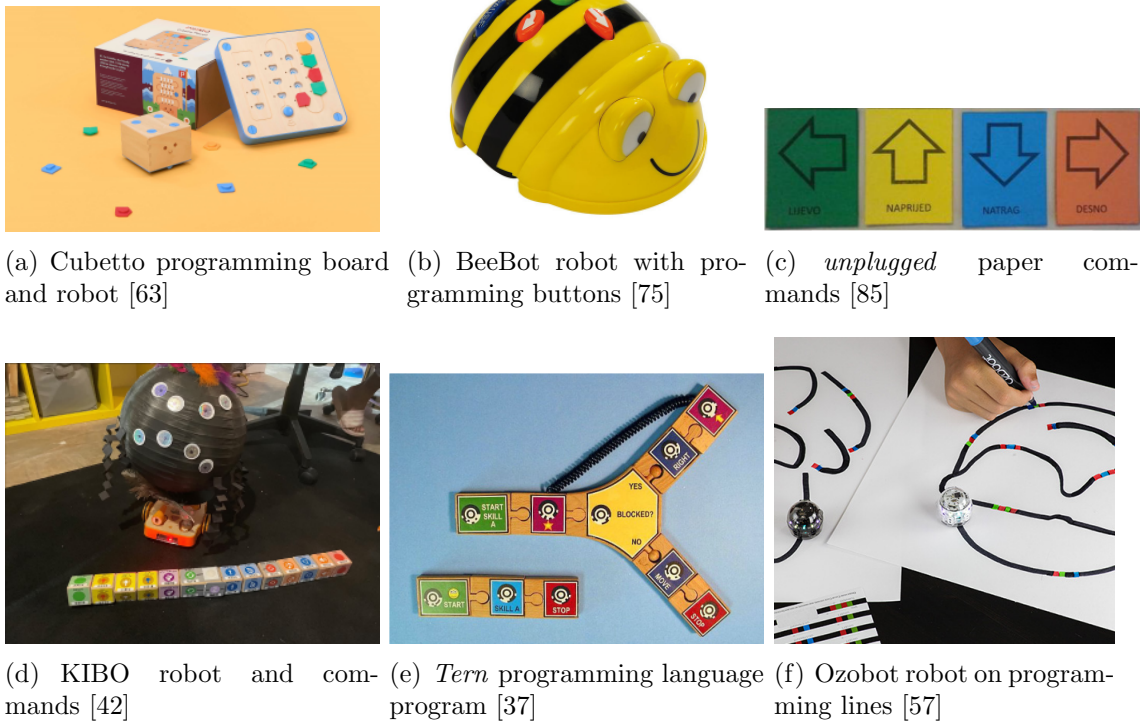


Figure 2.1.: Examples of different types of *unplugged* programming

the body, thinking becomes conscious and understanding deepens [5]. Even more complex thinking, such as problem-solving, can be aided by physical interaction in the real world. According to Clark [19], the body can be the connection for the mind to shape the environment around us and thus “simplify and transform internal problem-solving”. *Embodiment*, however, does not end with the mere use of the body. As Clark [19] notes, technology or other tools can be used and interacted with “to extend and augment cognition” [43]. Similarly, according to Papert [59], technology, or more specifically, computers, can be a powerful tool to enhance thinking and thus enhance learning.

As it stands, *embodiment* can come in different types: acting by themselves or through something. Both can be categorized, as Sung et al. [74] report, as *full* or the latter, *low embodiment*. According to Sung et al. [74] *full embodiment* in the context of problem-solving refers to a person who uses and moves the whole body. As for *low embodiment*, the person uses only the hands or a “surrogate” such as a figure.

These theories of embodiment are not only for the fully developed cognition, but universal even for children. The theories of *constructivism* by Piaget [62] and derived from it, *constructionism* by Papert [58] are both focused on the learning of children and share beliefs with the theories of *embodiment*. For Piaget, an adult or a child does not just change their mind and thinking, but would only change

2. Theoretical Background

it if they experienced or acted with their environment [1]. Teaching something without letting children experience it will not make much difference, instead they will construct what they will learn based on what they already know and what they experience by interacting with their environment [1]. In Papert’s theory of *constructionism* he emphasizes that children need to do something actively to gain knowledge and that this knowledge can also be formed depending on the context [1].

2.1.2. Block-based Programming

Graphical programming or also *block-based* programming has its origin in *LOGO* [70] and as a result, implementations of *block-based* programming share similarities in the way they work, their characteristics and the idea behind what they teach [53].

It is very different from the “classic programming” experience. Building a program is done quite literally, by selecting the right block from a palette [78] or visible selection. Through dragging and dropping the blocks into the editor where the program is built, blocks snap horizontally or vertically to each other. This makes the *block-based* programming languages simple, as syntax errors are prevented by default [53, 78]. The combinations of blocks work with visual cues [78], through the shape of the blocks makes it clear where they can be attached. A good example of this gives one of the most popular *block-based* programming language: Scratch [11], see Figure 2.2(a). The color indicates the type of block, and the shapes of the blocks and slots indicate their connection. Similar to physical programming 2.1.1, *block-based* programming blocks are designed with minimal to no text, depending on the target age.

Directly related to the well-known Scratch [11], is ScratchJr [22]. It was designed by Flannery et al. [27] with the goal of having a programming learning tool for children who are not yet confident with basic reading, writing and arithmetic. The target age group is from five years old to the second year of school [22]. The blocks are without text and also the interface is designed with self-explanatory symbols. The programming goals of both *Scratch* and *ScratchJr* are undefined. Instead of controlling a figure in a 2D maze or field with a fixed starting point and goal, both give the freedom to program digital characters, objects, and any inserted thing in a scene. To make it easier to use, the programming options are reduced in *ScratchJr*. The main focus is on animating own stories with given or drawn characters in one to four 2D scenes. In these cases, the child programmer or adult mentor will need to plan the story and animation themselves or look up the example tasks, as there are no predefined tasks within *ScratchJr*.

With this simplicity of *block-based* programming in early childhood, however, the question arises as to how well what has been learned can be transferred to “classical programming”.

2. Theoretical Background

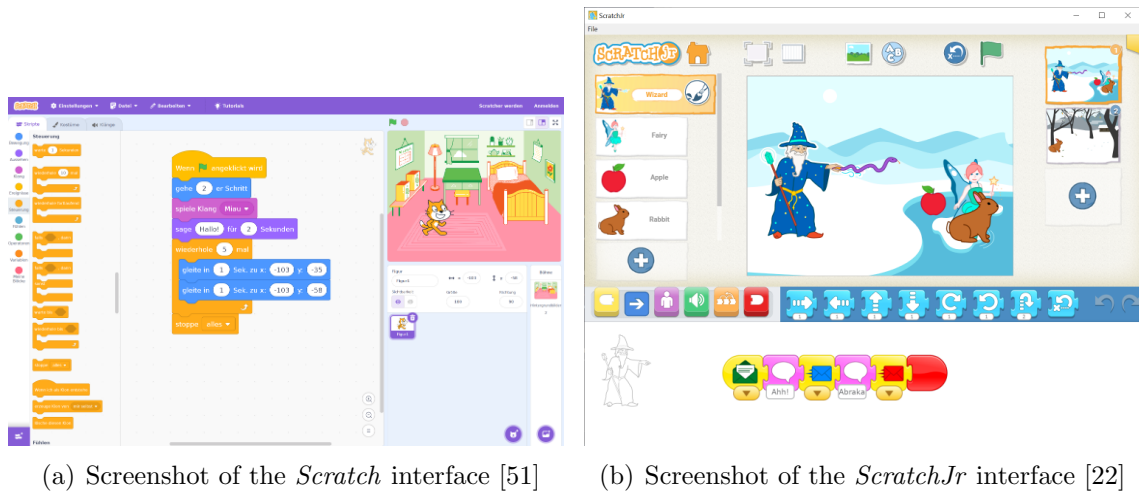


Figure 2.2.: *Scratch* and *ScratchJr* side-by-side comparison

Hurdles of Block-Based Programming

While *block-based* programming is a good way for young children to interact with the world of programming, there is often discussion that the transition becomes more difficult for them. Although programs such as *ScratchJr* and *Scratch* make the introduction to programming easy, they can lead to poor programming practices and difficulties when starting *text-based* programming [53, 78]. In the work of Moors et al. [53] they argue that the simplicity of *block-based* programming, which makes it so good as an introduction for beginners, can cause problems down the line. The main argument is the lack of syntax when programming with blocks. With blocks that only snap to the right blocks and commands that are easy to remember because of the colors and shapes, students can be overwhelmed with new information when learning the importance of syntax when programming with *text-based* languages. As a result, students may lose confidence in their own ability to program [53].

However, using *block-based* programming as a good starter tool to learn about programming concepts [8] and as a stepping stone to motivate children and generate interest to move on to other programming projects can have a different effect [53, 84].

To overcome this difficulty, some *block-based* languages can switch between block and text editors. By switching, the block or text commands are translated into the other programming type (see Figure 2.3). An example of a programming language with this capability is *Blockly* [29].

2.2. Debugging

Programming something that will work correctly the first time is rare. When writing program code, most programmers, from novice to expert, spend a good deal of their time debugging [35, 47]. Debugging is the process of noticing that the program is not

Try Blockly

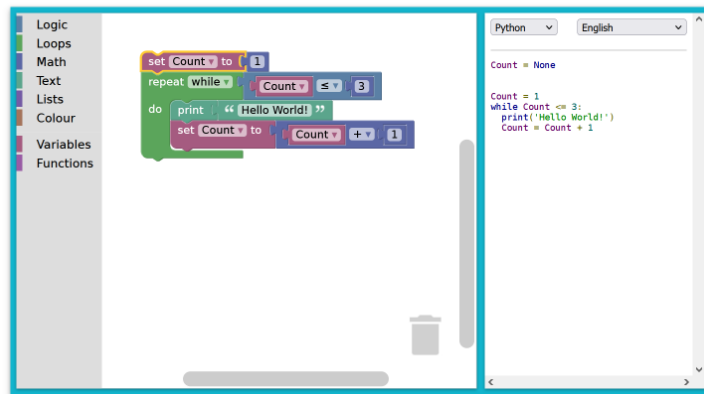


Figure 2.3.: Introductory *block-based* and *text-based* programming program on the Blockly website [29]

working as intended, searching for the source of the faulty behavior, and finding an appropriate repair [6]. This “problem-solving activity” [34, 32, 7] is not only specific to programming, but is useful in many STEM fields and in general [41, 14]. When debugging, the programmer spends more time identifying the error than figuring out how to fix it [38]. This is true for both novices and experts, but experts are slightly faster at this process [32]. However, debugging is not a skill that is simply acquired through programming, but must be actively learned and practiced [38]. Conversely, being good at debugging also benefits and leads to better programming [2], as learning from errors can benefit the learning process [3, 14]. During debugging the programmer needs to have programming knowledge to understand the program [2, 58] but also problem-solving skills [44].

Debugging has many connections to problem-solving [34, 32, 7], so Katz and Anderson [38] took the troubleshooting characterization of Morris and Rouse [54] and applied the model to their debugging techniques (see Figure 2.4). According to these techniques, the programmer must first understand the program. This is automatically granted by writing the program themselves, but if the program comes from another source, they have to become familiar with it. Then it is tested. If the program produces incorrect output during testing, the error must be located and fixed.

Klahr and Carver [44] analyzed the debugging process, specifically how to locate the error, and identified 5 phases into which debugging can be divided: *program evaluation*, *bug identification*, *program representation*, *bug location*, and *bug correction*. The first phase is to realize that the program does not work as intended by testing it. The next phase describes getting a general idea of the error by observing the behavior of the program in order to narrow down possible errors. In the following phase, the programmer builds a representation of the program, if not written by the programmer themselves. When working on an own program this is easier, because

2. Theoretical Background

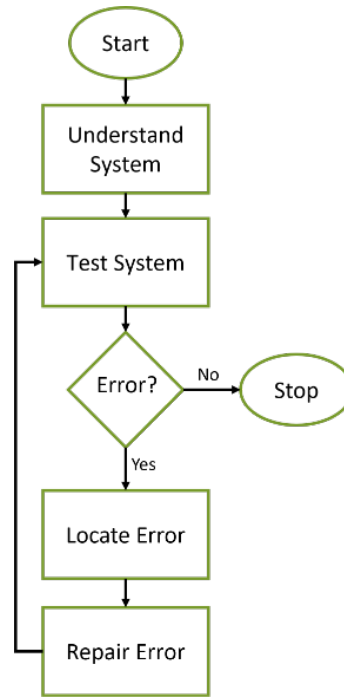


Figure 2.4.: Error locating technique model by Katz and Anderson [38] derived from Morris and Rouse [54]

this phase happens during the programming itself [38]. However, when debugging an unfamiliar program, this knowledge must be acquired. Throughout the previous phases, the programmer has collected clues about the error in the program to narrow down the possible locations. If the clues are not sufficient to localize the error, the programmer must resort to inspecting the program step-by-step for the error. The final phase of debugging is simply to fix the error. Four of these phases have to do with actually finding the error in the program.

Debugging Strategies The model of debugging stages is kept simple to demonstrate the process behind it, but there are many different strategies that programmers utilize during debugging. According to Klahr and Carver [44], good debugging skills cannot simply be learned by programming, but must be taught.

As mentioned before, debugging mostly consists of searching for the error in the program. There are a number of strategies that have emerged for finding the error. However, it is important to note that these strategies are different from tool-assisted debugging strategies. They focus on finding the error without the help of the integrated help in the programming environment.

Katz and Anderson [38] discovered three debugging strategies in their study of students: *simple mapping*, *hand simulation*, and *causal reasoning*. Using *simple mapping*, the programmer starts by looking at the faulty behavior of the program and follows the clues to the location of the error. In professional programming,

2. Theoretical Background

this strategy is also called *backwards reasoning* [79]. By using *hand-simulation*, the programmer executes the program like a computer, looking for the point at which the program's behavior deviates from the expected behavior. When using *casual reasoning*, the programmer tests parts of the program more thoroughly, gathering clues and information about the program along the way to narrow down the location of the error.

In their research with children, Klahr and Carver [44] identified and taught debugging strategies: *brute force*, *serial search*, and *focused search*. However, they also found that some children tend to stick with the bugged program instead of fixing it. During the *brute force* strategy, the programmer does not collect clues to narrow down and locate the error, but checks everything for correctness in no particular order until it is fixed. In the *serial search* strategy, the programmer starts at the beginning of the program and checks it for correctness until the error location is found. In the *focused search* strategy, the programmer limits the search to some places in the program. The clues collected beforehand decide where and which locations are searched.

3. Related Work

One of the earliest studies of children’s debugging is Klahr and Carver [44], in which they defined the debugging process in phases (see section 5.2.4). In order to determine the thinking and thus the debugging, the children were asked to *think aloud*, i.e. to voice their thoughts. In addition, they prepared predesigned programs with built-in errors to ensure that children were involved in the debugging process. In their study, they taught children a debugging strategy of *focused search* with a specific question to lead to the error in the program. Previously, children used *brute force* and switched to the taught strategy along with debugging before the actual debugging process, i.e., before testing and realizing an error.

Many modern studies of programming and debugging with children refer to the study of Klahr and Carver [44] and still use their debugging phases.

Programming and debugging studies with children always depend on age, development and education. However, especially with younger children up to elementary school age, the preferred methods are those described in section 2.1, such as *block-based* programming with a character or simulated robot [21, 56], a robot in combination with *unplugged* programming [68, 50], or both one after the other [4]. Because the children often have no previous experience with programming, most start with teaching the basics and programming or debugging strategies [4, 21, 56, 68].

Debugging Strategies Through their studies, the researchers found various debugging strategies that children used without being taught or trying to *brute force* to debug the error. With the exception of a few, children use *brute force* or *trail and error* in the beginning before moving on to a more effective strategy [56]. A common strategy is *delete and rewrite* [4, 50, 56], in which the children discard their faulty programs and start over. Other often used strategies are *replace error* and *compensate error* strategy [4, 56]. During the *replace error*-strategy, the children searched for the error and replaced the faulty commands with working ones. On the other hand, in the *compensate the error* strategy, the children did not replace the faulty commands with new ones, but added commands to the program until it worked as intended. Another debugging strategy is *verbal debugging*, in which children express errors in the program and what needs to be changed before physically doing it, or then leave it unchanged and start a new task [50].

Some of the strategies found are more specific to a programming method. By using the provided *debugging tools* [68] or *build-in debugging tools* [21] the children used them in their debugging strategy. In Sipitakiat and Nusen’s study [68], they prepared debugging tools for *unplugged* robot programming: flags to mark errors

3. Related Work

in the program and a protractor to help children understand the angles at which the robot must turn. Misirli and Komis [50] had two options for children to program: buttons on the robot or what they called *pseudocode* (*unplugged* programming cards). Some children chose to debug the robot by laying out the *pseudocode* and debugging it before repeating the actions on the robot. DeLiema et al. [21] focused their study on children’s self-reflection after programming sessions and found and categorized several factors that helped children debug: self- and emotional reflection, social support, changing perspective and using prior knowledge, coding tools, and testing.

Some of the children’s observed strategies begin before debugging during programming, which helps them avoid errors. Similar to *verbal debugging* during the *verbal programming* strategy, children plan their program before implementing it [50]. The factor of constructing the program disappears and the children can concentrate on planning alone. Another error-preventing programming strategy observed is *step-by-step* programming [50, 68]. Used in maze tasks, children plan the next step ahead before adding the right command(s) to the program.

The studies showed that not only the actual programming process and debugging strategies are important to focus on, but also the planning and (error) reflection phases are crucial [21]. By pointing with their finger, the children showed planning behavior on their own initiative [56]. At the same time, when working in groups, children used this advantage to plan programming and debugging with group members or teachers [21].

Error prevention strategies such as *step-by-step programming* helped children synchronize their thinking with the robot’s movements, which helped them detect errors during program execution [68]. While using this strategy, children barely need to start the debugging process [56]. Similarly, Ahn [4] found by comparing *unplugged* programming and debugging with a control group. The *unplugged (embodiment)* groups performed better at debugging, linking it to synchronized thinking with body movements.

Larger programs are more challenging for children [68, 56]. Only then children started to use *debugging tools* more often [68]. When errors in programs increase, children had problems to distinguish single errors and consequential errors [56, 68]. In case of multiple errors, children used the *bottom-up approach* to solve the errors in the program [56].

Nikolos et al. [56] observed that the first process of debugging, locating the error or realizing that the program is not working as intended, is often brief, and is only noticeable when children use verbal expressions such as “uh-oh”.

DeLiema et al. [21] observed in their study that children use more than one strategy to debug. Similarly, Nikolos et al. [56] argue that children would come up with more strategies if they had to focus only on debugging and not on programming beforehand. They suggest giving children ready-made programs with errors.

3.1. Research Questions

The current state of research on children’s debugging addresses error detection and debugging strategies in a particular type of programming: *unplugged* or *block-based*, and focuses on debugging in that particular mode. Klahr and Carver [44] have looked at the transfer of debugging strategies when used as problem-solving in a programming unrelated task. However, this concept is not represented in more recent research or with a different programming type.

Often, children are introduced to debugging strategies before they can develop their own, so that they do not start with a blank slate when they first begin debugging. Because children have no prior programming experience, researchers observe children debugging their own errors rather than giving them complete programs with errors to debug, as Nikolos et al. [56] suggested.

As a result, I identified the following research questions, which are the focus of this master thesis:

- How do children react to programming errors, and are there differences between their own and implemented errors?
- How does programming and debugging compare in *unplugged* and *block-based* programming?
- Is there a transfer of knowledge from *unplugged* to *block-based* programming?
- What debugging strategies do children use when none have been taught?
- Are there differences in debugging between different age groups?

4. Experimental Planning

In this chapter, the experimental design and the materials for the programming course that were created for the purpose of answering the research questions will be explained and summarized.

4.1. Pre-Course Procedure

4.1.1. Pilot Studies

Prior to the study, a small pilot study was conducted in two sessions. In the first session, the audio devices, programming in general, the cardboard commands, the outside programming environment, and an earlier version of the tasks were tested with four volunteer children. Based on observation and feedback, changes were made such as the new and final step-by-step tasks, the changed programming perspective, and as a new addition, the role cards. For the second session, three children volunteered to test an earlier version of the programming course with the modifications from the first session. As a final adjustment, the roles became more defined and tasks were assigned to them.

4.1.2. Participants

The participants of the study are children who chose to participate in a children's programming course at the after-school day care center Großstädteln in Markkleeberg. In three consecutive time slots, six children from first to fourth grade can be enrolled at a time. The first time slot is for first and second grade, the second for third and fourth grade, and the third for all school grades. One child who participated in the pilot study attended the course in all time slots.

Parents and children were informed in advance about the programming course through a notice in the after-school day care center. The notice informed that it is a study. Children who enrolled were given a data protection information, participant information and information about the audio recording, and for everything, a consent form for their parents and guardians. The children were provided with participant information in an appropriate language that they could understand. The participating children received no compensation other than attending the programming course.

The study and the documents for the consent were reviewed and approved by the Ethics Committee.

4.1.3. Experimental Design

The study was decided to be a qualitative research with audio data that will be transcribed afterwards. The data collection includes semi-structured interviews conducted during the course to gain insight into the children’s thinking and problem-solving process during programming and debugging. At the end of the programming course, a feedback sheet in the form of a questionnaire is given to the children to fill out. At all times during the course, observations are documented by the supervisors.

4.2. Programming Course and Material

The study is conducted over several dates and uses different materials for the lessons (from here on *units*). The programming and these materials have been previously tested in a pilot study (4.1.1). Due to the large amount of material used in the study, all of these can be found in a linked repository rather than in the Appendix¹.

4.2.1. Overview

The following Table 4.1 summarizes all of the *units*, their objective, and the material used. It also links to the corresponding section.

4.2.2. Material

Cardboard Command Cards

Command cards are used in more than one unit and come in two versions, a large and a table size version. The shape, color and symbol design is based on *ScratchJr’s* [22] digital commands, see Figure 4.1. Although only some of the commands have been preselected with a fixed amount based on fictional programs. The reason for the strong similarity is that the transition to *ScratchJr* is planned in later *units*.

