Challenges of End-user Programmers: Reflections from Two Groups of End-Users

Challenges of End-user Programmers: Reflections from Two Groups of End-Users

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. dr. ir. T.H.J.J. van der Hagen,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen
op woensdag 25 september 2019 om 15:00 uur

door

Alaaeddin Ayyoub SWIDAN

Master of Science in Management of Information and Communication Technology,
Cardiff Metropolitan University, United Kingdom,
geboren te Nablus, Palestina.

Keywords: End-User Programming, Spreadsheets, Programming Education

Printed by: PRINTENBIND.NL

Cover: Based on the picture taken by Justin McIntosh, titled: Mural on the Israeli West Bank Barrier, Mas'Ha. on 21 August 2004

This picture is licensed under the Creative Commons Attribution 2.0 Generic license

The picture was edited into a cover thanks to Shirley de Wit

ISBN ...

An electronic version of this dissertation is available at
`http://repository.tudelft.nl/`.

CONTENTS

1 Introduction **1**
1.1 Challenges in End-user programming . . . 2
1.2 Types of End-User Programmers . . . 4
1.3 End-user Programming: Spreadsheet Programmers . . . 5
1.4 End-user programming: School-age children . . . 7
1.5 Research Questions. . . . 11
1.5.1 Spreadsheet Users Challenges . . . 11
1.5.2 School-age Children Challenges . . . 11
1.5.3 Research Questions. . . . 12
1.6 Methodology . . . 13
1.6.1 Ethical Research and Data Collection . . . 14
1.7 Chapters and publications . . . 16

2 Spreadsheet Performance **19**
2.1 Introduction . . . 20
2.2 Background and motivation. . . . 20
2.2.1 Motivation . . . 21
2.3 Context . . . 22
2.3.1 Shortfall Analysis . . . 22
2.3.2 Features of the Model . . . 22
2.4 The Case Study. . . . 23
2.4.1 Design . . . 23
2.4.2 Analysis . . . 24
2.4.3 Implementation. . . . 24
2.4.4 Evaluation . . . 25
2.4.5 Maintainability of the Modified Model . . . 27
2.5 Related Work. . . . 27
2.6 Lessons Learned . . . 27
2.6.1 Challenges . . . 28
2.6.2 Opportunities. . . . 28

3 Spreadsheet Data Extraction **31**
3.1 Introduction . . . 32
3.2 Background . . . 33
3.2.1 Cross-tables . . . 33
3.2.2 Extraction and Transformation of Spreadsheet Data . . . 34
3.2.3 Motivation . . . 35

3.3 An Exploratory Analysis of Cross-tables in Industry . . . 36
3.3.1 Setup . . . 36
3.3.2 Criteria . . . 36
3.3.3 Results . . . 36
3.4 Approach . . . 37
3.5 Evaluation: Case Study . . . 41
3.5.1 Context and Dataset . . . 41
3.5.2 Results . . . 42
3.6 Related Work . . . 44
3.7 Discussion . . . 44
3.8 Conclusions and Future Work . . . 44

4 Identifier Naming Practices in Scratch 47
4.1 Introduction . . . 48
4.2 Related work . . . 50
4.3 Relevant Scratch Concepts . . . 51
4.4 Research design and dataset . . . 53
4.4.1 Overall design . . . 53
4.4.2 Dataset . . . 54
4.4.3 Identifier Extraction . . . 54
4.4.4 Data Analysis . . . 54
4.5 Results . . . 55
4.5.1 Replication Studies . . . 55
4.5.2 Scratch-specific Constructs . . . 60
4.6 Threats to Validity . . . 65
4.7 Conclusion . . . 65

5 Programming Education to Preschoolers 67
5.1 Introduction . . . 68
5.2 Background: Preschoolers Education . . . 68
5.2.1 Constructivism . . . 68
5.2.2 Piaget's theory . . . 69
5.3 Study Goal . . . 71
5.4 Setup . . . 71
5.5 Materials . . . 71
5.5.1 Sorting and Classification . . . 71
5.5.2 Robot Turtles . . . 72
5.5.3 Ozobot . . . 73
5.6 Observations . . . 74
5.7 Observation Analysis . . . 77
5.7.1 Piaget's theory in agreement with the observations . . . 77
5.7.2 Piaget's limitations in disagreement . . . 78
5.7.3 Age factor . . . 79
5.7.4 Motivation . . . 79
5.8 Conclusions . . . 79

6 Programming Misconceptions for School Age Students 81
6.1 Introduction . . . 82
6.2 Background . . . 82
6.3 Setup. . . . 83
6.3.1 Participants. . . . 83
6.3.2 Study environment . . . 83
6.3.3 Misconceptions selection . . . 84
6.3.4 Questions . . . 86
6.4 Results . . . 87
6.4.1 Most common misconceptions . . . 87
6.4.2 Insights from children's explanations . . . 88
6.4.3 Effect of age and previous programming knowledge . . . 94
6.5 Discussion . . . 95
6.5.1 General observations . . . 95
6.5.2 Scratch-specific issues . . . 96
6.5.3 Reflections on and implications of the results . . . 96
6.5.4 Threats to validity . . . 97
6.6 Conclusions . . . 97

7 Effects of Vocalizing Code on Comprehension 99
7.1 Introduction . . . 100
7.2 Background and Related Work . . . 101
7.2.1 Reading Aloud and Comprehension: Natural Language Perspective . 101
7.2.2 The Role of Voice in Programming and CS Education. . . . 101
7.2.3 Bloom's Taxonomy in CS Education . . . 102
7.3 Methodology . . . 103
7.3.1 Setup. . . . 103
7.3.2 Participants. . . . 103
7.3.3 Lesson Design and Materials . . . 104
7.3.4 RA Design and Implementation. . . . 104
7.3.5 Assessment. . . . 105
7.4 Results . . . 106
7.4.1 RQ1: What is the effect of RA on the Remembering-level? . . . 106
7.4.2 RQ2: What is the effect of RA on the Understanding-level? . . . 107
7.4.3 RQ3: How do students follow the vocalization guideline when they read code later?. . . . 107
7.5 Discussion . . . 110
7.5.1 Reflection and explanation of the results . . . 110
7.5.2 RA and the granularity of the vocalization . . . 110
7.5.3 Threats to validity . . . 110
7.6 Conclusions and Future Work. . . . 111

8 Conclusion 113
8.1 Contributions. 114
8.2 Revisiting the research questions 114
8.2.1 [RQ1.A] How do spreadsheet users deal with challenges they face when working with spreadsheets?. 115
8.2.2 [RQ1.B] How can these challenges be addressed?. 115
8.2.3 [RQ2.A] How do school-age children deal with challenges they face when learning to program? 116
8.2.4 [RQ2.B] How can these challenges be addressed?. 117
8.2.5 Comparing the two groups 117
8.3 Future Work 119

Summary 121

Samenvatting 123

Bibliography 125

Curriculum Vitæ 143

List of Publications 145

Acknowledgments 147

1 Introduction

The use of software is an essential part