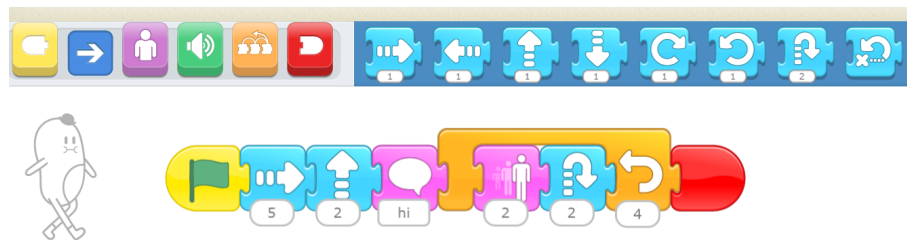


Figure 4.1.: Example of *ScratchJr* commands [22]

All cards have the same puzzle piece shape except for the *start*, *end*, and *repeat* commands, see Figure 4.2. This puzzle shape is intended to make it easier to build

¹All of the materials can be found at https://drive.google.com/drive/folders/1z0DEMKZ504A5U0sTe9AF2yIYrYRrIpiL?usp=drive_link, see. Appendix A for QR code

4. Experimental Planning

Table 4.1.: Overview of the *units*

Unit	Unit Objective	Programming environment	Material
Unit 1 (4.2.3)	Introduction to rules and commands	Outside, chalked playing field on the ground with programming area next to it (4.7)	<i>Step-by-step guide</i> cards (4.2.2), Cardboard Command Cards (4.2.2), <i>Unplugged</i> Role Cards (4.2.2)
Unit 2 (4.2.3)	Learn the missing new commands and debug a program with implemented errors	Inside with two groups, at a table with a small playing field on a blackboard and a programming area next to it (4.8)	Table Command Cards, remaining <i>Step-by-step guide</i> cards (4.2.2), <i>Bonus Tasks</i> i.e. program with implemented errors (4.8), Playmobil figure, chalk
Unit 3 (4.2.3)	Introduction to <i>ScratchJr</i> UI and commands	In two groups with one tablet each with <i>ScratchJr</i>	<i>ScratchJr</i> introduction, <i>ScratchJr</i> story guides (4.2.2)
Unit 4 (4.2.3)	Apply knowledge from previous <i>units</i> and program a <i>ScratchJr</i> story	In two groups with each one tablet with <i>ScratchJr</i>	Empty story sheet and an example (4.2.2)

the program with the commands, because visually the commands only interlock in a certain way. *Start* and *end* commands have only one connector for the same reason. The *repeat* command consists of two parts that are larger than normal commands to fit them inside.

There are six colors to symbolize a category of commands: yellow (program start), red (program end), blue (motion commands), green (sound commands), purple (action commands), and orange (control and state commands).

The symbols on the commands differ slightly from *ScratchJr* because they had to be handmade and therefore simpler. The main difference is the lack of input boxes on almost all commands, but especially on *movement commands*. All movements are meant to be just a single move. The *repeat* commands have a field for a number of repeats.

4. Experimental Planning

Some commands have been modified to be more suitable for a real live environment. The *ScratchJr* action commands *hide* and *show* are replaced with *disguise* and *remove disguise*. The sound commands are also replaced with predefined sounds: *clap* and *stomp*.

All cardboard command cards are made of colored 220 g/qm cardboard and the symbols are made of self-adhesive vinyl paper. The finished assembled commands are laminated to protect them from the outside environment and to make them easy to clean.

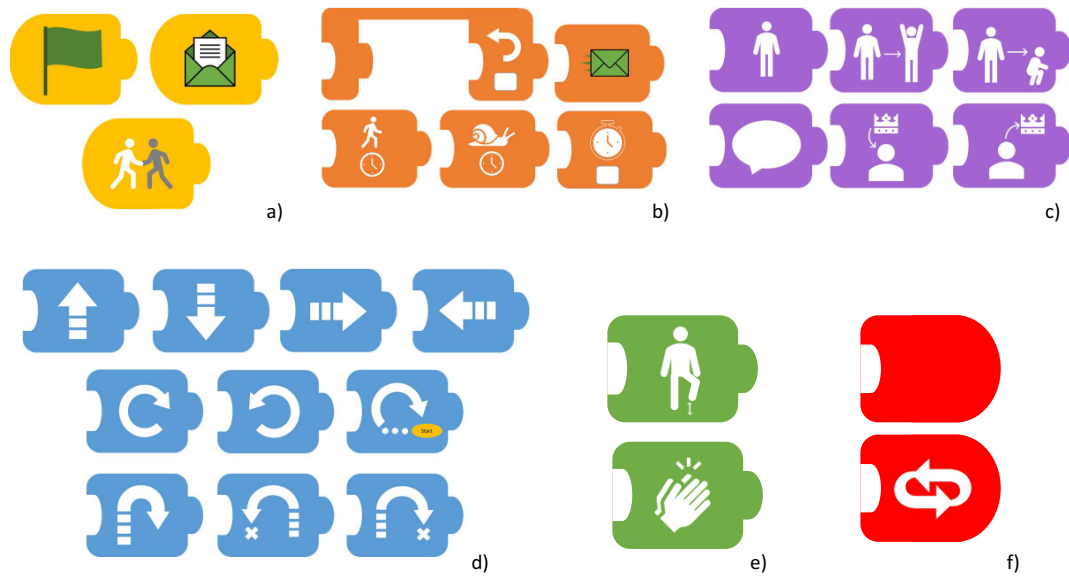


Figure 4.2.: All cardboard command cards sorted by category: a) program start, b) control and state commands, c) action commands, d) motion commands, e) sound commands, f) program end

Robot Adventure Cards

The *unplugged* tasks are designed as an adventure of a robot that wants to reach its goal with the commands given by the task. There are two versions of those task cards. The larger one for the first *unit* is a *step-by-step guide* and meant as a progressive introduction to programming and new commands. The *activity cards* are the second version and are for the second *unit*. They assume that all commands are known, they are different tasks and they are playing card size. In both variants,

4. Experimental Planning

there are tasks that involve finding and debugging errors in a intentionally bugged program.

Step-by-step Guide The *step-by-step guide*¹ contain tasks to build a program using specific commands, and new commands are introduced step-by-step, see Figure 4.3. The cards are numbered up to 17 because they build on each other. For the study, only a few crucial cards are selected due to time constraints. Each card, see Figure 4.3, has a title (1), a written task (2), an overview of the playing field (3), and the commands given for this task (4). The first eight titles and written tasks are simple descriptions of the program and its intent. The others have a story theme based on the story “Alice in Wonderland” to make some of the challenges in the tasks easier to explain. The overview of the playing field is a representation of the layout of the real playing field. The start, end, obstacles and other objects are marked in it, but not a predetermined path.

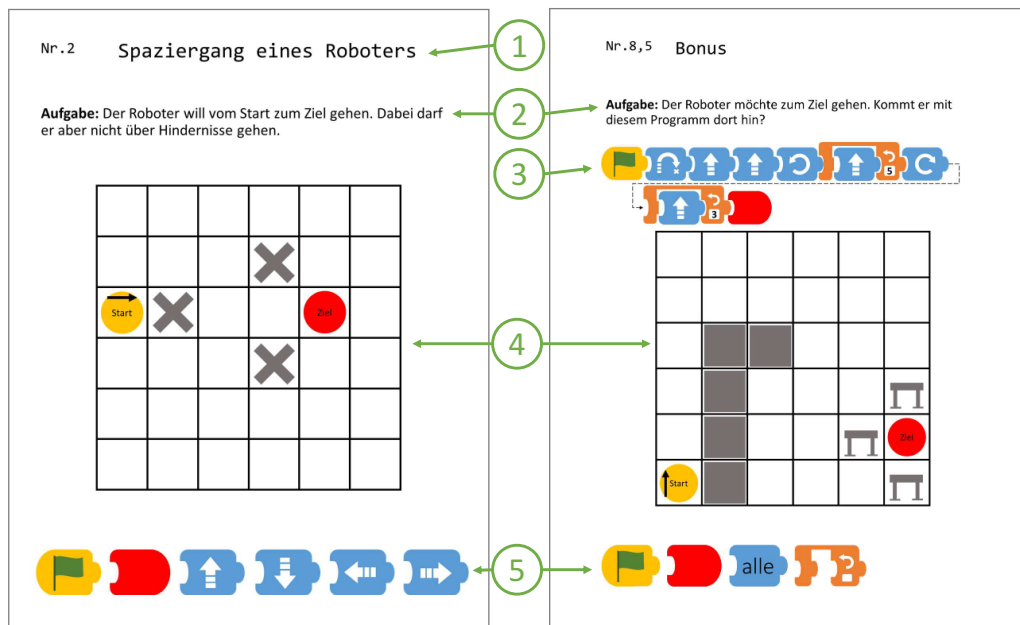


Figure 4.3.: A *step-by-step guide* card and an identical task card with a program that has intentional errors in it

Bonus Cards In between the *step-by-step guide* are ready-made bugged programs called *bonus cards*¹, see Figure 4.3. The task is to look for errors in the given program. These programs consist only of commands from the previous *step-by-step guide* and contain two to five errors. These tasks are designed to force error detection and debugging.

4. Experimental Planning

Activity Cards The *activity cards*¹ are designed as playing cards that can be turned over to reveal the programming task, much like a card game, see Figure 4.4. The task on them are different and it can be one of the following ones: *Build a program!*, *Build a playing field!*, *Where is a command missing?*, *Debug the program!*, and *Where is the goal sign?*. The tasks *Build a program!* are similar to the tasks on the *step-by-step guide*, and *Where is a command missing?* are programs with intentional errors to debug. The difference on the *activity cards* is that the title is missing, the written tasks have no story theme, and all commands are allowed. It is assumed that all important commands in the second unit are known from the *step-by-step guide* or could be easily explained during the *unit*.

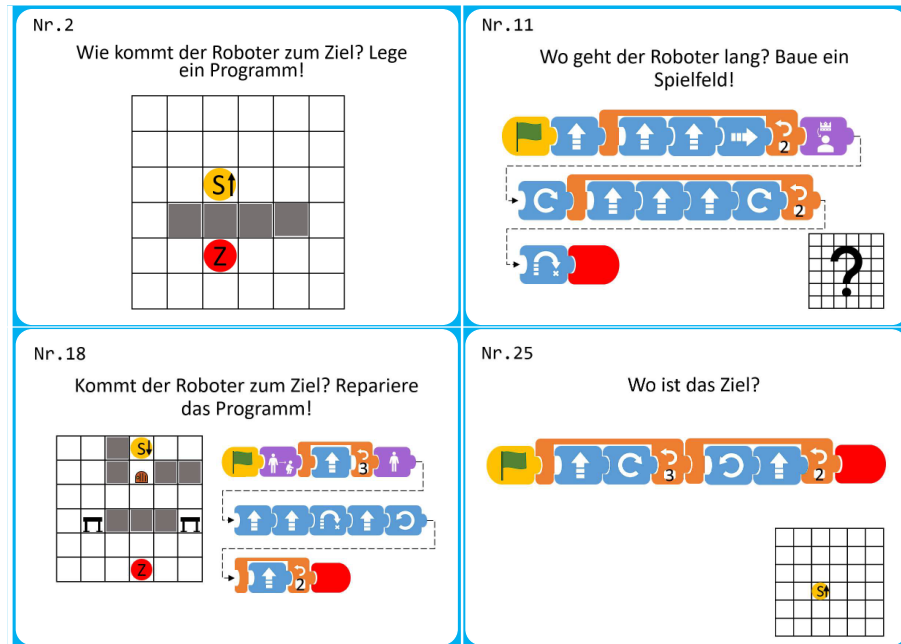


Figure 4.4.: Four examples of *activity card* tasks

ScratchJr Programming Tasks

In the *ScratchJr units*, the tasks change and become more independent. They focus more on seeing how children transfer the knowledge from the previous *units* and provide a guide to follow along.

ScratchJr Introduction This introduction contains four ready-made programs to be analyzed before programming in *ScratchJr*, see Figure 4.5. Two of them contain two *ScratchJr characters* to demonstrate the interaction of *characters*. This is a method that can provide an insight into the transfer of knowledge.

ScratchJr Story Guides The *ScratchJr story guides*¹ provide a starting point for children to slowly get to know *ScratchJr* and its UI and functionalities, see

4. Experimental Planning

Figure 4.5. The guides are a mix of story and tasks to build the story step-by-step with multiple *characters*.

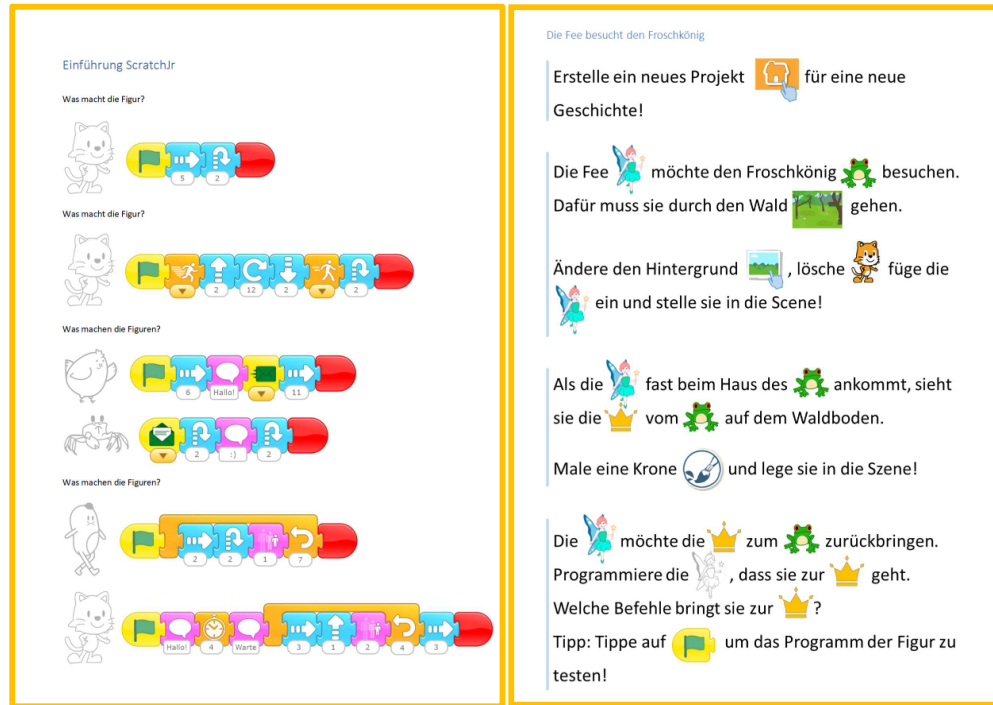


Figure 4.5.: Left: *ScratchJr* introduction, Right: Example of a *ScratchJr* Story Guide

ScratchJr Programmed Stories In the fourth *unit*, children in their group write their own story and then exchange it so that the other group has to program it. To guide them, they are first given an example and then a similar-looking template to fill in with their own story, see Appendix B. The example is prepared on the tablets.

Role Cards

As an addition based on the pilot study 4.1.1, children can have one of three roles per program: a *robot*, a *compiler*, and three *programmers*, see Figure 4.6. Both *robot* and *compiler* build the playing field as specified on the task card. During program execution, the *robot* listens to the commands read by the *compiler* and follows them around the playing field. The *programmers* build the program and watch for errors as the program is executed.

Programming Environment

The first *unit* takes place outside, see Figure 4.7. A large six-by-six grid is drawn on the ground with chalk, and the programming area is defined next to it. Each grid is

4. Experimental Planning

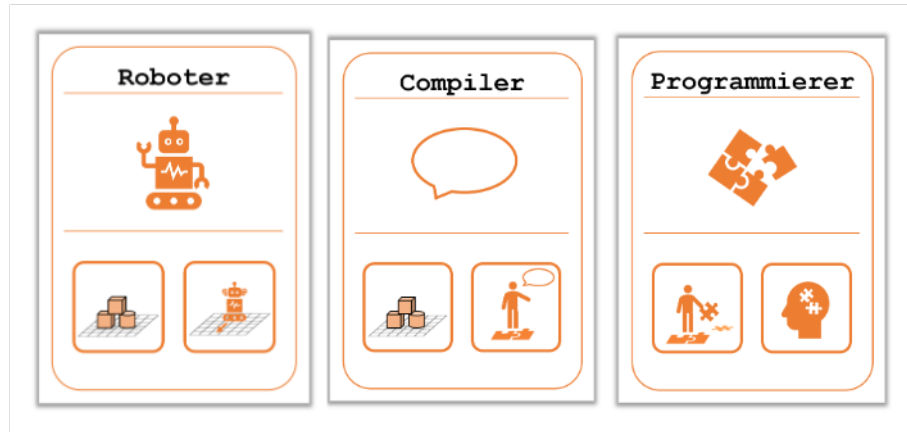


Figure 4.6.: Three roles as cards with their tasks: robot, compiler, programmer

large enough for a child to stand in. Obstacles and other objects are represented by real-world objects available on site or brought in by one of the study supervisors. If no suitable object is available, the obstacle or object is drawn with a pen on paper and placed on the field.

The second *unit* is done indoors at tables. The six by six playing field is drawn with chalk on a 50 by 35 centimeter blackboard placed on the table, see Figure 4.8. Obstacles and objects are also drawn on the board or on paper and then placed on the board. The programming area is next to the blackboard on the table.

For the *unit 3* and *4* participants are given tablets. The tablets run *ScratchJr*, where all the programming takes place.

Other Material Used

To represent the start and end points on the board, there are yellow and red cardboard circles, which are also laminated. For outdoor use, the circles are the same size as the cardboard commands, and for indoor use, they are smaller to fit on the board.

The second *unit* is a miniature of the first, and to represent the *robot* there are Playmobil figures.

The third and fourth *units* are on an iPad and an Amazon Fire tablet running the latest version of *ScratchJr* (1.3.0).

After the programming course it is planned to get feedback from the children, see Appendix C. The questions are about what they think about each *unit*, the perceived difficulty of the course and the duration of the *units*. Open-ended questions ask the children to state what they liked best, what they liked least, if they would participate again, and if there is anything else they would like to say about the programming course.

4. Experimental Planning

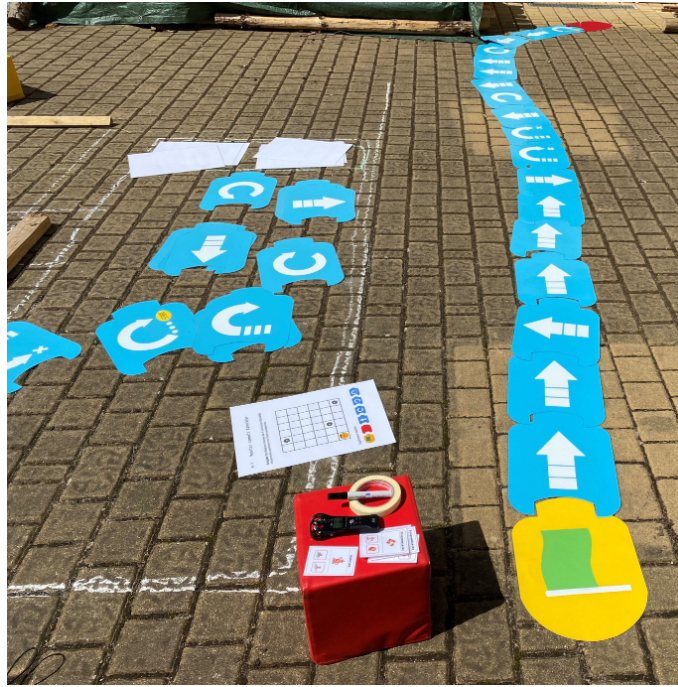


Figure 4.7.: Outside programming environment for *unit 1*

Experimental Equipment

The data collection is done by recording the audio through two recording devices, Zoom H1n and Zoom H2n.

In the *units* with the tablets, the screens are recorded. The iPad has a built-in screen recorder, and the Amazon Fire tablet screen is recorded using XRecorder, an application available from the Google Store.

4.2.3. Procedure

The *units* are planned in two parts, one *unplugged* with the cardboard material and the other *block-based* with *ScratchJr* on tablets. Six children participate in each *unit* for 45 minutes. There is a 15 minute preparation time between each group. Since the time is more limited than originally planned, supervisors review the unit after each one and modify the next one accordingly.

Unit 1

Before the *unit* starts, the audio recorders are placed, one in the programming area and the other behind the playing field.

The first *unit* serves as an introduction to the rules and commands, so it has many explanations. Programming is introduced as a game with rules and playing cards. After the first explanations, the children draw *role cards*. Then a *step-by-step*

4. Experimental Planning



Figure 4.8.: Inside programming environment for *unit 2*

guide card is drawn. The task is discussed and new commands or components on the playing field are explained. Each child has a role and does the task until the playing field and the program are finished. Then the *compiler* positions themselves behind the program and the *robot* on the playing field so that they cannot see the program. The program is then executed. If an error is found, the *programmers* fix it and the *compiler* and *robot* start over. Otherwise, if the program was correct and its execution is finished, the playing field and the programming area are cleared for the next task.

After the first playthrough, the explanations are reduced, but the procedure remains the same. New cards are drawn until the end of the *unit*. If there is time left, the children can come up with their own program and build it until the end of the 45 minutes.

The time of the *unit* was much more limited than expected, so not all crucial *step-by-step guide* cards were completed.

Unit 2

The second *unit* takes place inside with the programming environment on two tables for two non-competing groups, with the downsized commands and the same tasks on each table. One supervisor sits at each of these tables with the children, and there is an audio device recording. Everything, the programming, the tasks, the commands, the rules and the roles is the same but smaller to fit on the table.

As recap a short review of the last unit is done with both groups. As the *step-by-*

4. Experimental Planning

step guide cards could not be completed in the first *unit*, those were used as tasks instead of the playing card-like *activity cards*. The first task acts as an introduction. Then one program with implemented errors is presented. The remaining time the rest of the *step-by-step guide* cards are completed until the *unit* is over.

Unit 3

The children are again split into two non-competing groups at two tables, each with one tablet with the screen recording and an audio recording device. Before working in their groups, the children are asked if they have used a tablet before and then the *unit* procedure and *ScratchJr* are explained. It is assumed that *ScratchJr* is new to most or all of the children and is introduced as a tool for programming a story. During the programming, a supervisor sits with them but only offers help when directly asked, as the children are encouraged to explore more independently.

At first, the *ScratchJr introduction* is given, in which the children analyze and describe the *ScratchJr* program in the task before programming it. At the end of the introduction, the children are asked if they would like to explore *ScratchJr* by programming their own story or using the *story guides* provided. In both cases this is done until the end of the *unit*.

Unit 4

The setup and group work is similar to *unit 3*. Before working in their groups, the *unit* is explained as an applied use of the things they worked on in the last *unit*. Then the children read the *ScratchJr Story Example* and watch the prepared animation on the tablet. The groups then have 15 minutes to plan and write the story. During this time, the supervisors only help with writing and explaining. Then the stories are exchanged and the children program the stories independently. Meanwhile, the supervisors do not sit with the groups, but come in case there are questions, explanations and help needed.

The children have until five minutes before the end of the *unit* and then present their programmed story to the other group. After that, the participants are given a feedback sheet to review the programming course.

4.3. Transcription and Analysis Procedure

After the study was completed, the audio of each *unit* and group was transcribed using a manual transcription tool *oTranscribe* [28] that can be used offline. After transcription, the names of the participants were replaced with a non-identifying code names in the style of *P* for participant with an appended number in the order of the participant list.

For this thesis, the children's quotes are translated from German into English. The original quotes are included in the Appendix E, and all transcripts can be

4. Experimental Planning

found in a linked repository².

As it was stated in Heikkilä and Mannila’s research [34], the transcribing process itself contributes to the analysis and for this reason notes, observations and initial assumptions were made during transcription. Afterwards, the transcripts were analyzed through closed card sorting based on the previous notes and new observations while looking at the transcripts a second time.

Due to ambient noise and incorrect settings on an audio recording device, the audio quality of the programming group was lost during *unit 1* of all groups. Unfortunately, the screen recordings made on the tablets themselves were unworkable, as the videos stuttered and froze on one frame on several different tried media players.

²All of the transcripts can be found at https://drive.google.com/drive/folders/1z0DEMkZ504A5U0sTe9AF2yIYrYRrIpiL?usp=drive_link, see. Appendix A for QR code

5. Results and Discussion

To gain insight into children's thinking and behavior during programming and debugging, a children's programming course was prepared to answer the research questions posed.

5.1. Answer to How do children react to programming errors, and are there differences between their own and implemented errors?

For this question, observations were made on how children react and behave when confronted with errors. In *unplugged* programming, errors in the program would prevent the *robot* from reaching the goal and in *ScratchJr* the characters would behave unexpectedly. In the second *unit*, contact with programming errors is ensured by a given bugged program. In the third and fourth *unit*, the children programmed more independently and errors were expected.

Overall, reactions from children to errors were positive. In no instance they were discouraged when encountering something unexpected. Moreover, there are times when children have been happy to report errors:

"Error, error, error, error!" (P4, unit 2)

"(inhale audibly) We made a mistake! We made a mistake! This should not be here! It has to be here!" (P4, unit 2)

"Immediately a mistake discovered." (P4, unit 2)

"I think I already see a mistake in it." (P6, unit 2)

"The crab stayed in place, haha." (P12, unit 3)

During the *unit* the children stayed in their groups and only in the next *unit* new groups were formed. Due to that it can be assumed that the reactions are natural and not originating from only some children. Analysis of the transcripts revealed common responses and behaviors. These can be categorized into: *Aha! Moment*, *Describe Behavior*, *Questioning* and *Logical Reasoning*. Due to the frequency of incidents with the tablet in *unit 3* and *unit 4*, *Problems Due to the Tablet* is also analyzed, even if it is not technically a programming error. Findings are presented below.

5.1.1. Aha! Moment

This reaction occurred when the children both found an error and immediately found a solution. This reaction is typically audible in expressions such as: “ah”, “oh” and “right”:

“Right! We need another forwards here.” (P1, unit 2)

“Ah, there’s one missing.” (P8, unit 2)

“Oh, we just forgot one forward.” (P13, unit 2)

“Oh my... make it big only(?) twice. Hn.” (P2, unit 3)

“There? Oh... target flag.” (P11, unit 3)

“Huh? He jumps in the air and comes always... ah, he should take off from the ground and then jumps onto the bed.” (P18, unit 3)

“Nah, that’s saying. Ah yes, yes (inaudible).” (P11, unit 3)

“Yes. No wait, that’s right, you’re right. We need seven.” (P16, unit 3)

“Oorh how, ah right!” (P4, unit 4)

“Why does it do that? Oh, the baby is doing it.” (P4, unit 4)

“Wait no, we can delete this one, wait, I have to put it on well mhm. Mhm. Mhmmm. Mm-hmm. Oh. Wait.” (P16, unit 4)

“Ah, there’s nowhere more to go. After that...” (P13, unit 4)

“Ah, that’s right, they don’t actually disappear.” (P16, unit 4)

In problem-solving situations the *Aha! Moment* happens spontaneously, without any solving process beforehand [46]. Recognizing and understanding the issue and gaining insight into the problem comes without the use of a strategy [46]. In addition, by identifying and fixing the error at once the *debugging phases* [44] are skipped. This reaction was more frequent in *unit 2* than the *ScratchJr units*. One explanation could be that the programming was easier for children to understand, that they skipped from the *bug identification phase* directly to the *bug correction phase* [44].

5.1.2. Describe Behavior

A reaction to errors that mainly occurs during programming with *ScratchJr* is to describe the behavior of the program:

“Here, go, three times forward, the ball is taken along somehow. And that is somehow so funny.” (P3, unit 3)

“Ey, he hasn’t even jumped yet!” (P1, unit 3)

“Huh? It’s not coming out of his hand right now.” (P7, unit 3)

“The crab stayed in place. HaHa.” (P12, unit 3)

“He’s not getting any smaller, is he?” (P8, unit 3)

“And where is the ball? It’s not going to be magically removed.” (P3, unit 4)

“Huh? And the fairy is still here.” (P1, unit 4)

5. Results and Discussion

“Yes, but it’s just stupid, because the wizard, he’s... Now it’s gone... Well, it doesn’t work somehow.” (P13, unit 4)

“The lizard is also doing something, but somehow it doesn’t appear here.” (P13, unit 4)

“But there the crown moves by itself.” (P12, unit 4)

“But... Yes, but they’re not supposed to be here. They are supposed to be gone.” (P16, unit 4)

In the debugging strategy *simple mapping* [38] or *backwards reasoning* [79] the programmer analyses the behavior of the program to find clues to debug it. This could be something that children do to help them understand what is happening and to solve the error. Otherwise it might be a form of *verbal debugging* [50] in which children describe what is happening before solving it. However, as will be discussed later, it was more common for errors to be kept in.

As an other explanation, it could be just a symptom of *thinking aloud*, that children freely did when occurring an error.

5.1.3. Questioning

As a result of working together on a program in a group, the children made their questions audible when errors occurred. Often someone would speak up when someone else did something they felt was wrong, or did not understand why something was being done.

“There’s an error, stop! Stop, stop. Why go up twice?” (P2, unit 2)

“Eh, wait a second... Forward. Left?” (P2, unit 2)

“Look, ...(inaudible) jump over then, one, two, then you turn to there?! (emphasis on ‘there’)” (P17, unit 2)

“Eh, shouldn’t we do five times here?” (P2, unit 2)

“Eh, he’s already big, isn’t he?” (P1, unit 2)

“One f- another one to- yes, to one, eh? Huh?” (P2, unit 2)

“That is also wrong, isn’t it?” (P8, unit 2)

“He’s not getting any smaller, is he?” (P8, unit 3)

“Does that thing still have to go in there? Oh, it doesn’t.” (P7, unit 3)

“Huh? It’s not coming out of your hand right now.” (P7, unit 3)

“Why is he jumping now- huh? Why is he jumping?” (P15, unit 3)

“Where is this bracket? Where is it?” (P8, unit 3)

“Eh, how come Putin¹ hasn’t done anything now?” (P18, unit 4)

¹This is a character in the story of the other group.

5. Results and Discussion

By working together, the children used the opportunity to discuss within the group. The advantage of this is that together they help themselves to understand the program better. In the debugging process this is the *understand phase* [38] or *program representation* [44] to get a deeper insight into the program.

In summary, while the children were working on their program they frequently were audible when something unexpected happened, be it from the task, a group member or the execution of the program. Working in a group came as a plus because the children were attentive about things others do. They also often questioned things they did not understand.

5.1.4. Logical Reasoning

In *unit 2*, on many occasions, in finding the error, the children used their experience of the real world environment and implied it into the program:

“From there on. But then he crashes into the wall.” (P2, unit 2)

“Yes, because otherwise we’ll crash into the wall here.” (P1, unit 2)

“When he gets big, he’ll crash into the tunnel.” (P6, unit 2)

“But- but if he makes himself big here again, then he crashes into th’ tunnel wall.” (P1, unit 2)

“That’s r- that’s really possible! (in the door frame of the room, showing something) But it is possible, you can stand in the door.” (P3, unit 2)

“Noooo! Otherwise he’ll crash into the obstacle. Then he has to jump over the obstacle.” (P15, unit 2)

“Well, so that he can go over- through the door. Nnnnn hahaha. Because the door is not standing.” (P5, unit 2)

The playing field on the table got fantasized into a 3D world by all of the children and rules of physics like the *walls*, *obstacles* and *doors* became real for the *robot* and at the same time for them. It could be that the previous *unit 1* helped with practice to know that the *robot* can not just simply fly or jump over objects. This is consistent with the report of Silvis et al. [67] that children are facing the problem to solve not only “the semantic problem (the program) but also the pragmatic problem (the environment)”.

What also helped children while finding the error is simulating the way the *robot* would take often without placing the figurine:

“And, but the third one is not right again. Because if he’s standing here, then, if he’s standing here.... One, two, three is right. But here he has to jump over the field, otherwise he’ll crash into the obstacle.” (P15, unit 2)

“Here... one, two, three, four, five and then forward once doesn’t work.” (P9, unit 2)

“...because if you go up twice then you also miss putting on the crown.” (P4, unit 2)

“I discovered a mistake right away. Because when he grows up, then I put it here that he’ll be normal again in a moment. But then he’s not through the door yet.” (P4, unit 2)

5. Results and Discussion

“Then he already crashes here at the second step. And that’s where the end should be. Here.” (P15, unit 2)

“I would simply say 5 times to the right here. ... Want 1, 2, 3, 4, 5, then he would already be right here. Then 1 more time down, once - skip once and then he’s already at the goal.” (P4, unit 2)

Mental simulation was something children did in many cases other than finding an error, like programming and testing before executing the program.

During *ScratchJr* programming, real world allusions were less frequent, tended to focus on things children visually saw and added just some imagination. It also differed from *unplugged* programming in that it had nothing to do with errors.

“We walk around on the moon.” (P5, unit 3)

“Huh? He jumps in the air and then always comes... ah, he should take off from the ground and then jumps onto the bed.” (P18, unit 3)

“The crab is sitting so nicely in our basketball hoop.” (P4, unit 3)

“But the chicken is supposed to be a spectator.” (P1, unit 3)

“Why is he going through the door like that?” (P18, unit 3)

“He runs into the wall hehehe.” (P15, unit 3)

“Hehehe. No, the dog has to jump out of the window.” (P14, unit 3)

One reason for this may be that, because of the digital nature of programming, it is harder for children to have a sense of imagination. However, it could also be that because *ScratchJr* is greatly visual, there is less need to imagine things.

Children used *Logical Reasoning* because of their implication from the real environment onto the miniature programming environment. The *robot* on the field was not just an object but *he* would execute their program and follow the path, thus *he* is bound on physics. The visual feedback provided by the playing field with the objects and obstacles also helped children planning and coming to a solution. Children benefit from the age-appropriate programming and material [83, 66] and thus it was possible for them to rely on previous knowledge and experience regardless the new setting of programming concepts.

As with digital *ScratchJr* programming, *Logical Reasoning* was not used in the context of error description or explanation.

5.1.5. Reaction to Given Bugged Program

The tasks with implemented errors in *unit 2* had two purposes: to ensure that children encountered an error and to observe how they reacted when the error was not of their making. However, due to time limitations, only one of these tasks could be completed by each group.

Often, however, the children did not read the question in the task or saw the given program, so it had to be pointed out to them. Even after noticing and building the given program, the task was overlooked by most groups, leading to confusion or acknowledgment that this is strange:

5. Results and Discussion

“Huh? No, it doesn’t go any further.” (P4, unit 2)

“What do they want here? (giggling)” (P17, unit 2)

After the supervisors made sure that the groups read the task if the program leads to the goal, most groups treated it with no reaction or answer to the question. However, the program was not simply assumed to be correct, but was built and tested by most groups before proceeding to find and debug the error(s). In addition, none of the groups discarded the bugged program to start over with their own, as has been reported in similar studies, see [4, 50, 56].

Only two groups explicitly answered the task question:

“Wait... Jump, one, two... eh... I don’t think he’ll make it to the goal. He’s already stuck here. Look! One, two, jump this way. Then he goes forward twice. Then he goes that way once and is supposed to go that way five steps. But it doesn’t work! After that he turns like this and then he has to go forward three more steps.” (P15, unit 2)

“But they said we should try it out and see if it works! [...] So it could be that it doesn’t work. [...] No, um, it said - it only said that um, we should try out if it works!” (P5, unit 2)

“And it does not work!” (P4, unit 2)

Due to the fact that the two groups initially paid attention to the question, they started the tasks differently than the other groups, who simply placed the command cards and tested the the program. One of these groups was multitasking, with one still reading the task while the other was looking for commands. This led to a small argument between them:

“We are not allowed to do that. We have to do this!” (P3, unit 2)

“But you can do that too.” (P4, unit 2)

“Hopefully the pro- program will work, otherwise we will have to do our own.” (P5, unit 2)

The other group started to answer the task question without laying out the command cards, just looking at the task card and then *verbally debugging*.

“I don’t think he’ll make it to the goal. He’s already stuck here. Look!” (P15, unit 2)

In summary, it did not make a massive difference whether the error was self-made by the group or intentionally given by the task. The children’s changed behavior was due to the change in the task rather than to errors they did not make.

5.1.6. Problems Due to the Tablet

Technically, these problems or “errors” are not related to programming, but the frequency with which they occur may provide insight into the research question. The problems range from not being able to find a feature, not knowing how to do something, or something not working right away. While these problems were being encountered the children were a little upset.

5. Results and Discussion

“The tablet does not want what we want.” (P3, unit 3)

“Ey, what, you can erase too?” (P4, unit 3)

“How do you actually delete these things?” (P7, unit 3)

“Theeen, is there an eraser here?” (P13, unit 3)

“It won’t! What did you..?” (P15, unit 3)

“Where are we now? Help!” (P3, unit 4)

“And how can you delete the stamps again?” (P6, unit 4)

“We have to delete that. How can you delete it?” (P13, unit 4)

“(whispers) Sooo. (wiping sound and suddenly everyone reacts quite startled)... Oh no, the scissors were on!” (P10, unit 4)

“First we have to name him. Oh, no, you can’t.” (P11, unit 4)

“Hehe and that one I can’t delete. Shit.” (P16, unit 4)

One possibility for the contrasting response to these issues may be that these problems are felt to be genuinely out of control for children, as opposed to implemented errors. The fact that programming takes place on an unmodifiable tablet, rather than with physical objects that can be manipulated directly, may contribute to this changed reaction. In addition, when analyzing the responses during transcription, the change in responses was noticeable not only with problems due to the tablet, but also with programming errors. If something did not work for a while, the children sometimes sounded a little upset or frustrated. This, too, could be explained in terms of the perception of control over the tablet.

“Hehe. The game is stupid. Tablets don’t want as much as we want.” (P3, unit 3)

“Why can’t you delete the one behind it?” (P18, unit 3)

“K where is a K here? Orrrh ey, I hate the keyboard!” (P4, unit 4)

“I can’t find the exclamation mark, though. Eyy.” (P4, unit 4)

“Huh why doesn’t it paint anymore?” (P16, unit 3)

“Yes we want... we are supposed to change the background here, but it doesn’t change.” (P17, unit 4)

To conclude, the responses to errors, no matter where they come from, have changed when interacting with the tablet.

5.1.7. Conclusion to How do children react to programming errors, and are there differences between their own and implemented errors?

There were no differences between the groups in the children’s responses to errors, whether they had the insight to solve the error immediately, analyzed the behavior of the program, asked questions, or tried to implement real-world physics into the *unplugged* or *block-based* programming. In some instance they were happy to have found the error.

When confronted with intentional errors in the tasks, children treat them no differently than they would their own program to build, test, and debug. However, there are slight variations in the reactions when programming *unplugged* vs. in *ScratchJr*. The source of the error did not matter. It may be that errors that occur when programming on the tablet are perceived differently, and this may come from the perception that they cannot control how the tablet does things.

Errors that occur in a learning environment are expected by children, so they handle them without difficulty.

When the settings of both are very similar, the reactions to implemented errors do not differ from those to self-made errors.

Problems and errors resulting from working on the tablet changed the response to errors, as they were perceived as more out of control.

5.2. Answer to How does programming and debugging compare in unplugged and block-based programming?

For this research question, the programming and debugging parts were analyzed regarding the behavior across the different groups and *units*. The programming was designed to be as similar as possible in command design and function and building the program that the programming mainly differed in *unplugged* and *block-based*. Programming and debugging parts were closely intertwined, so *debugging phases* like *testing*, *bug identification* and *bug fixing* were mixed into programming. This was due to the children solving the tasks in an exploratory way, i.e. they also discovered something wrong or an error while programming, which led to debugging before testing a finished program.

While analyzing, some behavior showed up that was repeated often in each of the *units*. Prominent are *Pair Programming*, *Thinking Aloud*, clear *Testing Phases* and scattered *Debugging Phases*, which appeared during all of the *units*. However, some behaviors were specific to *unplugged units*, such as *Mental Simulation* and *Embodiment*, and in *ScratchJr units* *Backseat-Programming* and *Distraction by the Tablet*.

5.2.1. Pair Programming

Pair programming is something a developer team does to bring a novice programmer up to speed when paring with an experienced developer [80]. The new programmer gains some insight about techniques and procedures while watching the experienced person. At the same time pair programming can be done with equal experienced programmers, who mutually help and control each other. While programming during the *units*, the children used something very similar to pair programming in their groups. In the analysis, interactions among the children, such as someone asking

5. Results and Discussion

the group for help with something, someone taking the lead and directing the programming, someone correcting something, and someone explaining something to the other members of the group, were attributed to pair programming (see Table 5.1).

Table 5.1.: Quotes from types of Pair Programming Behavior

Pair programming attributes	Quotes
Asking for help from group	<p>“Can you give me one go down?” (P6, unit 2)</p> <p>“Jumping? No. I don’t know now.” (P5, unit 2)</p> <p>“Huh, he can leave those on, can’t he?” (P8, unit 2)</p> <p>“Step forward, where is step forward here?” (P4, unit 2)</p> <p>“Then... what comes next?” (P11, unit 3)</p> <p>“Three times forward, right?” (P13, unit 3)</p> <p>“Say hello. Where was that speaking?” (P9, unit 3)</p> <p>“Now the end and this- this end?” (P15, unit 3)</p> <p>“And then we need a wizard too, don’t we? Don’t we?” (P2, unit 4)</p> <p>“What else do we need?” (P4, unit 4)</p> <p>“All right, so what does the frog do?” (P4, unit 4)</p> <p>“How many times the (smaller?)?” (P8, unit 4)</p>
Leading and directing group	<p>“And there has to be a get bigger command.” (P6, unit 2)</p> <p>“We need 1 time right and 2 times this orange ones.” (P4, unit 2)</p> <p>“So a 2 has to go in here...” (P4, unit 2)</p> <p>“Wait. No, that was just right the way you laid it. Like this. And then step forward.” (P11, unit 2)</p> <p>“That’s what we need! We need that! We need that!” (P17, unit 2)</p> <p>“Why don’t you put the red one there?” (P3, unit 3)</p> <p>“Like this. We still need these in blue, we need these in blue.” (P12, unit 3)</p> <p>“So. And now we still have to do the crab.” (P12, unit 3)</p> <p>“I think it’s on this one. Yes. Can I? Now you can do it.” (P9, unit 3)</p> <p>“Now you. And put five up there.” (P8, unit 3)</p> <p>“Just write another 1.” (P18, unit 3)</p> <p>“We still have to do the red wizard quickly.” (P13, unit 4)</p> <p>“So we need an Eskimo. (exhales) I know how you erase. You need to go back.” (P18, unit 4)</p>
Correcting group	<p>“You don’t need to, because you’re standing in the doorway.” (P18, unit 2)</p> <p>“Uh, we don’t need that.” (P15, unit 2)</p> <p>“No, he has to turn around first. Clever guy.” (P18, unit 2)</p> <p>“Wait. No, that was just right the way you laid it. So.” (P11, unit 2)</p> <p>“There it is. Only the blue ones.” (P3, unit 3)</p> <p>“No you don’t need that!” (P13, unit 3)</p> <p>“You mean the bracket.” (P7, unit 3)</p> <p>“For the chicken, we first need to do the beginning.” (P15, unit 3)</p> <p>“Eh hee? But this is a green letter, not a yellow one.” (P14, unit 3)</p> <p>“Why did you actually make two babies now?” (P4, unit 4)</p> <p>“Oh, [P8]. We just remembered, we have, no, there’s no...” (P13, unit 4)</p> <p>“Okay, here we do one, oh I’ll do three, we need three here.” (P11, unit 4)</p>
Explaining to group	<p>“Look, we’re- Look. Here is the start. Then up. Put on the crown.” (P1, unit 2)</p> <p>“Um, don’t have to go up there again. Because look...” (P2, unit 2)</p> <p>“Jumping on the spot... Because the jumping over field is this one.” (P15, unit 2)</p> <p>“Yes, but look, here it’s one, two. And here you have to do one, two, three.” (P15, unit 2) (P3 and P6, which are not programming, count the placed commands aloud) (unit 3)</p> <p>“First of all, he has to get smaller.” (P1, unit 4)</p> <p>“Exactly. But snail’s pace is not yet. Look, if you click them yet, they will become even faster.” (P3, unit 4)</p> <p>“No, this is where we have to take them! There. There, no.” (P14, unit 4)</p>

5. Results and Discussion

Children did benefit from working in a group and programming together. They helped each other, asked for help with searching and finding the right commands, corrected each other and sometimes someone took the lead and said what has to be done. Pair programming can be faster for simple tasks or more efficient for more complex tasks [33]. Both types could be present, as asking for help to find something or correcting someone was faster than searching or later fixing the error. Another result of pair programming is that by communicating with their group members, children gain a deeper understanding of what they are doing [16]. This refers to the need to follow what others are doing, thinking about it, and using terms and concepts previously unknown to them (e.g. commands), or making up their own that the others understand (cf. section 5.2.8). Strawhacker and Bers [72] reported similar behavior with “multiplayer programming”, which could benefit social reasoning skills.

While it was also a habit during *ScratchJr* programming, pair programming has slightly declined. Pair programming was more prominent during the introduction to *ScratchJr* at the beginning of *unit 3*, helping the child who was programming by prompting the commands used and in *unit 4* while figuring out how to build the story scenes.

“First the start...” (P2, unit 3)

“So, now up twice. It says jump on it twice.” (P3, unit 3)

“The first thing we need is a start.” (P4, unit 3)

“So now two times up.” (P4, unit 3)

“There must be a 1.” (P7, unit 3)

“But wait a minute here green.” (P12, unit 3)

“At the end we have to do the castle.” (P11, unit 4)

“Stopp, stopp, stopp. Ice desert we must now (inaudible).” (P14, unit 4)

One possible explanation for the decline of pair programming as a collaboration in the group could be that the tablet made it impossible for other non-programming children to chime in and help. Instead, what some were doing was telling the child programmer in an imperative language to do something. This will be addressed in section 5.2.6. Another reason may be that the time they spend programming has also decreased.

To summaries, the children worked collaboratively within their groups, similar to pair programming. It was especially present in *unit 2*, which helped them gain a deeper understanding of the programming concepts presented. During the programming of *ScratchJr*, the communication between the children was reduced in comparison to *unit 2*, and there was more frequent use of a command tone.

5.2.2. Thinking Aloud

Thinking aloud is way to “verbalize their thoughts” to better understand someones problem-solving process [24]. During the programming children used it often without

5. Results and Discussion

being reminded by one of the study supervisors. These are just a few examples of how children formulated their next action(s) in *unit 2*:

“(quietly) *Two* (inaudible). (louder) *Two to.*” (P2, unit 2)

“*To... this... Then... this...*” (P8, unit 2)

“*Eh, we need ri... No, wait a moment. Making small... yes. Step forward...*” (P13, unit 2)

“*No wait he can... ah yes right... He has to...*” (P17, unit 2) “*Oops, first you have to... hat has to go there. Or?... Nope.*” (P15, unit 2)

“*Two steps ttt-to. Two steps to the right... no, left... And right... Then...make yourself very small, please.*” (P14, unit 2)

This kind of *thinking aloud* are forms self-talk found in younger children during problem-solving activities [82]. Both forms, *overt* (private speech) and *covert* (whispers and muttering) [82] occurred here when children were programming, which research found to be an indicator for “guiding, planning and regulating their own thinking and behavior” [82].

This appeared more often during *unit 2*. While working on the tablet children used *thinking aloud* less frequent in *unit 3* and barely in *unit 4*.

“*4 to... So first... Start.*” (P2, unit 3)

“*Then... Wait, we’ll d- we’ll do it together. Fooour.*” (P6, unit 3)

“*Does that thing still have to go in there? Oh, it doesn’t.*” (P7, unit 3)

“*First we need the...*” (P4, unit 3)

“*Um... So. And there-Ne. Yes and then speak again. Speech bubble.*” (P12, unit 3)

“(quit) *Jumping.*” (P8, unit 3)

“*That- and now uh uh uh...*” (P14, unit 3)

“*Sooo. Theeeen... it goes 6 times...*” (P15, unit 3)

During the few times someone *thought aloud* they were the one programming on the tablet. This becomes more evident when looking at the transcript of *unit 3* where one child decided to work alone:

“*Ahhh... Hmmhmm... This is the one for now. That’s right. This is this... Now this... two to (inaudible), that’s already... And now the end.*” (P16, unit 3)

“*Ah, meeting... Hmm... Two up... So, two uppp. Um... Gosh.*” (P16, unit 3)

“*1, 2, 3, 4, 5, 6, 7, 8, 9, 10, (begins to whisper) 11, 12, 13, 14, 15, 16, 17... 17 boxes.*” (P16, unit 3)

“*Umm... hmhmhmhhmmmm... puuuhhh.... so... now comes the people.*” (P16, unit 3)

“*Soo. Nah, one bigger. Ffffff, hm, hm, make bigger.... (Inaudible)... Mhm, mhm, mhm, mhm... mhm, mhm, mhm... mhm, mhm, mhm... nah, nobody can sit on that. Ffffff... Perfect, and now this has to go, like this. And now I’ll look at it again, or wait.... Beginning...*” (P16, unit 3)

One explanation could be that the tasks in both *ScratchJr units* were designed to be more independent and without a set goal. As a result the problem-solving from task to finished product was not as strict as in *unit 2*. However, another explanation may be that programming occurred less frequently on the tablet because children chose *ScratchJr* projects and programs that did not require problem-solving. Another reason could be the distraction caused by the features in *ScratchJr*. This will be picked up in section 5.2.7.

To summaries it *thinking aloud* was often used by children during problem-solving situations. During *ScratchJr* programming this declined and is limited to the child on the tablet.

5.2.3. Testing

In each *unit*, the children tested their programs to see if they would work. This was done automatically after the program was finished, but in some cases the children decided to test parts of the program before an error was found or the program was finished, or specifically while programming in *ScratchJr* to see what a character did.

“Can we first test how it works?” (P3, unit 2)

“Let me take a quick look. (wants to test)” (P17, unit 2)

“We can easily test this ourselves.” (P17, unit 2)

“So now let’s test it.” (P3, unit 3)

“So, now test it. (P3, unit 3)

“We have to see if it works first.” (P3, unit 3)

“That’s (inaudible), we’ll just give it a try.” (P1, unit 3)

“Nah, let it start first.” (P15, unit 3)

“Ah yes, here we have already. We have to go through the story once!” (P9, unit 4)

“Go ahead and run the program already.” (P15, unit 4)

“And now we’ll test it again. So let’s get rid of this for now.” (P16, unit 4)

Children intuitively tested their programs before debugging, in comparison to Klahr and Carver’s debugging phases, the *program evaluation* [44]. In addition, testing was not used as part of the programming, i.e. placing a command and testing how the *robot* or *ScratchJr character* moves and repeating this process. Instead command cards were placed as a sting of actions or the program was completed and then tested. This is a direct difference to the *trial and error* or *brute force* strategy used by children in similar studies, see [44, 56].

As a difference during *ScratchJr* programming, testing was mainly used as a way to see what characters will do. Through behavioral observations and transcript analysis, it appears that children did not always predict in advance which character movements the program would translate beyond the introductory *ScratchJr* tasks.

In summary, tests were frequently used by the children groups as a way to verify their programs or action of a *ScratchJr* character instead of a phase of active debugging an error.

5.2.4. Debugging

Since programming and debugging were not clearly separated as the *debugging phases* [44] or *error location techniques* [38, 54] states, problem-solving approaches were mixed into the programming.

However, there was an apparent contrast between debugging in *unplugged* programming and *ScratchJr* programming. During the *unplugged* debugging, children had no problems to repair the errors once they found them, as it also was stated by Klahr and Carver [44] in their study.

“Oh now we have a problem. Jump forwards not backwards.” (P3, unit 2)

“We made a mistake! This should not be here! It has to be here!” (P4, unit 2)

“There is a step forward missing.” (P9, unit 2)

“And you had that wrong. Either- there should be a step forward.” (P16, unit 2)

“But that’s where we have to make us tall.” (P14, unit 2)

“Because here’s the problem. He wants to go there.” (P4, unit 2)

Even if they were not sure about a fix, they still voiced their guess to their group, which could help because of their pair programming collaboration (cf. section 5.2.1).

“One turn to the riiiiight, no left.” (P5, unit 2)

“Aaaand... and one more forward.” (P8, unit 2)

“Mmh... Then we have to go a bit to the side here.” (P15, unit 2)

“Sooo... I would say jump twice.” (P15, unit 2)

“We meh - we just take this...” (P4, unit 2)

Through visual feedback, the children had an overview of the program and the playing field with the start, goal, and state of the program. This may have helped them with their thinking process and also with debugging, as some children may be overwhelmed when faced with more cognitive load during programming, i.e. multiple errors and longer programs [68, 56, 72].

During *ScratchJr* debugging, children often only voiced the programs or character behavior as described in section 5.1.2. Actual cases of fixing the error were less common. Similar behavior was found by Ahn [4] where children performed better at *unplugged* debugging compared to *digital debugging*.

“Wait t-, let’s just leave it. (error is just left like that)” (P3, unit 3)

“Uhh the stop is missing. And we have to play it slowly.” (P7, unit 3)

“No, that, goal. The goal is missing. The end.” (P9, unit 3)

“You have to- you have to place the end there.” (P13, unit 3)

“Then it has to go one step further.” (P16, unit 3)

“We have forgotten the end.” (P2, unit 4)

“The goal was not there.” (P4, unit 4)

5. Results and Discussion

“That’s, that’s funny. He’s not using the others’. It starts with the... Wait, the lizard shrinks first.” (P13, unit 4)

“Look, there’s always this one. You see, that’s why they disappear!” (P15, unit 4)

“Nah, changing the scene comes with it.” (P18, unit 4)

In these cases, debugging is similar to *unplugged*, pointing out the missing or incorrect command. However, just describing what happened without saying anything about how to fix it or discussing it afterwards was observed more often. Observation provides more insight into this matter; children during these times either stopped communicating with their group members, sought help from a supervisor, or just left it the way it was and did something else (programming or designing characters, see section 5.2.7). The fact that more questioning was done during *ScratchJr* programming and debugging, shows that it possibly was more difficult for the children than *unplugged* programming and debugging.

“Yes, but why doesn’t that go there?” (P3, unit 3)

“Nah, where do I put this?” (P16, unit 3)

“What? What do I have to do now?” (P8, unit 3)

“Here... Maaaaan, can you come and help us out a bit?” (P13, unit 4)

“How do you run to the other screen? How do you switch?” (P8, unit 4)

“Ready... Okay! Start... Oh no, it doesn’t make any seeense! Uh... (signs for help)” (P13, unit 4)

“Oh. Wait. Oh, oh yeah now. Wait. On the white and ah nah there’s something to write on. I don’t know, wait a minute. (signs for help)” (P11, unit 4)

“Um, how can you make the dragon breathe fire?” (P6, unit 4)

“We have to ask again. [...] We want him to call out ‘Show yourself’ after this Northerner, because he’s hiding, but now we’re going to Putin again and there...?” (P18, unit 4)

“Yes we want... we are supposed to change the background here, but it doesn’t change?” (P17, unit 4)

Especially during *unit 3* and *unit 4* it became more obvious. In *unit 3* the children had a choice of what to do in their group, choosing things they wanted and concepts they understood, similar was reported by Bers et al. [14] where children chose concepts that matched their programming knowledge. In *unit 4*, however, they were limited to the story the other group chose and what actions to program. This led to more difficulty, as some groups even chose to write more difficult stories for the other group:

Children trying to make it difficult:

“Do we want to make it difficult for the-” (P1, unit 4)

“It’s going to be difficult.” (P3, unit 4)

“We make it extra difficult for them.” (P14, unit 4)

“With this story, it could be quite difficult.” (P16, unit 4)

5. Results and Discussion

“We are making a difficult story for them.” (P13, unit 4)

Difficulty perceived by children:

“Pffh (exhaling) They chose a completely stupid story. Our story was still fine.” (P9, unit 4)

“Your story is just a bit difficult.” (P15, unit 4)

“Your stories so difficult (inaudible)” (P17, unit 4)

“Yours was also so strangely difficult.” (P18, unit 4)

Moreover, non-communication with the group, or lack of help or participation from members during the times and doing other things, would also indicate an increased perceived difficulty of debugging by the children. This behavior could be related to the problems they had with the tablet, see section 5.1.6, and similarly classify the errors as unsolvable or helpless against.

One other possible explanation could be that the *ScratchJr* programming was more abstract and depended on the imagination of the children to program an action to appear like the action and not to take it literally:

“But how can you magically move the ball? (wonder if there is this command)” (P7, unit 3)

“Is there any magic here?” (P1, unit 3)

“Where is the kissing program?” (P3, unit 3)

“Um, how can you make the dragon breathe fire?” (P6, unit 3)

There was no real goal other than ‘this is what it should look like in the end’ as in *unplugged* programming with an exact goal to achieve.

However, on a positive note, during *unit 4*, when the programmed story was shown to everyone, sometimes the children of the other group helped debug the remaining errors by pointing them out and making debugging suggestions.

“There with this, look when you put these commands in, uh, wait a minute, we go to back and that, then you take the command.” (P13, unit 4)

“Take that away. Take it away. And then... [...] That command.” (P8, unit 4)

“Then you go, you take away the command and that. And take this.” (P8, unit 4)

After the supervisor asked what the error was:

“Well, they did. I think they forgot to put this in. And then it didn’t work.” (P8, unit 4)

“The goal was not there.” (P4, unit 4)

“Well, we just... Well, we forgot to put this in.” (P13, unit 4)

To sum it up, children had an easier time debugging *unplugged* programs than *ScratchJr* ones. This could be due to the visual feedback provided by *unplugged* programming in terms of seeing what is happening. The observed fewer debugging attempts during *ScratchJr* debugging could be due to the perceived difficulty, similar unpleasant experiences with tablet problems (see section 5.1.6), or increased abstraction of programming.

5.2.5. Mental Simulation and Embodiment

Mental simulation is the transfer of a thought into an action [23]. During the *unplugged* programming, children often used it before putting down the commands or to mentally test the way of the *robot*. It can also be a form of *thinking aloud* (cf. section 5.2.2), in which the thoughts are formulated with substitute words for simplification.

“We have to go up. And then so and so (quietly, gets talked over) so and so and down through and so....” (P1, unit 2)

“Or like this here. Soooo, soooo, so, so and so.” (P8, unit 2)

“One, hm hm hop and then hn hn.” (P4, unit 2)

“One step forward, then jump over the field ding dong ding.” (P3, unit 2)

“Then standing there I would turn like this.” (P8, unit 2)

“Nope, nope, nope. Wop, wop. He’ll be heeere. So, we have to go here now. Because he’s here.” (P13, unit 2)

“Ummm, hm hm hm hm hm hm.” (P16, unit 2)

“And there one, two, three, four, five, that’s right.” (P15, unit 2)

During the simulation, the children used various techniques: counting, command names (e.g. *step forward*), their interpretation of the command (e.g. *hop* instead of *jump over a field*), and non-verbal sounds (e.g. “hm”, “so”).

Thinking and imagining about the action the figurine would execute is also a form of *embodiment* [30]. At the same time, the children used the figure before executing the program without being instructed to do so, which also indicates *embodiment*. Both helps them to build their understanding of things through a substitute physical experience [5]. The playful programming environment may have provoked this behavior to explore and play with the possibilities, as noted earlier in section 5.1.4.

This behavior was exclusive to *unplugged* programming. During the programming of *ScratchJr*, the children did not use neither mental simulation nor *embodiment* as a means to help with the programming. However, in all groups it was observed that they were imitating the sounds of *ScratchJr* or the touch sound the tablet would make.

“She does- she does more like this. (pretending).” (P3, unit 3)

“Duhh.” (P4, unit 3)

“Dub, dub, duuh. (sounds)” (P1, unit 3)

“(does pop sounds)” (P5, unit 3)

“Dud. (Geräusch)” (P9, unit 3)

“Dududududu.” (P18, unit 3)

“Dududu.” (P14, unit 3)

“Ppp. (Geräusch)” (P17, unit 4)

“Dud.” (P12, unit 4)

5. Results and Discussion

A possible reason for the lack of this behavior in *ScratchJr* programming could be the lack of enough visual objects to transfer the thought to. Since these are not actions that children can do (i.e. actions that the characters do) [23], except for touching the tablet and then imitating the sound. So simple mouth-made sounds could be a sort of substitute for mental simulation.

In summary, children used mental simulation during *unplugged* programming to help them think about their programming actions. During *ScratchJr* programming, this behavior was not observed, but instead the children substituted sounds with those they made themselves.

5.2.6. Backseat-Programming

A behavior that was only observed during *ScratchJr* programming is *backseat-programming*. The name is directly derived from “backseat gaming,” which refers to “non-players interfering with the game by making unsolicited comments and opinions” [55]. In this context, it is the communication from the group members to the child who is programming with the tablet in an imperative mood.

“(whispers) You have to go on that.” (P2, unit 3)

“And now press the red button.” (P3, unit 3)

“Here you have the function list. You first have to press here now. And now you have the function list here.” (P3, unit 3)

“You have to do that there.” (P5, unit 3)

“You have to go on orange, on orange!” (P1, unit 3)

“Now you. And put five on there.” (P8, unit 3)

“No, you have to do tw-.” (P14, unit 3)

“There you have to go down 11, there you have to go down 11.” (P18, unit 3)

“Come on, do it!” (P4, unit 4)

“And now you still have to program these.” (P15, unit 4)

An explanation for this could be that only one child is in control of the tablet, as they were encouraged to do so and then take turns. The only thing the other group members could do during the programming was to verbally help or discuss something, unlike in *unplugged* programming where children could constantly help with something actively. As a result, when the children could not do something, they sometimes tried to control what the child was doing with the tablet.

Similar to this behavior is pair programming when leading or directing group members (cf. section 5.2.1). The difference, however, is that the child doing it was programming at the same time.

5.2.7. Distraction by the Tablet

ScratchJr has many features besides programming, such as designing and coloring own characters. As a result, children spent a lot of time doing that instead of

5. Results and Discussion

programming. Almost always, when children chose a character for their program, they changed the color or customized it in some other way.

“(records) Hello (inaudible). How are you? How’s it going?” (P3, unit 3)

“And here you can even paint, your own background.” (P3, unit 3)

“Hehehe. Can we finally color in the wizard now?” (P7, unit 3)

“And the belly I’ll do in red. The hat I’ll do in blue.” (P11, unit 3)

“But we can also change the cat!” (P9, unit 3)

“Shall we take underwater or jungle?” (P15, unit 3)

“We have to here! We can paint there.” (P3, unit 4)

“Wait, we have to color the dragon...” (P6, unit 4)

“We take the horse. Which we then change.” (P10, unit 4)

“Here we have to paint Emilia² again. Green.” (P9, unit 4)

“Yes, that’s nice... Yes, or this? And the dots?... What? Yes, like this.” (P13, unit 4)

As a consequence, programming became secondary and led to children not exploring the commands and functions of *ScratchJr*, resulting in some children missing something or getting confused, especially in *unit 4*.

“The ball moves too. The cat has to do it.” (P3, unit 3)

“The Eskimo- the Eskimo says hello to the penguin. Where do you use the hello button?” (P1, unit 4)

“We have to ask again.” (P18, unit 4)

“How can you change the backgrounds in this?” (P10, unit 4)

“Wait, I want to get the Tic² in here now. How do I do that?” (P17, unit 4)

“We have to delete that. How can you delete it?” (P13, unit 4)

In *unit 4* the groups had to program the written story of the other group. Out of the 17 minutes for the first group (because they needed more time for explanations) and 27 minutes for the other groups, the groups spent on average 61.19% of the time programming and 17.47% of the time designing the scenes and *characters*. The remaining time was spent discussing or doing something else. See Appendix D for the calculation.

In a short, programming in *ScratchJr* led to some distractions during programming, may it be because only one can actively program, the tools and functions in *ScratchJr* or both learning how to navigate *ScratchJr* and understanding this programming. All of this combined could have led to children not getting the understanding of programming that they got during *unplugged* programming, where there were fewer interruptions.

²This is a character in the story of the other group.

5.2.8. Other Interesting Things

There were other little things that were observed during the programming that did not fit into a category, but could be insightful.

Number of Commands One command that is used in almost every program is the *one forward* command. In *unplugged* and *ScratchJr* it works similarly, but in *ScratchJr* the perspective is different and one command block can count for multiple steps. Figuring out how many steps are needed was approached differently by the children depending on whether they were programming *unplugged* or in *ScratchJr*. During the *unplugged* programming, the children counted the fields the *robot* has to go on.

“One, two, three and!” (P3, unit 2)

“I just want to say five times to the right here. ... Want 1, 2, 3, 4, 5, then he would already be right here.” (P4, unit 2)

“(quietly) One, two, three.” (P9, unit 2)

“Five. One, two, three, four, five.” (P8, unit 2)

“One, one, two, three, four, five, six. (places with the figure) One, two, three, four, five, six. Okay. Perfect.” (P12, unit 2)

“One, two, three times you have to go forward.” (P15, unit 2)

“Yes, but it would be much easier if he (taps on the pitch) one, two... three, four, five... that would be easier.” (P14, unit 2)

“Baaash... One, two, three. So only three.” (P15, unit 2)

“Now he’s rotating here. And there one, two, three, four, five, that’s right. We can take that down too. And... five.” (P15, unit 2)

“Then ma then we better do the repeat command... One, two, three, four, five, six times...” (P15, unit 2)

However, while programming *ScratchJr*, the distance the character has to move was not counted, even after the supervisor showed the *grid function* for easier counting. For the most part it was guessed, and in some cases the number was greatly exaggerated.

“Turn 220 times to the right.” (P4, unit 3)

“Yes, we put that in a 55. Should that (inaudible) the thing should do 55 times.” (P7, unit 3)

“How many times do we want to do the loop? 18 times?” (P7, unit 3)

“150(?) times, nah 100 times up!” (P7, unit 3)

“99.. 10 times is then enough upwards. There 10 times is enough upwards.” (P7, unit 3)

“...we do until he’s out of the scene. That’s about... let’s just do... 11.” (P8, unit 3)

“We’ll put a hundred in there!” (P14, unit 3)

“Hun- write a hundred and eighty there and then it will work.” (P18, unit 3)

5. Results and Discussion

“Then we write 66 in there.” (P18, unit 3)

“No, then we write 90. That is (inaudible)” (P14, unit 3)

One explanation for this contrast between the two different approaches could be that the children did not perceive the perspective of the environment and the characters, so they chose a high number to be sure. The clues they had during the *unplugged* programming were probably more understanding for them to use counting intuitively, and so with the changed perspective it may not be as obvious or too abstract for children. Alternatively, they may have chosen a high number just to see what would happen and have some fun.

Short Communication Furthermore, during the programming of *ScratchJr* it was noticeable that the communication within the group sometimes became shorter and more reactionary. This refers to one-word sentences and having to say something about almost everything. In addition, the children were quieter at times, which may have been during times when they were paying attention to the tablet.

Changing and Making up Own Commands In some instances, the children used different names for some of the commands than they learned in *unit 1* or can read in *ScratchJr*. When programming and executing the program, they use their new names, such as *“So first the flag”* (P3, unit 2) for *start*, *“Do the baby.”* (P6, unit 2) for *crouch* command, *“[...] snail’s pace.”* (P3, unit 2) for *slow* command, *“So, here is the house.”* (P12, unit 2) for *goal*, *“And now close the bracket again.”* (P11, unit 2) and *“[...] multiple-thingy command execute or lie down.”* (P3, unit 2) for the *repeat* command.

It was also observed that children came up with their own commands during both *unplugged* and *ScratchJr* programming. Why they did this could be sometimes playful imagination, thinking about possibilities, making the program easier with their own command, or projecting their expectations onto it.

“No, the robot has transformed. Transformation card.” (P17, unit 2)

“There’s a TnT on it, if you do that, um, all the obstacles are gone for(?) the robot.” (P16, unit 2)

“Or there is also flying.” (P17, unit 2)

“[...] take a journey through time to here?” (P6, unit 2)

“Is there a do or opener here too?” (P6, unit 2)

“Is there any par- particular sign to go through the door?” (P4, unit 2)

During the programming in *ScratchJr* this imagination was less common, but in a similar but different way, kids sometimes asked for functionalities they knew from other tools.

“Where is the delete all button?” (P3, unit 3)

“Hahaha! We’d better take that one. Delete. Where is the recycle bin?” (P3, unit 3)

5. Results and Discussion

“Wait, there’s a delete button here somewhere, isn’t there? No, go back. Where is the delete button??” (P3, unit 3)

“Is there also an eraser??” (P16, unit 3)

“Do you also have thiiiiis button?” (P16, unit 4)

“Is there an eraser or something?” (P4, unit 4)

Making up and changing command names to their own creation could be due to the fact that the programming course was introduced as a game. So it was natural for children to use their imagination. It is notable that the other children in their group understood what they meant. Making up their own terms for the commands and objects may indicate that the children were comfortable with the concepts and understood them well enough.

5.2.9. Conclusion to How does programming and debugging compare in unplugged and block-based programming?

The programming is similar in some aspects and different in others between the *unplugged* and *ScratchJr* programming, see Table 5.2 for an overview. In summary, programming has changed or even become less effective with *ScratchJr*. This could be due to the change of perspective from top view to 2D side view, the change of tasks from precise tasks to more or less independent story programming, programming as a group to have only one person in control, and due to the previously explained things that led to less time programming. These changes may have influenced the active learning of programming and debugging. Furthermore, the presented and used *course method* follows the concept of presenting a problem and then solving it with the available programming commands. The change to another, more independent method (programming a story), which assumes that the children have learned the commands, recognize them and are then able to program multiple characters to form a story, could also be the main reason for the change and the impediment to learning to program.

ScratchJr is a nice tool for children to learn that their indirect action (building the program) can affect the characters, and then act as a stepping stone to learn the basics before moving on to *Scratch*. The playful setting with characters, backgrounds, and scenes makes programming more about exploring and building a story and less about solving a problem. This is perfectly fine, but since this programming course focused on programming as a means to solve problems and become more independent with it, this *course method* was not a good fit for learning to program with *ScratchJr*. Nevertheless, Bers [11, 12] and Strawhacker and Bers [72, 71] showed that *ScratchJr* can be a good tool for learning programming if used in an adequate way.

On the other hand, *unplugged* programming seems to have been effective, as it was also reported by Bati [9]. Children used intuitive problem-solving techniques, worked together in groups, and debugged. They did this independently, which is supported by research in early childhood education that children should be encouraged to try

5. Results and Discussion

Table 5.2.: Overview comparing programming and debugging between *unplugged* and *ScratchJr* programming

Behavior	<i>Unplugged</i> programming	<i>ScratchJr</i> programming
Pair programming	✓	✓(to a lesser extent)
Thinking aloud	✓	✓(to a lesser extent only with the active programming child)
Testing	Testing the program	See what happens
Debugging	Error detected and fixed	Error/behavior described and often left alone
Mental simulation and embodiment	Used for problem-solving	× (imitating tablet sounds)
Backseat-programming	×	✓
Tablet distraction	×	✓
Choosing command number	Count the fields	Guess, often an exaggerated high number
Short communication	×	✓
Changing/making up commands	✓	✓(to a lesser extent)

things on their own [77, 60]. It seems that the *course method* was suitable for this type of programming.

There are two ways to deal with the above complications without being limited to one programming method. One is to change the tasks presented while programming with *ScratchJr*. Instead of making them very independent and having loose structure, there could be predesigned *ScratchJr* programs for children to fill in, change or to complete with commands. Similar was done in the study of Chou [18]. Alternatively, to leave a little more freedom, the tasks could be for children to “reverse-engineer the *ScratchJr* program” by having them watch the story or animation and try to replicate it, as was done in the Strawhacker and Bers [72] study.

Another way could be to change the *block-based* programming part of *ScratchJr* to something else. This could be *robot programming* on a similar playing field or maze with *block-based* programming on a tablet or as an intermediate step *hybrid programming*. Using these programming methods, programming is much more identical to *unplugged* programming with the top view perspective and having tasks to

lead something from start to the goal.

Programming and debugging are much different with the presented *course method*.

The presented *course method* is appropriate for *unplugged programming*.

5.3. Answer to Is there a transfer of knowledge from unplugged to block-based programming?

In an attempt to answer this research question, the beginning of *unit 3* was devoted to an introduction in which children were shown printed *ScratchJr* programs and then asked what they would expect from the program without programming it first. For the first program they answered as the character would be the 3D *robot* accordingly, e.g. *jump over a field* instead of *ScratchJr jump up and down*.

“Yes, that’s easy. Jump one forward and then the goal.” (P3, unit 3)

“Um, he runs... he ju... [...] Or up? Or jumps. Up, I think up.” (P11, unit 3)

“He goes forward twice. [...] Up, then he turns 12 times, then he goes down twice, then he runs again...” (P12, unit 3)

“She runs two steps forward, then turns twelve times, to the right, goes back two steps, jogs, and jumps two times and then it’s over.” (P16, unit 3)

“Twice up...uuuh....rotate twelve times?! (confused) ...twice down.” (P4, unit 3)

“Goes forward. So six times forward, then it says hello and then...” (P13, unit 3)

“Hello. Then it sends a letter and then it goes.” (P8, unit 3)

“And shrink twice.” (P4, unit 3)

“Huh, we’ve already gone at a snail’s pace, [P3].” (P2, unit 3)

After programming it and seeing the different movement of the character compared to the *robot*, the children quickly recognized this and changed what to expect of the character’s movement for the next programs. By the time they got to tasks 3 and 4, which had two *characters* that were supposed to interact in *ScratchJr*, the children had trouble realizing that they belonged together.

Perhaps they expected the same in *ScratchJr* because they knew from the previous *units* that there was only one *robot* with only one program. Another reason could be, as mentioned before, that the children often did not read the task, so they may not have realized that the problem definitions are a separation between the tasks and the programs.

Another indication that the children were able to remember what they had learned is that on many occasions they identified the commands that they had previously learned. The same happened even with some command functions that were changed to fit real-world physics (e.g. *character* gets smaller to *crouch*). Children still recognized them before switching to the *ScratchJr* name after reading or seeing it.

5. Results and Discussion

“That’s the robot!” (P6, unit 3)

“Hold on, we have to give commands somehow... So look here.” (P15, unit 3)

“Six six forward. This way. No!” (P9, unit 3)

“Uh, goal, 5 forward, jump over twice and then on the goal.” (P11, unit 3)

“Well, I’ll start... first... Where’s the goal flag there?” (P9, unit 3)

“And 5. So now I’m going to jump over twice.” (P9, unit 3)

“First the starting flag. So...” (P16, unit 3)

“Getting faster. Where is getting faster?” (P6, unit 3)

“First, walk like Sonic.” (P4, unit 3)

This can be attributed to the design of the *unplugged* commands, even with some small changes, they were closely related to the *ScratchJr* models that children could easily understand the “new” *ScratchJr* commands only with their previous knowledge.

Furthermore, the children not only transferred the knowledge of the single components, but also the knowledge of how to program and the rules. When first programming with the tablet, the obstacle was not how to program, but how to get the commands to the programming environment. Though the supervisors mostly observed the children not recognizing how to do it and so the supervisors explained at the first sign of frustration.

“Oh, you have to drag.” (P6, unit 3)

“Where is this shown here?” (P3, unit 3)

“So like this?” (P9, unit 3)

“And where can you find the commands...” (P8, unit 3)

“You have to press there.” (P18, unit 3)

This may also be due to the fact that every child has handled a tablet before, so they might have some experience of what to expect.

In addition, the rules explained in *unit 1* are still followed regarding the program always starting with the *start program* command and ending with the *end program* command. Comparing programming to a game could have been beneficial in teaching programming rules. Just like game instructions or rules, some programming conventions must be followed, equivalent to simple programming syntax.

“But first you have to use start.” (P13, unit 3)

“You have to- you have to put the end there.” (P13, unit 3)

“Why is the start command missing?” (P18, unit 3)

“First start. First Start must go there.” (P18, unit 3)

“No, that, goal. The goal is missing. The end.” (P9, unit 3)

“You have to- you have to place the end there.” (P13, unit 3)

“There’s no end on there, that’s why it doesn’t work. Um...” (P18, unit 4)

“First start. (inaudible).” (P18, unit 4)

5. Results and Discussion

As a bonus, one group was enthusiastic about remembering the *roles* and chose and assigned their roles before the explanation began..

“I am the compiler.” (P6, unit 3)

“I am a programmer, so I am the tablet.” (P3, unit 3)

“I am the compiler, compiler.” (P6, unit 3)

“Because I am the robot.” (P2, unit 3)

5.3.1. Conclusion to Is there a transfer of knowledge from unplugged to block-based programming?

In summary, the children transferred what they learned from the previous *unplugged units* to *ScratchJr* in terms of commands, how programming works, and rules.

This means that the intention of designing the *unplugged* programming with the ulterior motive of switching the programming to *ScratchJr* worked as planned. Similarly, studies have used *unplugged* programming as an introduction to programming before *block-based* programming, see [4, 74, 85].

However, as discussed in section 5.2, children had their difficulties with programming and debugging during *ScratchJr* programming. They may have recognized the similarities, but this did not help them when it came to more abstract programming such as that done in *unit 4* story programming. The children had the basic knowledge, but to translate it from the top view and low level abstraction programming to the *ScratchJr* side view and higher level abstraction programming, it needed a little more introduction there.

Unplugged programming may prove to be a good entry point for young children to learn the basics of programming concepts, rules, and commands before moving on to other types of programming.

If other types of programming have similar features, basic programming knowledge can be transferred to them.

5.4. Answer to What debugging strategies do children use when none have been taught?

As discussed in section 5.2, the children were able to point out and debug errors in their programs, all during *unplugged* and some during *ScratchJr*. In the study by Klahr and Carver [44] they found that children did not have a problem with debugging the error, but finding it. Here it was observed that the children developed their own strategies for finding errors.

5.4.1. Unplugged Programming

Children may have performed better in *unplugged* programming and debugging, as discussed in section 5.2. As a result, three debugging strategies have been identified.

Strategy 1: Pair Programming: In the section 5.2.1, already explained as a programming behavior within the group, children also used it to their advantage as a debugging strategy. In the groups, one child took the lead and did the active programming by finding and placing the commands. The others helped by asking questions, finding commands, giving feedback, and observing the playing field and the program. Through this group effort, errors were quickly found and solved through discussion and suggestions.

Within this strategy, a different strategy emerged, which was reported by Misirli and Komis [50]: *verbal debugging*. The difference here is that the child who does this is one that does not actively program and is only pointing out the solution to the one who is programming. The pair programming strategy was only possible because the children worked collaboratively as a group.

Strategy 2: Simulation: As an other strategy children used mental simulation (see section 5.2.5) to visualize the *robot* path of their program and thereby encounter the error. In *unit 1* children executed the program in their own person (*full embodiment* [74]) before then switching to the *robot* (*low embodiment* [74]) which might have helped them mentally simulation the program and actions and thus finding the error.

This strategy is comparable to the *hand simulation* [38]. However, it is natural given, as the programming and execution is done by the children and not the *robot*. However, by doing so children did gain comprehension of their program and this may have helped them.

Strategy 3: Serial Search: This strategy is one that Klahr and Carver have reported [44] and one that they have taught to the children in their study. Here, the children naturally used it as they tested and ran their programs to find the error. By executing it step-by-step, the children found the error when they encountered a discrepancy between the path they expected and the path the *robot* took.

Bonus Strategy: Real-World Knowledge: This strategy is rather one that may have helped them to have a better understanding of the programming and therefore to have a better chance of finding a error. By designing the programming and materials to be age-appropriate, the children were enabled to recognize similarities to the real world and to project their previous knowledge and experience onto it. This kept programming concepts at a low level of abstraction that they could understand and combine with things they already understood.

5.4.2. ScratchJr Programming

During *ScratchJr*, children had their struggles with debugging, as they could see the errors but found it difficult to debug them. As a result, the debugging part was less

frequent than with *unplugged* programming, and sometimes the errors were even left alone.

Strategy 4: Symptom Debugging: This strategy is essentially the *simple mapping* strategy found by Katz and Anderson [38]. During testing, the children saw the behavior of the *ScratchJr* characters and found the error by comparing it to their expectation of what the characters should do instead. The children then proceed to debug the error, sometimes by *replacing* or *compensating* it. Both are strategies already reported by Ahn [4] and Nikolos et al. [56]. When they *compensated* the error, they tackled the symptom, but sometimes delayed the error, hence the strategy name.

Bonus Strategy: Code Review This strategy unfolded after the programming was finished and the groups presented their stories to everyone. During the presentation of the story, the other group pointed out the wrong behavior of the characters and errors, and sometimes suggested solutions when the presenting group had no approach how to solve it.

5.4.3. Conclusion to What debugging strategies do children use when none have been taught?

As they debugged in *unplugged* and *ScratchJr*, the children developed their own strategies for finding and solving the errors. During the *unplugged* programming, the children performed better in terms of debugging. This was also reported by Ahn [4]. In addition, the children did not limit themselves to one strategy, but developed and used several. Similar results were found in the studies of DeLiema et al. [21] and Sipitakiat and Nusen [68].

One explanation for the fewer strategies identified in the *ScratchJr* debugging is that the children did not spend as much time programming as in the *unplugged* units. Additionally, the amount of debugging in *ScratchJr* was lower, which could be due to children not understanding the programming as well as in *unplugged*.

To address this issue and possibly get more advanced strategies, even more programs with implemented errors could be integrated into *unplugged* units and also introduced into *ScratchJr* units. Nikolos et al [56] also suggested this by having a balance between programming and having finished programs with only debugging.

Without being taught, children can develop their own debugging strategies.

5.5. Answer to Are there differences in debugging between different age groups?

When designing the children's programming course, three groups were divided by grade level and one group was mixed. Since their ages are not that far apart,

5. Results and Discussion

the differences are not that obvious. However, during debugging discussions, some slight differences are noticeable. Younger children tend to be more vague about errors and solutions. They are more likely to use unspecific locative adverbs that are unclear without visual context. In contrast, the slightly older children use more often concrete language to describe the location of the error and in some cases immediately add a solution.

Young children:

“From then on. But then he hits the wall.” (P2, unit 2)

“Um left. Forward, forward. Huh, this way.” (P7, unit 2)

“We should do it like this five times.” (P2, unit 2)

“Wait t-, let’s just leave it. (error is just left like that)” (P3, unit 3)

“Why does the ball do that?” (P3, unit 3)

“Maybe wait a while. This will go away for now.” (P2, unit 3)

“(exhale) We did something wrong. Now we can’t get any further here.” (P5, unit 4)

Older children:

“Oh, we just forgot one forwards.” (P13, unit 2)

“Look, ...w(?) jump over then, one, two, then you turn to there?!” (P17, unit 2)

“Huh? It has to rotate twelve times.” (P8, unit 3)

“And then it says right after that, there he is, although the frog says something before that.” (P13, unit 3)

“That’s, that’s funny. He’s not using the others’. It starts with the... Wait, the lizard shrinks first.” (P13, unit 4)

One reason for this could very well be the result of cognitive development, which can change rapidly in young children [26, 27]. However, since the developmental stage is not cut by age or grade, these differences may not be clear.

Another difference noted, not really related to debugging, is the difference in planning how to program some actions. Younger children are more likely to look for the literal command action, where older children have a much better understanding of this abstract programming and look for commands to just make it look like it.

Young children:

“But how can you magically move the ball? (wonder if there is this command)” (P7, unit 3)

“Is there any magic here?” (P1, unit 3)

“Where is the kissing program?” (P3, unit 3)

“Um, how can you make the dragon breathe fire?” (P6, unit 4)

“On the corner. And we want to make it look like it’s evening? The cave has to be in the first scene. It won’t work. It won’t work.” (P9, unit 4)

Older children:

“Yes, we can use him as a lord.” (P14, unit 4)

5. Results and Discussion

“(inhale, excited) Yes, stop! We just have to draw a person... and then something on it, we don’t have to draw it ourselves! We just have to draw something on it like dummies. Oh man.” (P14, unit 4)

“(reads) ‘And waited for Putin to discover him. Putin did not discover him.’ Um and let’s just do it like this (inaudible)” (P18, unit 4)

Also, in *unit 4*, when writing the story for the other group, some older children thought about when the other group would have to program it and how they would do it, showing some planning skills and thinking ahead.

“So, he could be big, small, big, small, big, small. (disguised voice) Yesyesyesyes.” (P14, unit 4)

“Yes, they have to paint it really stupid.” (P14, unit 4)

“No, he’s not invisible, he’s hiding. But on the screen, it looks like...” (P16, unit 4)

“No, maybe there is no such thing! We know... under the sea.” (P15, unit 4)

“There must be one in there too. The...” (P18, unit 4)

“They don’t know who that is. We have to do without him now, come on. Either we would have put him in at the beginning or not now.” (P18, unit 4)

“They need lots of pictures, different backgrounds.” (P13, unit 4)

This was only noticed during *ScratchJr* programming. One explanation may be that during *unplugged* programming, commands were introduced step-by-step, so children knew every command they could and had to use. Another reason could be that *unplugged* programming was more tangible and easier for children to understand, as discussed earlier.

5.5.1. Conclusion to Are there differences in debugging between different age groups?

During debugging, there were some minor differences in the language used by younger and older children when addressing an error and its solution. Also during programming, older children tended to be better at abstract story programming.

Both cases suggest that *unplugged* programming is a good introduction to programming regardless of age, and that the introduction to *ScratchJr* should be handled differently based on the cognitive development of children.

Even when programming is age-appropriate, the type of programming can be critical to children’s understanding of programming.

Unplugged programming may be more accessible to younger children.

5.6. Further Exploration

Programming Roles During *unit 1* the three roles of *robot*, *compiler* and *programmer* were introduced as crucial for the “programming game”. For this *unit*,

5. Results and Discussion

the roles were decided with hidden drawing cards. After the first few tasks, the children understood and remembered what these roles meant and what tasks they were associated with. Then, during *unit 2*, the roles and associated tasks were recalled and distributed independently within the children's group.

"Here guys. I am the compiler." (P6, unit 2)

"I want to be the compiler again." (P1, unit 2)

"I announce." (P11, unit 2)

"Come on, so I can place it." (P6, unit 2)

"Yes, but I am the programmer." (P17, unit 2)

"Soo, I'm controlling the robot again.... And you're announcing again." (P15, unit 2)

"Hn actually you are only a compiler and builder." (P3, unit 2)

"That's why I have to draw everything. Yoo-hoo!" (P4, unit 2)

"Hey [P4], I am supposed to do this! (paint)" (P5, unit 2)

"As a robot, you have to do the same thing as [P5]." (P3, unit 2)

The roles were treated with importance and obeyed in their tasks. With them, children knew what they had to do during each part of the programming. Moreover, in *unplugged* programming it showed the importance of a correct and functioning program. Separating the one who reads the program from the one who plays the robot might have prevented cheating, i.e. just saying the right way instead of reading a correct program.

At the same time, the roles could have shown the children the "magic" behind a computer which simply reads the program as it is, and could have given them some insight into programming and computers, as it was introduced as "communicating in another language with the *robot*".

Audio Recording Device In each of the *units*, naturally, the children were clearly aware that the audio recording device was present, and some of them interacted with it during the tasks. Some just spoke into the device, sometimes personifying it or using it as a prop.

"Hello?" (P17, unit 2)

"(interrupts) Can you understand yourself?" (P15, unit 2)

"Nana. Nana." (P9, unit 2)

"Hello, stupid microphone. We can also do something funny again." (P3, unit 3)

"Can you hear me, you, you little thiiiiingy? Hello? Can you hear me? Hello. Boo." (P6, unit 3)

"Hello! Hello! Hello! Hello! Hello, Hello Hello, Hello." (P15, unit 3)

"No, really. Hello. Programming is a lot of fun. Andorn(?)." (P15, unit 3)

A few children asked what it was and why it was there during the *unit*. However, it did not seem that they acted differently or pretended to do anything for the recording, except when talking to it.

5. Results and Discussion

Competition At the beginning of each *unit* it was stated that the groups were not in competition with each other. Nevertheless, in every *unit*, except the first one, some kind of competition or comparison was noted. Those instances were sometimes comparing progress to their own group, just looking at what the other group is doing, having the goal of doing something better than the other group, or in *unit 4* making the story extra complicated for the other group to program (cf. section Debugging 5.2.4).

“They’re only just putting the crown on over there.” (P6, unit 2)

“The others are at the same.” (P4, unit 2)

“I’ll go and see what they’re doing.” (P17, unit 2)

“Ey, they have tunnels too.” (P6, unit 2)

“So done! We have won.” (P3, unit 2)

“We have already done this program.” (P15, unit 2)

“Come on, let’s demonstrate our strength.” (P1, unit 3)

“We are now making our strength, let’s demonstrate it to them.” (P1, unit 3)

“Our program is the best.” (P18, unit 3)

“Where are they? [...] Well, the others?” (P16, unit 3)

“Select reverse gear at the end. See if the other group is doing it right.” (P3, unit 4)

This competition was only noticed between groups, not within them, and may be just a playful and natural competition between friends or classmates. However, it may have motivated some to program better or faster.

Reading Along During the *ScratchJr* programming, when using the *Say* command and playing the animation, some children synchronized or read the said text out loud.

“Hello.” (P4, unit 3)

“Hello.” (P11, unit 3)

“Hello.” (P12, unit 3)

“’Neigh an apple.’ (P8, unit 3)

“So, ‘neigh an apple’ says the horse. Here’s an apple’ says the (inaudible). What does yours look like?” P8, unit 3)

“’Ouch.’ (P3, unit 4)

“’Abracadabra.’ (P1, unit 4)

“’Oh no!’ Hehe.”(P1, unit 4)

This was text that they wrote themselves or had to write based on the story they had to program. Compared to other things they had to read, such as the text of the tasks, these were rarely read at all. Therefore, if there was something that needed to be read by the children in order to understand a task, it could be done this way.

5. Results and Discussion

Feedback After the programming course in *unit 4* was completed, feedback sheets were distributed asking how each *unit* was enjoyed, what they liked most, what they liked least, and if they would participate again. A total of 82.35% said they would participate again. Of these, 17.65% said only programming on the tablet and 11.76% said on the tablet but something else. In the open answer of the most liked thing in the programming course most of the children (64.70%) answered that they liked the programming with the tablet the most which is reflected in the answers how each *unit* was enjoyed with the majority of positive answers for the *unit 4* followed by *unit 3*.

In addition, some of the children continue to stay after the end of the *units* to program some more. Sometimes the remaining children formed a new group with the other remaining children.

Based on both cases, it can be said that the children enjoyed the course and the programming.

5.7. Limitations

There are limitations to this study. One is how the programming course was implemented. Originally it was planned to have five *units* to have enough time for introductions and learning before moving on to applying the knowledge. However, due to time constraints, the course had to be held in only four *units*. Therefore, after each *unit*, the next *unit* was slightly modified to fit the important lessons and study objectives. Additionally, due to the short time, potential exploration time at the end of each *unit* had to be skipped in most cases because the introduction of the commands was more crucial.

As a result and as another limitation, the programming course took place only once a week and each type of programming and tasks i.e. *unplugged* programming on the big playing field and table and *ScratchJr* programming were present only once in each *unit*. These long intervals and changes could have affected the learning of programming. This was tried to be limited with a revision of the last *unit* and a repetition of a task at the beginning.

Another limiting factor is that the children volunteered for the programming course and were free to skip *units*. So it often happened that someone was missing for the *unit* or arrived later. This may have affected the acquisition and transfer of programming knowledge.

A possible other limitation is that most of the course planning, one of the supervisors of the course, and most of the transcribing was done by me, which may have affected the analysis.

5.8. Lessons learned

Through observations, in hindsight and the analysis some things that can be improved were revealed.

Units in General One of the modifications that should be made to the *units* is the length of a *unit* which was identified through observation and feedback done with the children at the end of the course. Both children and supervisors perceived that 45 minutes was not enough time. Increasing the length of a *unit* to 60 minutes should be within reason and allows enough time to plan exploitative programming after the tasks are completed at the end.

In addition, as originally planned, at least five *units* should be conducted, two of them dedicated to learning the programming, rules and for the step-by-step introduction of the commands. This leaves the remaining *units* for the presentation of the other types of programming, i.e. here *unplugged* and *ScratchJr*, with the basic knowledge of the commands.

Unplugged Units In the *unplugged unit*, it was noticed that the tasks were partially too easy, especially after the children were done with the step-by-step introduction of the commands. The tasks could be designed to become progressively more difficult. This may require children to use more problem-solving and come up with more or different strategies.

ScratchJr Units As indicated in the discussion, the *ScratchJr units* were designed with too much independence too early, making the switch to *ScratchJr* too abrupt. Even if children remember the commands and functions, an introduction would be beneficial. This could be done in the first *unit* with *ScratchJr* through prepared *ScratchJr* programs where the goal for children is to recognize the commands in the program and to fill in and complete programs. Then the tasks can become increasingly independent and difficult. This ensures that children have a similar step-by-step introduction to *ScratchJr*, but still need to recall previous programming knowledge.

Also, the *unit* objective of programming a story is a lot to do in a short amount of time. Instead, this could be split into two *units*, with one dedicated to planning the story and exploring possibilities, and the other to programming and debugging.

Material The children had no difficulty with the commands, not even the *repeat* command, and came up with their own. Based on this, new commands could be implemented that would make *unplugged* programming stand out from the *ScratchJr* design model. This could be the adoption of other programming concepts such as *if else* commands or the ability to create a custom with a blank command.

In addition, the way tasks are presented could be changed to digital as tablets are already available. This allows tasks to be easily changed, replaced, or added.

5. Results and Discussion

Programming and Debugging Assessment As stated earlier in section 5.4, more prepared programs with implemented errors could lead to more debugging and strategies. More of them should be interspersed between other tasks. To increase the effectiveness of the programs with the implemented errors, they could be completely prepared, e.g. digitally or on cards, instead of being laid out by the children first. This would allow them to concentrate on the debugging itself.

Prepared programs should also be introduced for *ScratchJr*. This could be done by describing the desired animation that the *characters* should do, or by showing a short video of the correct animation and letting the children debug the prepared faulty programs. This will make the debugging more controlled and have a set goal to achieve.

Screen Recordings on Tablets In light of the problems with screen recording encountered in the study, it would be advisable to limit the recording time to individual tasks or to keep it as short as possible. Another approach would be to film the screens with a video camera set up to film the screen from top down.

6. Conclusion and Future Work

This thesis and research focused on children's programming and debugging when they were not taught strategies but had to develop their own. For this purpose, a programming course using *unplugged* and *ScratchJr* programming was designed and implemented with volunteer elementary school children over a period of four weeks.

By analyzing the transcripts of the audio recordings, it was found that the children's reactions to implemented errors did not differ from their own errors, and that children came up with four different debugging strategies for *unplugged* and *ScratchJr* programming. In addition, the children had no problem recognizing the similarities between the two types of programming and were able to transfer their knowledge of programming and commands. However, the method presented here to switch to and teach *ScratchJr* was not suitable, so while the programming was not hard for the children to recognize, the debugging was more difficult for the children and it seems that they could not transfer their previous knowledge to the *ScratchJr* debugging. Additionally, during debugging, some slight differences were found between the age ranges in terms of the language they use when encountering and solving an error. Overall, it was found that the *unplugged* programming method used here is appropriate and easily understood by children of this age group, so that they can program and explore independently.

These results suggest that it is possible to encourage children to freely explore programming concepts and even debugging if the programming and materials are age appropriate. A beneficial way to learn programming is *unplugged* programming, as it is less abstract and tangible, and offers opportunities for children to explore at their own pace and apply their previous knowledge to the real world. Same is reported through other studies [13, 9, 14]. From there, children can take the basic programming skills they have learned and apply them to other types of programming, as long as they are similar.

Future work could test the hypotheses presented in this thesis. Further, testing whether the roles have an impact on children's perceptions of how computer programming or robot programming works could bring new investigations. Moreover, it may be interesting to introduce new programming related roles for children to take and act on. In addition, research could be done to see if children can learn and then recognize types of errors faster and if this would help them with unknown errors.

7. Acknowledgments

I would like to thank professor Janet Siegmund and my supervisor Elisa Hartmann for the opportunity and help to create and conduct the programming course for children and for their insights and comments on the research and on this thesis. I would also like to thank my husband Vincent Kühn for his encouragement and support and my cats Nova and Juno for emotional comfort.

Bibliography

- [1] Edith Ackermann. 2001. Piaget’s Constructivism, Papert’s Constructionism: What’s the Difference. *Future of Learning Group Publication* 5, 3 (2001), 438.
- [2] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An Analysis of Patterns of Debugging Among Novice Computer Science Students. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. 84–88.
- [3] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2007. The Impact of Improving Debugging Skill on Programming Ability. *Innovation in Teaching and Learning in Information and Computer Sciences* 6, 4 (2007), 72–87.
- [4] Junghyun Ahn. 2020. *Computational Thinking in Children: The Impact of Emulation on Debugging Practices in Programming*. dissertation. Columbia University. <https://academiccommons.columbia.edu/doi/10.7916/d8-k5ee-q864>
- [5] Alissa N Antle. 2009. Embodied Child Computer Interaction: Why Embodiment Matters. *Lifelong Interactions* 16, 2 (2009), 27–30.
- [6] Keijiro Araki, Zengo Furukawa, and Jingde Cheng. 1991. A General Framework for Debugging. *IEEE software* 8, 3 (1991), 14–20.
- [7] Barbara Arfé, Tullio Vardanega, and Lucia Ronconi. 2020. The Effects of Coding on Children’s Planning and Inhibition Skills. *Computers & Education* 148 (2020), 103807.
- [8] Michal Armoni, Orni Meerbaum-Salant, and Mordechai Ben-Ari. 2015. From Scratch to “Real” Programming. *ACM Transactions on Computing Education (TOCE)* 14, 4 (2015), 1–15.
- [9] Kaan Bati. 2022. A Systematic Literature Review Regarding Computational Thinking and Programming in Early Childhood Education. *Education and Information Technologies* 27, 2 (2022), 2059–2082.
- [10] Tim Bell, Jason Alexander, Isaac Freeman, and Mick Grimley. 2009. Computer Science Unplugged: School Students Doing Real Computing Without Computers. *The New Zealand Journal of Applied Computing and Information Technology* 13, 1 (2009), 20–29.

BIBLIOGRAPHY

- [11] Marina Umaschi Bers. 2018. Coding and Computational Thinking in Early Childhood: The Impact of Scratchjr in Europe. *European Journal of STEM Education* 3, 3 (2018), 8.
- [12] Marina Umaschi Bers. 2019. Coding as Another Language: A Pedagogical Approach for Teaching Computer Science in Early Childhood. *Journal of Computers in Education* 6, 4 (2019), 499–528.
- [13] Marina Umaschi Bers, Jessica Blake-West, Madhu Govind Kapoor, Tess Levinson, Emily Relkin, Apittha Unahalekhaka, and Zhanxia Yang. 2023. Coding as Another Language: Research-Based Curriculum for Early Childhood Computer Science. *Early Childhood Research Quarterly* 64 (2023), 394–404.
- [14] Marina Umaschi Bers, Louise Flannery, Elizabeth R Kazakoff, and Amanda Sullivan. 2014. Computational Thinking and Tinkering: Exploration of an Early Childhood Robotics Curriculum. *Computers & Education* 72 (2014), 145–157.
- [15] Imed Bouchrika. 2023. *What Age Should a Child Get a Smartphone: Pros and Cons of Early Phone Use.* <https://research.com/education/what-age-should-a-child-get-a-smartphone>
- [16] John Byrne and Paula Prendeville. 2020. Does a Child’s Mathematical Language Improve When They Engage in Cooperative Group Work in Mathematics? *Education 3-13* 48, 6 (2020), 627–641.
- [17] Giuseppe Chiazzese, Marco Arrigo, Antonella Chifari, Violetta Lonati, and Crispino Tosto. 2018. Exploring the Effect of a Robotics Laboratory on Computational Thinking Skills in Primary School Children Using the Bebras Tasks. In *Proceedings of the Sixth International Conference on Technological Ecosystems for Enhancing Multiculturality*. 25–30.
- [18] Pao-Nan Chou. 2020. Using Scratchjr to Foster Young Children’s Computational Thinking Competence: A Case Study in a Third-Grade Computer Class. *Journal of Educational Computing Research* 58, 3 (2020), 570–595.
- [19] Andy Clark. 2008. *Supersizing the Mind: Embodiment, Action, and Cognitive Extension*. OUP USA.
- [20] Nancy DeJarnette. 2012. America’s Children: Providing Early Exposure to Stem (Science, Technology, Engineering and Math) Initiatives. *Education* 133, 1 (2012), 77–84.
- [21] David DeLiema, Maggie Dahn, Virginia J Flood, Ana Asuncion, Dor Abrahamson, Noel Enyedy, and Francis Steen. 2019. Debugging as a Context for Fostering Reflection on Critical Thinking and Emotion. *Deeper Learning, Dialogic Learning, and Critical Thinking: Research-based Strategies for the Classroom*. Hrsg. von Emmanuel Manalo. New York: Routledge (2019), 209–228.

BIBLIOGRAPHY

- [22] DevTech Research Group, Lifelong Kindergarten Group, Playful Invention Company. 2014. *ScratchJr*. <https://www.scratchjr.org>
- [23] Ryan Elder and Krishna Aradhna. 2014. Grasping the Grounded Nature of Mental Simulation. *The Inquisitive Mind* 20, 4 (2014).
- [24] K Anders Ericsson and Herbert A Simon. 1998. How to Study Thinking in Everyday Life: Contrasting Think-Aloud Protocols With Descriptions and Explanations of Thinking. *Mind, Culture, and Activity* 5, 3 (1998), 178–186.
- [25] Georgios Fessakis, Evangelia Gouli, and Elisavet Mavroudi. 2013. Problem Solving by 5–6 Years Old Kindergarten Children in a Computer Programming Environment: A Case Study. *Computers & Education* 63 (2013), 87–97.
- [26] Kurt W Fischer and Louise Silvern. 1985. Stages and Individual Differences in Cognitive Development. *Annual Review of Psychology* 36, 1 (1985), 613–648.
- [27] Louise P Flannery, Brian Silverman, Elizabeth R Kazakoff, Marina Umaschi Bers, Paula Bontá, and Mitchel Resnick. 2013. Designing Scratchjr: Support for Early Childhood Learning Through Computer Programming. In *Proceedings of the 12th International Conference on Interaction Design and Children*. 1–10.
- [28] MuckRock Foundation. 2018. *oTranscribe*. <https://otranscribe.com>
- [29] Neil Fraser, Quynh Neutron, Ellen Spertus, and Mark Friedman. 2012. Blockly. <https://developers.google.com/blockly>. Accessed: 2023-08-20.
- [30] Carl Gabbard. 2013. The Role of Mental Simulation in Embodied Cognition. *Early Child Development and Care* 183, 5 (2013), 643–650.
- [31] Tancicleide Carina Simões Gomes, Taciana Pontual Falcão, and Patrícia Cabral de Azevedo Restelli Tedesco. 2018. Exploring an Approach Based on Digital Games for Teaching Programming Concepts to Young Children. *International Journal of Child-Computer Interaction* 16 (2018), 77–84.
- [32] Leo Gugerty and Gary Olson. 1986. Debugging by Skilled and Novice Programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 171–174.
- [33] Jo E Hannay, Tore Dybå, Erik Arisholm, and Dag IK Sjøberg. 2009. The Effectiveness of Pair Programming: A Meta-Analysis. *Information and Software Technology* 51, 7 (2009), 1110–1122.
- [34] Mia Heikkilä and Linda Mannila. 2018. Debugging in Programming as a Multimodal Practice in Early Childhood Education Settings. *Multimodal Technologies and Interaction* 2, 3 (2018), 42.

BIBLIOGRAPHY

- [35] Thomas Hirsch and Birgit Hofer. 2021. What We Can Learn From How Programmers Debug Their Code. In *2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*. IEEE, 37–40.
- [36] Michael S Horn, R Jordan Crouser, and Marina U Bers. 2012. Tangible Interaction and Learning: The Case for a Hybrid Approach. *Personal and Ubiquitous Computing* 16, 4 (2012), 379–389.
- [37] Michael S Horn and Robert JK Jacob. 2007. Designing Tangible Programming Languages for Classroom Use. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction*. 159–162.
- [38] Irvin R Katz and John R Anderson. 1987. Debugging: An Analysis of Bug-Location Strategies. *Human-Computer Interaction* 3, 4 (1987), 351–399.
- [39] Elizabeth R Kazakoff and Marina Umaschi Bers. 2014. Put Your Robot In, Put Your Robot Out: Sequencing Through Programming Robots in Early Childhood. *Journal of Educational Computing Research* 50, 4 (2014), 553–573.
- [40] Elizabeth R Kazakoff, Amanda Sullivan, and Marina U Bers. 2013. The Effect of a Classroom-Based Intensive Robotics and Programming Workshop on Sequencing Ability in Early Childhood. *Early Childhood Education Journal* 41 (2013), 245–255.
- [41] ChanMin Kim, Jiangmei Yuan, Lucas Vasconcelos, Minyoung Shin, and Roger B Hill. 2018. Debugging During Block-Based Programming. *Instructional Science* 46 (2018), 767–787.
- [42] KinderLab Robotics. 2014. *KIBO Learning Robots in Action*. <https://kinderlabrobotics.com/kibo/in-action/>
- [43] Julian Kiverstein. 2012. The Meaning of Embodiment. *Topics in Cognitive Science* 4, 4 (2012), 740–758.
- [44] David Klahr and Sharon McCoy Carver. 1988. Cognitive Objectives in a Logo Debugging Curriculum: Instruction, Learning, and Transfer. *Cognitive psychology* 20, 3 (1988), 362–404.
- [45] Nina Körber, Lisa Bailey, Luisa Greifenstein, Gordon Fraser, Barbara Sabitzer, and Marina Rottenhofer. 2020. An Experience of Introducing Primary School Children to Programming Using Ozobots (Practical Report). In *The 16th Workshop in Primary and Secondary Computing Education*. 1–6.
- [46] John Kounios and Mark Beeman. 2009. The Aha! Moment: The Cognitive Neuroscience of Insight. *Current Directions in Psychological Science* 18, 4 (2009), 210–216.

BIBLIOGRAPHY

- [47] Chen Li, Emily Chan, Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2019. Towards a Framework for Teaching Debugging. In *Proceedings of the Twenty-First Australasian Computing Education Conference*. 79–86.
- [48] Cecilia Martinez, Marcos J Gomez, and Luciana Benotti. 2015. A Comparison of Preschool and Elementary School Children Learning Computer Science Concepts Through a Multilanguage Robot Programming Platform. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*. 159–164.
- [49] Sabine Martschinke, Susanne Palmer Parreira, and Ralf Romeike. 2021. Informatische (Grund-) Bildung Schon in Der Primarstufe? Erste Ergebnisse Aus Einer Evaluationsstudie. *Technische Bildung im Sachunterricht der Grundschule* (2021), 133.
- [50] Anastasia Misirli and Vassilis Komis. 2023. Computational Thinking in Early Childhood Education: The Impact of Programming a Tangible Robot on Developing Debugging Knowledge. *Early Childhood Research Quarterly* 65 (2023), 139–158.
- [51] MIT Media Lab. 2003. *About Scratch*. <https://scratch.mit.edu/about>
- [52] Jaime Montemayor, Allison Druin, Allison Farber, Sante Simms, Wayne Churaman, and Allison D’Amour. 2002. Physical Programming: Designing Tools for Children to Create Physical Interactive Environments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 299–306.
- [53] Luke Moors, Andrew Luxton-Reilly, and Paul Denny. 2018. Transitioning From Block-Based to Text-Based Programming Languages. In *2018 International Conference on Learning and Teaching in Computing and Engineering (LaT-ICE)*. IEEE, 57–64.
- [54] Nancy M Morris and William B Rouse. 1985. Review and Evaluation of Empirical Research in Troubleshooting. *Human factors* 27, 5 (1985), 503–530.
- [55] Katharina Müller. 2021. *Was Bedeutet "Backseat-Gaming"? Bedeutung und Verwendung*. <https://www.netzwelt.de/abkuerzung/186244-bedeutet-backseat-gaming-bedeutung-verwendung.html>
- [56] Dimitrios Nikolos, Anastasia Misirli, and Vassilis Komis. 2021. Children’s Debugging Processes and Strategies With a Simulated Robot: A Case Study. In *Educational Robotics International Conference*. Springer, 64–74.
- [57] Ozobot’s Team. 2012. *Was ist Ozobot?* <https://ozobot-deutschland.de/ozobot/>
- [58] Seymour Papert. 1980. Mindstorms” Children. *Computers and powerful ideas* (1980).

BIBLIOGRAPHY

- [59] Seymour Papert. 1987. Information Technology and Education: Computer Criticism vs. Technocentric Thinking. *Educational researcher* 16, 1 (1987), 22–30.
- [60] Priyanka Parekh and Elisabeth R Gee. 2019. Tinkering Alone and Together: Tracking the Emergence of Children’s Projects in a Library Workshop. *Learning, Culture and Social Interaction* 22 (2019), 100313.
- [61] Radia Perlman. 1976. Using Computer Technology to Provide a Creative Learning Environment for Preschool Children. (1976).
- [62] Jean Piaget and Baerbel Inhelder. 1967. *The Child’s Conception of Space*. W. W. NortonCo. 375–418 pages.
- [63] Primo Toys. 2013. *Meet Cubetto*. <https://www.primotoys.com>
- [64] JOCHEN Prümper, Dieter Zapf, Felix C Brodbeck, and Michael Frese. 1992. Some Surprising Differences Between Novice and Expert Errors in Computerized Office Work. *Behaviour & Information Technology* 11, 6 (1992), 319–328.
- [65] Peter Rich and Brian Jones. [n. d.]. Connected Code: Why Children Need to Learn Programming. *Teachers College Record* ([n. d.]).
- [66] Andreas Schwill. 2001. Ab Wann Kann Man Mit Kindern Informatik Machen. *INFOS2001-9. GI-Fachtagung Informatik und SchuleGI-Edition* (2001), 13–30.
- [67] Deborah Silvis, Jody Clarke-Midura, Jessica Shumway, and Victor R Lee. 2021. Objects to Debug With: How Young Children Resolve Errors With Tangible Coding Toys. In *Proceedings of the International Society of the Learning Sciences (ISLS) annual meeting*.
- [68] Arnan Sipitakiat and Nusarin Nusen. 2012. Robo-Blocks: Designing Debugging Abilities in a Tangible Programming System for Early Primary School Children. In *Proceedings of the 11th International Conference on Interaction Design and Children*. 98–105.
- [69] Catherine E Snow and Marian Hoefnagel-Höhle. 1978. The Critical Period for Language Acquisition: Evidence From Second Language Learning. *Child Development* (1978), 1114–1128.
- [70] Cynthia Solomon, Brian Harvey, Ken Kahn, Henry Lieberman, Mark L Miller, Margaret Minsky, Artemis Papert, and Brian Silverman. 2020. History of Logo. *Proceedings of the ACM on Programming Languages* 4, HOPL (2020), 1–66.
- [71] Amanda Strawhacker and Marina U Bers. 2015. “I Want My Robot to Look For Food”: Comparing Kindergartner’s Programming Comprehension Using Tangible, Graphic, and Hybrid User Interfaces. *International Journal of Technology and Design Education* 25 (2015), 293–319.

BIBLIOGRAPHY

- [72] Amanda Strawhacker and Marina Umaschi Bers. 2019. What They Learn When They Learn Coding: Investigating Cognitive Domains and Computer Programming Knowledge in Young Children. *Educational Technology research and Development* 67 (2019), 541–575.
- [73] Felix Suessenbach, Eike Schröder, and Mathias Winde. 2023. Informatikunterricht: Deutschland Abgehängt in Europa. <https://doi.org/10.5281/zenodo.7515985>
- [74] Woonhee Sung, Junghyun Ahn, and John B Black. 2017. Introducing Computational Thinking to Young Learners: Practicing Computational Perspectives Through Embodiment in Mathematics Education. *Technology, Knowledge and Learning* 22 (2017), 443–463.
- [75] Terrapin. 2017. *Bee-Bot*. <https://www.betzold.de/prod/83721/>
- [76] Yune Tran. 2019. Computational Thinking Equity in Elementary Classrooms: What Third-Grade Students Know and Can Do. *Journal of Educational Computing Research* 57, 1 (2019), 3–31.
- [77] Shirin Vossoughi and Bronwyn Bevan. 2014. Making and Tinkering: A Review of the Literature. *National Research Council Committee on Out of School Time STEM* 67 (2014), 1–55.
- [78] David Weintrop. 2019. Block-Based Programming in Computer Science Education. *Commun. ACM* 62, 8 (2019), 22–25.
- [79] Mark Weiser. 1982. Programmers Use Slices When Debugging. *Commun. ACM* 25, 7 (1982), 446–452.
- [80] Laurie Williams and Richard L Upchurch. 2001. In Support of Student Pair-Programming. *ACM Sigcse Bulletin* 33, 1 (2001), 327–331.
- [81] Jeannette M Wing. 2006. Computational Thinking. *Commun. ACM* 49, 3 (2006), 33–35.
- [82] Adam Winsler and Jack Naglieri. 2003. Overt and Covert Verbal Problem-Solving Strategies: Developmental Trends in Use, Awareness, and Relations With Task Performance in Children Aged 5 to 17. *Child development* 74, 3 (2003), 659–678.
- [83] Gary KW Wong and Shan Jiang. 2018. Computational Thinking Education for Children: Algorithmic Thinking and Debugging. In *2018 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*. IEEE, 328–334.

BIBLIOGRAPHY

- [84] Sarita Yardi and Amy Bruckman. 2007. What Is Computing? Bridging the Gap Between Teenagers' Perceptions and Graduate Students' Experiences. In *Proceedings of the Third International Workshop on Computing Education Research*. 39–50.
- [85] Goran Zaharija, Saša Mladenović, and Ivica Boljat. 2013. Introducing Basic Programming Concepts to Elementary School Children. *Procedia-Social and Behavioral Sciences* 106 (2013), 1576–1584.

A. File Share Code

A QR code to the folder containing each file. Alternatively see https://drive.google.com/drive/folders/1z0DEMKZ504A5U0sTe9AF2yIYrYRrIpiL?usp=drive_link



B. ScratchJr Programmed Stories

Example of a prepared story and the matching template for the own stories.

Geschichtenblatt

Schreibe eine kurze Geschichte auf!

Es war einmal...

ein blauer Zauberer, der lebte am Strand. Eines Tages
besuchte ihn eine Fee. Die war sehr frech und warf einen
Apfel auf den Zauberer. Daraufhin verwandelte der
Zauberer die Fee in einen Hasen.

Welche Figuren spielen mit in der Geschichte? (mindestens 3 Figuren!)

Zauberer (☆), Fee (△), Hase (♡)

Wo stehen die Figuren am Anfang?

☆ steht links
△ steht rechts
♡ steht rechts, versteckt

Was machen die Figuren miteinander?

△ geht zu ☆
△ wirft Apfel auf ☆ → Apfel bewegt sich von △ Hand zu ☆ Kopf
☆ verzaubert △ → ☆ sagt Zauberspruch
△ wird zu ♡ → △ versteckt sich und ♡ zeigt sich

Geschichtenblatt

Schreibe eine kurze Geschichte auf!

Es war einmal...

Welche Figuren spielen mit in der Geschichte? (mindestens 3 Figuren!)

Wo stehen die Figuren am Anfang?




Was machen die Figuren miteinander?

C. Feedback Sheet




Feedback sheet given after the programming course.

Wie hat dir die Programmier-AG gefallen?




Wie hat dir die AG die 1. Woche gefallen? (draußen auf großem Spielfeld)




Wie hat dir die AG die 2. Woche gefallen? (am Tisch in Gruppen)

Wie hat dir die AG die 3. Woche gefallen? (mit Tablets)?

Wie hat die heutige AG gefallen?

Die Programmier-AG war zu...

einfach genau richtig schwer

Die einzelnen Units waren zu...

kurz genau richtig lang

Was hat dir besonders gefallen?

Was hat dir am wenigsten gefallen?

Würdest du noch einmal eine Programmier-AG mitmachen?

Gibt es etwas was du noch zur Programmier-AG schreiben möchtest?

D. Percentage Calculation

The calculation of the percentages of programming and designing something in *unit* 4.

Transkript	Gruppe1_H1n	Gruppe1_H2n	Gruppe2_H1n	Gruppe3_H2n	Gruppe2_H2n	Gruppe3_H1n			
Beginn	25:45:00	25:38:00	12:03	17:00	14:42	16:47			
Ende	35:59:00	43:03:00	39:08:00	41:59:00	42:17:00	41:54:00			
Prog Start	26:26:00	27:51:00	12:30	17:00	14:42	17:00			
Prog Ende	27:38:00	31:25:00	13:43	18:57	16:56	26:08:00			
Design Start			13:48	19:00	16:58	26:38:00			
Design Ende			13:50	21:49	17:20	29:04:00			
Prog Start	28:14:00	34:44:00	13:52	21:56	17:20	29:04:00			
Prog Ende	29:43:00	35:50:00	14:07	25:05:00	19:21	35:08:00			
Design Start	29:51:00	37:22:00	14:13		19:26	35:11:00			
Design Ende	30:29:00	38:16:00	14:48		20:09	35:20:00			
Prog Start	30:33:00	38:19:00	14:52	27:32:00	20:12	35:56:00			
Prog Ende	30:48:00	39:39:00	15:49	26:53:00	21:55	39:34:00			
Design Start		41:36:00	15:49	27:05:00	22:03	39:39:00			
Design Ende		43:13:00	18:12	27:15:00	23:55	40:09:00			
Prog Start	31:13:00		18:19	27:26:00	24:05:00				
Prog Ende	34:38:00		19:38	28:10:00	27:56:00				
Design Start	34:49:00		19:46		29:08:00				
Design Ende	35:05:00		23:37		30:43:00				
Prog Start	35:16:00		23:41	29:00:00	32:35:00				
Prog Ende	35:19:00		38:48:00	38:18:00	37:39:00				
Design Start	35:21:00			38:31:00	37:52:00				
Design Ende	35:39:00			38:48:00	38:05:00				
Prog Start					38:10:00				
Prog Ende					41:56:00				
Design Start				39:17:00					
Design Ende				41:59:00					
Prog Start									
Prog Ende									
Design Start									
Design Ende									
Gesamtzeit	10:14:00	17:25:00	27:05:00	24:59:00	27:35:00	25:07:00	13:49:30	26:11:30	
Programmierzzeit	06:24:00	06:00:00	18:51	14:29	18:39	18:50:00	06:12:00	17:42	
Ablenkung	01:12:00	02:31:00	06:51	05:58	04:45	03:05:00	01:51:30	05:09	
% Prog	62,54071661	34,44976077	69,6	57,97198132	67,61329305	74,98341075			Durchschnitt: 61,19319375
% Ablenkung	11,72638436	14,44976077	25,29230769	23,88258839	17,22054381	12,27604512			Durchschnitt: 17,47460502

E. Original Quotes

From How do children react to programming errors, and are there differences between their own and implemented errors?

“Fehler, Fehler, Fehler, Fehler, Fehler!” (P4)

”(hörbar einatmen) Wir haben ein Fehler gemacht! Wir haben ein Fehler gemacht! Das darf nicht hier sein! Sondern hier muss das sein! (P4, unit 2)

”Gleich ein Fehler entdeckt.” (P4, unit 2)

”Ich glaub ich seh da schon ein Fehler drin.” (P6, unit 2)

”Die Krabbe ist am Platz geblieben HaHa.” (P12, unit 3)

Aha! Moment

“Stimmt! Wir brauchen noch eins nach oben hier.” (P1, unit 2)

“Ach, da fehlt eins.” (P8, unit 2)

“Oh, wir haben nur eins nach vorne vergessen.” (P13, unit 2)

“Oh meine... zweimal nur(?) groß machen. Hn.” (P2, unit 3)

“Da? Oh. ... Zielfahne.” (P11, unit 3)

“Hää? Der springt in die Luft und kommt immer... ah, der soll vom Boden starten und springt dann aufs Bett.” (P18, unit 3)

“Nee, das ist sagen. Ah doch, doch (unhörbar).” (P11, unit 3)

“Doch. Ne warte, das stimmt, du hast recht. Wir brauchen doch sieben.” (P16, unit 3)

“Oorh wie, ah stimmt!” (P4, unit 4)

“Warum macht das? Oh das Baby machts.” (P4, unit 4)

“Warte nee, das hier können wir löschen, warte, ich muss gut ansetzen mhm. Mhm. Mhmmm. Mhmmm. Oh. Warte.” (P16, unit 4)

“Ah da geht's ja gar nicht mehr. Danach...” (P13, unit 4)

“Ah, stimmt die verschwinden eigentlich gar nicht.” (P16, unit 4)

Describe Behaviour

“Hier, laufen, dreimal geradeaus, Ball wird mitgenommen irgendwie. Und das ist irgendwie so komisch. (P3, unit 3)

“Ey der ist ja noch nicht mal gesprungen!” (P1, unit 3)

“Hu? Der kommt aber gerade nicht aus der Hand geschossen.” (P7, unit 3)

“Die Krabbe ist am Platz geblieben HaHa.” (P12, unit 3)

“Der wird doch gar nicht kleiner?” (P8, unit 3)

“Und wo ist der Ball? Der wird doch gar nicht weggezaubert.” (P3, unit 4)

“He? Und hier ist noch die Fee da.” (P1, unit 4)

“Ja, aber es is halt blöd, weil der Zauberer, der also... Jetzt is das weg... Also, es geht irgendwie nicht.” (P13, unit 4)

“Die Eidechse macht ja auch was, aber die kommt hier irgendwie nicht vor.” (P13, unit 4)

“Aber da bewegt sich ja die Krone von selbst.” (P12, unit 4)

“Aber... ja, aber die sollen doch nicht kommen. Die sollen doch weg sein.” (P16, unit 4)

Questioning

“Da ist ein Fehler, stopp! Stopp, stopp. Wieso zweimal nach oben?” (P2, unit 2)

“Hä, warte mal.. Geradeaus. Links?” (P2, unit 2)

“Guck mal, ...wes(?) überspringst dann, ein, zwei, dann drehst du dich nach da?! (Betonung auf “da”)” (P17, unit 2)

“Hä, sollten wir hier nicht fünf Mal machen?” (P2, unit 2)

“Hä der ist doch schon groß?” (P1, unit 2)

“Eins nach o- noch eins nach- ja, nach eins, ne, hä? Hä?” (P2, unit 2)

“Das ist doch auch falsch, oder?” (P8, unit 2)

“Der wird doch gar nicht kleiner?” (P8, unit 3)

“Muss das Ding noch da rein? Oh es geht nicht.” (P7, unit 3)

“Hu? Der kommt aber gerade nicht aus der Hand geschossen.” (P7, unit 3)

“Warum hüpfen jetzt- hä? Warum hüpfst er denn?” (P15, unit 3)

“Wo gibt’s diese Klammer? Wo ist die?” (P8, unit 3)

“Wieso fehlt da das Startzeichen?” (P18, unit 3)

“Hä wieso hat Putin jetzt gar nix gemacht?” (P18, unit 4)

“He? Und hier ist noch die Fee da.” (P1, unit 4)

Logical Reasoning

“Ab dann. Aber dann prallt er ja gegen die Mauer.” (P2, unit 2)

“Doch, weil sonst prallen wir hier gegen die Mauer.” (P1, unit 2)

“Wenn er groß wird, dann donnert er gegen den Tunnel.” (P6, unit 2)

“Aber- aber wenn er sich hier wieder groß macht, dann donnert er gegen de' Tunnelwand.” (P1, unit 2)

“Da ist a- das geht wirklich! (in der Tür vom Raum, vorzeigend) Das geht aber, in der Tür kann man stehen.” (P3, unit 2)

“Neeiiiiin! Sonst knallt er ins Hindernis. Dann muss er das üb- Hindernis überspringen.” (P15, unit 2)

“Na damit er über- durch die Tür gehen kann. Nnnnn hehehe. Weil die Tür ja gar nicht steht.” (P5, unit 2)

“Und, aber das dritte ist schon wieder nicht richtig. Weil wenn er hier steht, dann, wenn er hier steht... Eins, zwei, drei ist schon richtig. Aber hier muss er das Feld ja überspringen, sonst knallt er ans Hindernis.” (P15, unit 2)

“Hier... eins, zwei, drei, vier, fünf und und dann einmal nach vorne geht nicht.” (P9, unit 2)

“...weil wenn man ja dann 2 Mal oben geht dann verpasst man ja auch das Krone aufsetzen.” (P4, unit 2)

“Gleich ein Fehler entdeckt. Weil wenn er sich groß macht, dann wird, dann habe ich hier hingelegt, dass er gleich wieder normal wird. Aber dann ist er noch nicht durch die Tür.” (P4, unit 2)

“Dann knallt er beim zweiten Schritt schon hier hin. Und da soll das Ende sein. Hier.” (P15, unit 2)

“Wollt ich hier einfach 5 mal nach rechts sagen. ... Will 1, 2, 3, 4, 5, dann wär er schon direkt hier. Dann noch 1 mal nach unten, einmal - einmal überspringen und dann ist er schon im Ziel.” (P4, unit 2)

“Wir laufen auf dem Mond rum.” (P5, unit 3)

“Hää? Der springt in die Luft und kommt immer... ah, der soll vom Boden starten und springt dann aufs Bett.” (P18, unit 3)

“Die Krabbe sitzt doch aber so schön, in unserem Basketballkorb.” (P4, unit 3)

“Aber das Huhn soll Zuschauer sein.” (P1, unit 3)

“Wieso geht er da so durch die Tür durch?” (P18, unit 3)

“Der läuft gegen die Wand hehehe.” (P15, unit 3)

“Hehehe. Nein, der Hund muss aus dem Fenster springen.” (P14, unit 3)

Reaction to Given Bugged Program

“Hä? Nein es geht ja gar nicht weiter.” (P4, unit 2)

“Was wollen die hier? Hehe.” (P17, unit 2)

E. Original Quotes

“Warte... Überspringen, eins, zwei... hääh... Ich glaube eher, der kommt nicht ins Ziel. Schon hier hängt er ab. Guck! Eins, zwei, also so überspringen. Dann kommt er zweimal nach vorn. Dann geht er einmal nach da und soll fünfmal in die Richtung laufen. Geht aber nicht! Danach dreht er sich so und dann soll er nochmal drei Felder nach vorne.” (P15, unit 2)

“Aber die haben gesagt wir sollen ausprobieren ob es funktioniert! [...] Also es kann ja sein, dass es nicht funktioniert. [...] Nein, ähm, da stand - das stand doch nur, dass ähm ausprobieren sollen ob es funktioniert!” (P5, unit 2)

“Und es funktioniert nicht!” (P4, unit 2)

“Das dürfen nicht. Wir müssen das da nach machen!” (P3, unit 2)

“Das kann man aber auch machen.” (P4, unit 2)

“Hoffentlich funktioniert das Pro - Programm, sonst müssen wir ein eigenes legen.” (P5, unit 2)

“Ich glaube eher, der kommt nicht ins Ziel. Schon hier hängt er ab. Guck!” (P15, unit 2)

Problems Due to the Tablet

“Das Tablet will nicht, wie wir wollen.” (P3, unit 3)

“Ey was du kannst auch wegradieren?” (P4, unit 3)

“Wie löscht man die Dinger eigentlich?” (P7, unit 3)

“Daaann, gibts hier einen Radiogummi?” (P13, unit 3)

“Geht ja nicht! Was hast du?” (P15, unit 3)

“Wo sind wir jetzt gelandet? Hilfe!” (P3, unit 4)

“Und wie kann man die Stempel wieder löschen?” (P6, unit 4)

“Das müssen wir löschen. Wie kann man das löschen?” (P13, unit 4)

“(flüstert) Sooo. (Wischgeräusch und plötzlich reagieren alle ganz erschrocken)... Oh nein, die Schere war doch an!” (P10, unit 4)

“Erstmal müssen wir den doch benennen. Ah ne kann man nicht.” (P11, unit 4)

“Hehe und den kann ich nicht löschen. Scheiße.” (P16, unit 4)

“Hehe. Das Spiel ist doof. Tablets wollen nicht so viel, wie wir wollen.” (P3, unit 3)

“Hä wieso kann man das dahinter nicht löschen?” (P18, unit 3)

“K wo ist hier ein K? Orrrh ey, ich hasse Tastatur!” (P4, unit 4)

“Ich finde aber das Ausrufezeichen nicht. Eyy.” (P4, unit 4)

“Hä wieso malt es nicht mehr?” (P16, unit 3)

“Ja wir wollen.. wir sollen hier den Hintergrund ändern, aber der ändert sich nicht.” (P17, unit 4)

From How does programming and debugging compare in unplugged and block-based programming?

Pair Programming

“Könnt ihr mir mal eins nach unten geben?” (P6, unit 2)

“Gibt es etwa nicht mehr(?)?” (P4, unit 2)

“Springen? Ne. Ich weiß nicht mehr.” (P5, unit 2)

“Hä, er kann die doch auch auflassen, oder?” (P8, unit 2)

“Nach oben, wo ist hier nach oben?” (P4, unit 2)

“Okay wollen wir nicht lieber, dass der dann nochmal nach unten geht?” (P1, unit 3)

“Dann... was kommt jetzt?” (P11, unit 3)

“Drei mal geradeaus, oder?” (P13, unit 3)

“Hallo sagen. Wo war das sprechen?” (P9, unit 3)

“Jetzt das Ende und das- dieses Ende?” (P15, unit 3)

“Und dann brauchen wir doch noch einen Zauberer. Oder?” (P2, unit 4)

“Was brauchen wir noch?” (P4, unit 4)

“Gut, also, was macht der Frosch?” (P4, unit 4)

“Wie viel Mal der (Kleinere?)?” (P8, unit 4)

“Und da muss noch ein größer werden Zeichen.” (P6, unit 2)

“Wir brauchen 1 Mal rechts und 2 Mal dieses Orangene.” (P4, unit 2)

“So also hier muss eine 2 rein...” (P4, unit 2)

“Warte. Ne, das war schon so richtig, wie du es gelegt hast. So. Und dann nach geradeaus.” (P11, unit 2)

“Das brauchen wir! Das brauchen wir! das brauchen wir!” (P17, unit 2)

“Schiebe das Rote doch mal dahin.” (P3, unit 3)

“So. Wir brauchen die noch in blau, wir brauchen die in blau.” (P12, unit 3)

“So. Und jetzt müssen wir noch die Krabbe machen.” (P12, unit 3)

“Das ist glaube ich auf dem hier. Ja. Darf ich? Jetzt könnt ihr machen.” (P9, unit 3)

“Jetzt du. Und da fünf raufmachen.” (P8, unit 3)

“Schreib einfach noch ne 1.” (P18, unit 3)

“Wir müssen noch schnell auf den roten Zauberer machen.” (P13, unit 4)

“Also wir müssen ein Eskimo. (ausatmen) Ich weiß, wie du löschst. Du musst nochmal zurück.” (P18, unit 4)

E. Original Quotes

- “Brauchst du doch gar nicht weil du doch da in der Tür stehst.” (P18, unit 2)*
- “Äh, das brauchen wir gar nicht.” (P15, unit 2)*
- “Ne der muss sich ja erstmal drehen. Schlaumeier.” (P18, unit 2)*
- “Warte. Ne, das war schon so richtig, wie du es gelegt hast. So.” (P11, unit 2)*
- “Da steht’s. Sind nur die blauen.” (P3, unit 3)*
- “Nein das braucht man nicht!” (P13, unit 3)*
- “Du meinst die Klammer.” (P7, unit 3)*
- “Für das Huhn müssen wir zuerst das Anfang.” (P15, unit 3)*
- “Hä hee? Aber das ist doch ein grüner Brief und kein gelber.” (P14, unit 3)*
- “Warum hast du eigentlich jetzt zwei Babys gemacht?” (P4, unit 4)*
- “Ach, [P8]. Uns fällt grad ein, wir ham doch, nein, da kommt doch kein...” (P13, unit 4)*
- “Okay, hier machen wir ein, och ich mach direkt 3 wir brauchen hier 3.” (P11, unit 4)*
- “Guck mal, wir sind- Guck mal. Hier ist der Start. Dann nach oben. Krone aufsetzen.” (P1, unit 2)*
- “Ähm, muss da nicht noch einmal nach oben. Weil guck mal...” (P2, unit 2)*
- “Auf der Stelle hüpfen... Weil, das Feld überspringen ist ja dieses.” (P15, unit 2)*
- “Ja, aber guck, hier so eins, zwei. Und hier muss man eins, zwei, drei.” (P15, unit 2)*
- (P3, P6: beiden anderen, die nicht programmieren zählen die gesetzten Befehle laut mit) (unit 3)*
- “Als erstes muss er ja kleiner werden.” (P1, unit 4)*
- “Genau. Aber Schnecken tempo ist noch nicht. Schaut mal, wenn ihr sie noch klickt, werden sie noch schneller.” (P3, unit 4)*
- “Nein hier müssen wir die! Da hin. Da, nein.” (P14, unit 4)*
- “Erstmal den Start...” (P2, unit 3)*
- “So, und jetzt zweimal nach oben. Da steht’s zweimal nach- darauf springen.” (P3, unit 3)*
- “Als erstes brauchen wir einen Start.” (P4, unit 3)*
- “So jetzt zwei mal nach oben.” (P4, unit 3)*
- “Da muss eine 1.” (P7, unit 3)*
- “Aber warte mal kurz hier grün.” (P12, unit 3)*
- “Als Endes müssen wir’s Schloss machen.” (P11, unit 4)*
- “Stopp, Stopp, stopp. Eiswüste müssen wir jetzt (unhörbar).” (P14, unit 4)*

Thinking Aloud

“(leise) Zwei (unhörbar). (lauter) Zwei nach.” (P2, unit 2)

“Nach.. das.. Dann... das...” (P8, unit 2)

“Ähh, wir brauchn ra... nee, warte mal kurz. Klein machen... ja. Geradeaus...” (P13, unit 2)

“Nein warte er kann... ah ja stimmt... Da muss er sich...” (P17, unit 2) “Ups, zuerst muss das da... Das muss da. Oder?... Nee.” (P15, unit 2)

“Zwei Schritte zzz nach. Zwei Schritte naaaach reecht... nein, links... Und rechts... Dann...mach dich ganz bitte ganz klein.” (P14, unit 2)

“4 nach... Also erst mal... Start.” (P2, unit 3)

“Dann... Warte, wir mach- wir machen das mal zusammen. Viiiier.” (P6, unit 3)

“Muss das Ding noch da rein? Oh es geht nicht.” (P7, unit 3)

“Erstmal brauchen wir das...” (P4, unit 3)

“Ähm... So. Und da-Ne. Ja und dann nochmal sprechen. Sprechblase.” (P12, unit 3)

“(flüstert) Das (unhörbar)” (P8, unit 3)

“(leise) Springen.” (P8, unit 3)

“Das- und jetzt äh äh äh...” (P14, unit 3)

“Sooo. Danaaaach... geht das 6 mal...” (P15, unit 3)

“Ahhh... Hmmhmm... Das hier is erstmal das. Das is schon richtig. Das is das... So, jetzt das... zwei nach (unhörbar), das is ja schon... Und jetzt Ende.” (P16, unit 3)

“Ah, Treffen... Hmm... Zwei hoooch... So, zwei hooch. Ähm... Mensch.” (P16, unit 3)

“1, 2, 3, 4, 5, 6, 7, 8, 9, 10, (beginnt zu flüstern) 11, 12, 13, 14, 15, 16, 17... 17 Kästchen.” (P16, unit 3)

“Äähm... hmhmhmhhmmmm... puuuhhh... so... jetzt kommt das Menschen.” (P16, unit 3)

“Soo. Nee, eins größer. Ffffff, hm, hm, größer machen... (Unhörbar)... Mhm, mhm, mhm, mhm... mhm, mhm, mhm... mhm, mhm, mhm... nee, da kann keiner drauf sitzen. Ffffff... Perfekt, und jetzt muss das noch weg, so. Und jetzt guck ich's mir noch einmal an, oder warte mal... Anfang...” (P16, unit 3)

Testing

“Können wir erstmal durchtesten, wie das dann so läuft?” (P3, unit 2)

“Lass mich kurz gucken. (will testen)” (P17, unit 2)

“Das können wir ja ganz leicht selber testen.” (P17, unit 2)

“So und jetzt testen wir mal.” (P3, unit 3)

“So, und jetzt testen. (P3, unit 3)

“Wir müssen erst mal gucken, ob es geht.” (P3, unit 3)

E. Original Quotes

“Das ist (unhörbar), wir probieren es einfach mal aus.” (P1, unit 3)

“Nee, lass erstmal starten.” (P15, unit 3)

“Ah ja hier haben wir auch schon. Wir müssen die Geschichte doch einmal durchgehen!” (P9, unit 4)

“Los und jetzt lass das Programm mal endlich laufen.” (P15, unit 4)

“Und jetzt testen wir es nochmal. Also erstmal das hier weg.” (P16, unit 4)

Debugging

“Oh jetzt haben wir ein Problem. Vorwärts springen nicht rückwärts springen.” (P3, unit 2)

“Wir haben ein Fehler gemacht! Das darf nicht hier sein! Sondern hier muss das sein!” (P4, unit 2)

“Da fehlt ein geradeaus.” (P9, unit 2)

“Und das hattet ihr nämlich falsch. Muss entweder- da muss ein geradeaus hin.” (P16, unit 2)

“Aber da müssen wir uns doch groß machen.” (P14, unit 2)

“Weil hier ist ja schon das Problem. Er will ja nach da.” (P4, unit 2)

“Eine Drehung nach reeeechtss, ne nach links.” (P5, unit 2)

“Uund... und noch eins nach vorne.” (P8, unit 2)

“Mmh... Dann müssen wir aber jetzt hier ein bisschen zur Seite.” (P15, unit 2)

“Alsooo... ich würde sagen, zweimal überspringen.” (P15, unit 2)

“Wir meh - wir nehmen einfach das hier...” (P4, unit 2)

“Warte tei-, lassen wir es einfach. (fehler wird einfach so gelassen)” (P3, unit 3)

“Ähh das Stopp fehlt. Und wir müssen es langsam abspielen.” (P7, unit 3)

“Nein, das, Ziel. Das Ziel fehlt. Das Ende.” (P9, unit 3)

“Du musst- du musst das zu Ende ran machen.” (P13, unit 3)

“Dann muss die noch einen Schritt weitergehen.” (P16, unit 3)

“Wir haben den Abschluss vergessen.” (P2, unit 4)

“Das Ziel war nicht da.” (P4, unit 4)

“Das is, das is komisch. Der nimmt jetzt nicht das von den anderen. Das fängt ja bei der... Warte, die Eidechse schrumpft erst mal.” (P13, unit 4)

“Guck da stehen dauernd dieses. Verstehst du, deswegen verschwinden die ja!” (P15, unit 4)

“Nee, Szene wechseln kommt da ran.” (P18, unit 4)

“Ja, aber wieso geht das nicht drauf?” (P3, unit 3)

“Nee, wo muss ich das hinmachen?” (P16, unit 3)

“Was? Was muss ich jetzt machen?” (P8, unit 3)

“Hier... Maaaaan, können Sie mal ganz uns mal ganz kurz helfen?” (P13, unit 4)

E. Original Quotes

- “Wie rennt man aufs andere Bild? Wie kann man das wechseln?” (P8, unit 4)
- “Fertig... Okay! Start... Ach neeee, es ergibt kein Siinnnnn! Ääh... (meldet sich für Hilfe)” (P13, unit 4)
- “Oh. Warte. Oh, oh ja jetzt. Warte. Aufs Weiße und ah nee da kann man was beschriften. Keine Ahnung, warte kurz. (meldet sich für Hilfe)” (P11, unit 4)
- “Ähm, wie kann man machen, dass der Drache Feuer speit?” (P6, unit 4)
- “Wir müssen nochmal fragen. [...] Wir wollen, dass der nach diesem Nordländer, weil der sich ja versteckt, dann so ruft 'Zeig dich', aber jetzt gehen wir nochmal auf Putin und da...?” (P18, unit 4)
- “Ja wir wollen.. wir sollen hier den Hintergrund ändern, aber der ändert sich nicht?” (P17, unit 4)
- “Wollen wirs den schwer mach-” (P1, unit 4)
- “Das wird schwer schon.” (P3, unit 4)
- “Wir machen das für die extra schwer.” (P14, unit 4)
- “Bei der Geschichte das kann nämlich ganz schön schwer werden.” (P16, unit 4)
- “Wir machen eine schwere Geschichte für die.” (P13, unit 4)
- “Pffh (ausatmet) Die haben sich ne voll blöde Geschichte gesucht. Bei uns war die ja noch ok.” (P9, unit 4)
- “Eure Geschichte ist eben ein bisschen schwierig.” (P15, unit 4)
- “Eure Geschichten so schwer (unhörbar)” (P17, unit 4)
- “Eure war auch so komisch schwer.” (P18, unit 4)
- “Da gibt es gar keine Höhle. Die haben sich was voll schweres für uns ausgesucht. Das Können wir gar nicht machen.” (P9, unit 4)
- “Aber wie kann man denn den Ball zaubern? (sich fragen ob es diesen Befehl gibt)” (P7, unit 3)
- “Geht hier irgendwo zaubern?” (P1, unit 3)
- “Wo ist das Programm Küssen?” (P3, unit 3)
- “Ähm, wie kann man machen, dass der Drache Feuer speit?” (P6, unit 3)
- “Da mit dem hier, guck mal wenn ihr diese Befehle einbaut, äh m, wartet mal, wir gehen auf zurück und das, dann nehmt ihr den Befehl.” (P8, unit 4)
- “Mach das mal weg. Mach weg. Und dann... [...] Den Befehl.” (P8, unit 4)
- “Dann geht ihr, nehmt ihr den Befehl und das weg. Und nehmt das.” (P8, unit 4)
- “Da mit dem hier, guck mal. Wenn man diese Befehle einbaut... äähm... wartet mal, wir gehen mal zurück auf das, dann nehmen den, den Befehl...” (P13, unit 4)
- “Na ja, sie haben halt. Ich glaub sie haben vergessen das hier einzubauen. Und dann ging das nicht.” (P8, unit 4)
- “Das Ziel war nicht da.” (P4, unit 4)
- “Naja, also wir haben halt... Also wir haben vergessen, das hier einzubauen.” (P13, unit 4)

Mental Simulation and Embodiment

“Wir müssen ja nach oben. Und dann so und dann (leise, überredet) so und so und untendurch und so...” (P1, unit 2)

“Oder so hier. Soooo, soooo, so, so und so.” (P8, unit 2)

“Eins, hm hm hopp und dann hn hn.” (P4, unit 2)

“Einmal vor, dann Feld überspring ding dong ding.” (P3, unit 2)

“Dann da stehen würd ich mich ja so drehen.” (P8, unit 2)

“Nee, nee, nee. Wop, wop. Der ist dann jaa hiiier. Also, müssen wir jetzt hier hin machen. Weil der ist ja hier.” (P13, unit 2)

“Ähmmm, hm hm hm hm hm hm hmhm.” (P16, unit 2)

“Und da eins, zwei, drei, vier, fünf, das ist richtig.” (P15, unit 2)

“Die macht- die macht eher gerade so. (vormachen)” (P3, unit 3)

“Döhh.” (P4, unit 3)

“Düb, düb, döööh. (Geräusche)” (P1, unit 3)

“(macht Pop-Geräusche)” (P5, unit 3)

“Düd. (Geräusch)” (P9, unit 3)

“Düdüdüdüü.” (P18, unit 3)

“Düdüdü.” (P14, unit 3)

“Ppp. (Geräusch)” (P17, unit 4)

“Düd.” (P12, unit 4)

Backseat-Programming

“(flüstert) Du musst da drauf.” (P2, unit 3)

“Und jetzt aufs Rote drücken.” (P3, unit 3)

“Hier hat man die Funktionsliste. Du musst erst mal hier drücken jetzt. Und jetzt hast du hier die Funktionsliste.” (P3, unit 3)

“Da musst du das da machen.” (P5, unit 3)

“Da muss man auf orange, auf orange!” (P1, unit 3)

“Jetzt du. Und da fünf raufmachen.” (P8, unit 3)

“Nein, das musst du zw-.” (P14, unit 3)

“Da musst du 11 runter, da musst du 11 drunter.” (P18, unit 3)

“Los jetzt mach!” (P4, unit 4)

“Und jetzt musst du noch die hier programmieren.” (P15, unit 4)

Distraction by the Tablet

“(nimmt Aufnahme auf) Hallo (unhörbar). Wie geht’s? Wie steht’s?” (P3, unit 3)

“Und hier kannst du sogar malen, deinen eigenen Hintergrund.” (P3, unit 3)

“Hehehe. Können wir jetzt endlich den Zauberer anmalen?” (P7, unit 3)

“Und den Bauch mache ich in rot. Den Hut mach ich in blau.” (P11, unit 3)

“Aber die Katze können wir doch auch verändern!” (P9, unit 3)

“Wollen wir Unterwasser nehmen, oder Urwald?” (P15, unit 3)

“Müssen wir hier! Da können wir malen.” (P3, unit 4)

“Warte, den Drache müssen wir mal mit Farbe...” (P6, unit 4)

“Wir nehmen das Pferd. Das wir dann verändern.” (P10, unit 4)

“Hier müssen wir Emila nochmal bemalen. Grün.” (P9, unit 4)

“Doch, das ist schön... Ja, oder das? Und die Punkte?... Was ist?... Ja, so.” (P13, unit 4)

“Der Ball bewegt sich ja auch. Die Katze muss es ja machen.” (P3, unit 3)

“Der Eskimo- der Eskimo sagt Hallo zu dem Pinguin. Wo geht die Hallo-Taste?” (P1, unit 4)

“Wir müssen nochmal fragen.” (P18, unit 4)

“Wie kann man denn dabei die Hintergründe wechseln?” (P10, unit 4)

“Warte, ich will jetzt den Tic hier reinkriegen. Wie krieg ich das hin?” (P17, unit 4)

“Das müssen wir löschen. Wie kann man das löschen?” (P13, unit 4)

Other Interesting Things

“Eins, zwei, drei und!” (P3, unit 2)

“Wollt ich hier einfach 5 mal nach rechts sagen. ... Will 1, 2, 3, 4, 5, dann wär er schon direkt hier.” (P4, unit 2)

“(leise) Eins, zwei, drei.” (P9, unit 2)

“Fünf. Eins, zwei, drei, vier, fünf.” (P8, unit 2)

“Eine eins, eins, zwei, drei, vier, fünf, sechs. (Setzt mit der Figur) Eins, zwei, drei, vier, fünf, sechs. Okay. Perfekt.” (P12, unit 2)

“Eins, zwei, dreimal musst du nach vorne.” (P15, unit 2)

“Ja, aber es wär doch viel einfacher., wenn er (tippt auf das Spielfeld) eins, zwei... drei, vier, fünf... das wär doch einfacher.” (P14, unit 2)

“Baaatsch... Eins, zwei, drei. Also nur drei.” (P15, unit 2)

“Jetzt dreht er hier. Und da eins, zwei, drei, vier, fünf, das ist richtig. Das können wir auch abbauen. Uund... fünf.” (P15, unit 2)

“Dann ma dann machen wir lieber das Wiederholungszeichen... Eins, zwei, drei, vier, fünf, sechs mal...” (P15, unit 2)

E. Original Quotes

- “220 mal nach rechts drehen.” (P4, unit 3)*
- “Ja das machen wir in einer 55 rein. Soll das (unhörbar) das Ding soll 55 mal machen.” (P7, unit 3)*
- “Wie viel mal wollen wir die Schleife nehmen? 18 mal?” (P7, unit 3)*
- “150(?) mal, ne 100 mal nach oben!” (P7, unit 3)*
- “99.. 10 mal reicht dann nach oben. Da reicht 10 mal nach oben.” (P7, unit 3)*
- “..machen wir bis er aus dem Bild ist. Das sind ungefähr.. machen wir einfach mal... 11.” (P8, unit 3)*
- “Da schreiben wir eine Einhundert rein!” (P14, unit 3)*
- “Hun- schreib da hundertachtzig hin und dann wird das gehen.” (P18, unit 3)*
- “Dann schreiben wir da 66 rein.” (P18, unit 3)*
- “Nein, dann schreiben wir 90. Das ist (unhörbar)” (P14, unit 3)*
- “Also erstmal Flagge” (P3, unit 2)*
- “Baby machen.” (P6, unit 2)*
- “[...] Schneckentempo.” (P3, unit 2)*
- “So, hier ist das Haus.” (P12, unit 2)*
- “Und jetzt die Klammer wieder zu.” (P11, unit 2)*
- “[...] Mehrfach-Ding-Befehl ausführen oder hinlegen.” (P3, unit 2)*
- “Nein der Roboter hat sich verwandelt. Verwandekarte.” (P17, unit 2)*
- “Da ist da so ein TnT drauf, wenn man das macht, ähm, sind alle Hindernisse weg für(?) den Roboter.” (P16, unit 2)*
- “Oder es gibt auch fliegen.” (P17, unit 2)*
- “[...] eine Zeitreise machen bis hierhin?” (P6, unit 2)*
- “Gibts hier auch ein Türöffner?” (P6, unit 2)*
- “Gibt es hier irgendn best- bestimmtes Zeichen um durch die Tür zu gehen?” (P4, unit 2)*
- “Wo ist die alle Lösche-Taste?” (P3, unit 3)*
- “Hahaha! Wir nehmen lieber das da. Löschen. Wo ist der Papierkorb?” (P3, unit 3)*
- “Warte, hier gibt es doch irgendwo eine Löschtaste, oder? Nein, zurück. Wo ist hier Löschtaste?” (P3, unit 3)*
- “Gibt es auch 'n Radiergummi?” (P16, unit 3)*
- “Gibt es bei euch auch diiese Taste?” (P16, unit 4)*
- “Gibts hier ein Radiergummi oder sowas?” (P4, unit 4)*

From Is there a transfer of knowledge from unplugged to block-based programming?

“Ja, das ist ja leicht. Eins geradeaus draufspringen und dann zu Ende.” (P3, unit 3)

“Ähm er rennt,... er sp... [...] Oder hoch? Oder springt. Hoch, ich glaub hoch.” (P11, unit 3)

“Er geht zweimal geradeaus. [...]]Hoch, dann dreht er sich 12 mal, dann geht er zweimal nach unten, dann rennt er wieder...” (P12, unit 3)

“Zwei mal (unhörbar) P13: Zwei mal...” (P7, unit 3)

“Die rennt zwei Schritte geradeaus, dreht sich dann zwölf Mal, nach rechts, geht zwei Schritte zurück, joggt, und springt zwei Mal und dann is Ende.” (P16, unit 3)

“Zweimal nach oben...äääh...Zwölfmal drehen?! (verwirrt) ...Zweimal nach unten.” (P4, unit 3)

“Geht geradeaus. Also sechs mal geradeaus, dann sagt es hallo ähm dann...” (P13, unit 3)

“Hallo. Dann schickt es einen Brief und dann gehts.” (P8, unit 3)

“Und zweimal schrumpfen.” (P4, unit 3)

“Hä wir haben doch schon Schnecken tempo gemacht, [P3].” (P2, unit 3)

“Das ist der Roboter!” (P6, unit 3)

“Moment, wir müssen irgendwie Befehle geben... Also guck hier.” (P15, unit 3)

“Sechs 6 mal geradeaus. Nach da. Nein!” (P9, unit 3)

“Äh, Ziel, 5 gradeaus, über-springt zweimal und dann aufs Ziel.” (P11, unit 3)

“Also ich fang.. erst mal... Wo ist die Zielfahne da?” (P9, unit 3)

“Und 5. Also jetzt mach ich zweimal überspringen.” (P9, unit 3)

“Erstmal die Stadtflagge. So...” (P16, unit 3)

“Schneller werden. Wo ist schneller werden?” (P6, unit 3)

“Man läuft erstmal wie Sonic.” (P4, unit 3)

“Ach, du musst ziehen.” (P6, unit 3)

“Wo ist denn hier angezeigt?” (P3, unit 3)

“Also so?” (P9, unit 3)

“Und wo kann man die jetzt die Befehle...” (P8, unit 3)

“Da muss man ja draufdrücken.” (P18, unit 3)

“Aber erstmal muss man Start nehmen.” (P13, unit 3)

“Du musst- du musst das zu Ende ran machen.” (P13, unit 3)

“Wieso fehlt da das Startzeichen?” (P18, unit 3)

“Erstmal starten. Erstmal Start muss da hin.” (P18, unit 3)

“Nein, das, Ziel. Das Ziel fehlt. Das Ende.” (P9, unit 3)

“Du musst- du musst das zu Ende ran machen.” (P13, unit 3)

“Da ist kein Ende dran, deswegen geht's nicht. Ähm...” (P18, unit 4)

“Erst mal Start. (unhörbar).” (P18, unit 4)

E. Original Quotes

“Ich bin Compiler.” (P6, unit 3)

“Ich bin Programmierer, da bin ich Tablet.” (P3, unit 3)

“Ich bin Compiler, Compiler.” (P6, unit 3)

“Weil ich bin ja der Roboter.” (P2, unit 3)

From Are there differences in debugging between different age groups?

“Ab dann. Aber dann prallt er ja gegen die Mauer.” (P2, unit 2)

“Ähm links. Geradeaus, geradeaus. Hä, hier lang.” (P7, unit 2)

“Wir sollen das so fünf Mal machen.” (P2, unit 2)

“Warte tei-, lassen wir es einfach. (fehler wird einfach so gelassen)” (P3, unit 3)

“Wieso macht das der Ball?” (P3, unit 3)

“Warte vielleicht erst mal. Das hier kommt jetzt erst mal weg.” (P2, unit 3)

“(ausatmen) Wir haben was falsch gemacht. Jetzt kommen wir hier nicht mehr weiter.” (P5, unit 4)

“Oh, wir haben nur eins nach vorne vergessen.” (P13, unit 2)

“Guck mal, ...wes(?) überspringst dann, ein, zwei, dann drehst du dich nach da?!” (P17, unit 2)

“Hä? Die muss sich doch zwölf Mal drehen.” (P8, unit 3)

“Und dann sagt es gleich danach, da ist er ja, obwohl der Frosch was sagt davor.” (P13, unit 3)

“Das is, das is komisch. Der nimmt jetzt nicht das von den anderen. Das fängt ja bei der... Warte, die Eidechse schrumpft erst mal.” (P13, unit 4)

“Aber wie kann man denn den Ball zaubern? (sich fragen ob es diesen Befehl gibt)” (P7, unit 3)

“Geht hier irgendwo zaubern?” (P1, unit 3)

“Wo ist das Programm Küssen?” (P3, unit 3)

“Ähm, wie kann man machen, dass der Drache Feuer speit?” (P6, unit 4)

“An die Ecke. Und wir wollen jetzt dastellen, dass das Abends ist? Die Höhle muss ja eigentlich aufs erste Bild. Das klappt nicht. Das klappt nicht.” (P9, unit 4)

“Ja den können wir als Meister nehmen.” (P14, unit 4)

“(einatmen, aufgeregt) Ja stopp! Wir müssen einfach einmal ein Mensch.. und dann was draufmalen, wir müssen den doch gar nicht selber zeichnen! Wir müssen einfach was dran malen wie dummies. Oh Mann.” (P14, unit 4)

“(liest) “Und wartete, bis Putin ihn entdeckte. Putin entdeckte ihn nicht.” Ähm und machen wir einfach mal so (unhörbar)” (P18, unit 4)

E. Original Quotes

“Also, er konnte sich groß, klein, groß, klein, groß, klein. (verstellte Stimme) Jaja-jajaja.” (P14, unit 4)

“Ja das müssen die dann so richtig dumm malen.” (P14, unit 4)

“Nein, unsichtbar ist er nicht, der versteckt sich. Aber auf dem Bildschirm, sieht das aus als...” (P16, unit 4)

“Nein die gibt es vielleicht gar nicht! Wir wissen es... unter dem Meer.” (P15, unit 4)

“Das muss den auch da drinnen geben. Den..” (P18, unit 4)

“Die wissen nicht wer das ist. Wir müssen jetzt ohne ihn machen, komm. Entweder wir hätten ihn am Anfang mit dazu getan oder jetzt nicht.” (P18, unit 4)

“Die brauchen ganz viele Bilder, verschiedene Hintergründe.” (P13, unit 4)

From Further Exploration

“Hier Jungs. Äh ich bin Compiler.” (P6, unit 2)

“Ich möchte wieder Compiler sein.” (P1, unit 2)

“Ich sage an.” (P11, unit 2)

“Los, damit ich es legen kann.” (P6, unit 2)

“Ja aber ich bin der Programmierer.” (P17, unit 2)

“Soo, ich steuer wieder den Roboter... Und du sagst wieder mal an.” (P15, unit 2)

“Hn eigentlich bist du nur Compiler und Aufbauer.” (P3, unit 2)

“Deswegen muss ich ja alles aufmalen. Yuhuu!” (P4, unit 2)

“Hey P4, ich darf das machen! (malen)” (P5, unit 2)

“Als Roboter musst du das selbe machen eigentlich wie [P5].” (P3, unit 2)

“Hallo? (P17, unit 2)

“(unterbricht) Versteht man sich selber?” (P15, unit 2)

“Nana. Nana. (P9, unit 2)

“Hallo, dummes Mikrofon. Wir können auch noch mal was Witziges machen.” (P3, unit 3)

“Hörst du mich, du, du kleines Diiiiiiiiingsbuuuuuums? (Mikro) Hallo? Hörst du mich? Hallo. Buh.” (P6, unit 3)

“Hallo! Hallo! Hallo! Hallo! Hallo, Hallo Hallo, Hallo.” (P15, unit 3)

“Nein wirklich. Hallo. Programmieren macht sehr viel Spaß. Andorn(?).” (P15, unit 3)

“Die setzen jetzt erst die Krone auf da drüben.” (P6, unit 2)

“Die sind beim selben wie wir gerade.” (P4, unit 2)

“Ich geh mal gucken was die machen.” (P17, unit 2)

“Ey die haben auch Tunnel.” (P6, unit 2)

“So fertig! Wir haben gewonnen.” (P3, unit 2)

“Dieses Programm haben wir schon gemacht.” (P15, unit 2)

“Komm mal wir demonstrieren unserer Stärke.” (P1, unit 3)

“Wir machen jetzt unsere Stärke, los wir zeigen es den.” (P1, unit 3)

“Unser Programm ist das Beste.” (P18, unit 3)

“Wo sind die? [...] Na die andern?” (P16, unit 3)

“Hinten Rückwärtsgang einlegen. Guck mal die andere Gruppe, ob die es richtig macht.” (P3, unit 4)

“Hallo.” (P4, unit 3)

“Hallo.” (P11, unit 3)

“Hallo.” (P12, unit 3)

“Hü ein Apfel.” (P8, unit 3)

“Also, ‘hü ein Apfel’ sagt das Pferd. ‘Hier ein Apfel’ sagt die (unhörbar). Wie sieht euers aus?” P8, unit 3)

“Autsch.” (P3, unit 4)

“Abracadabra.” (P1, unit 4)

“Oh nein!’ Hehe.”(P1, unit 4)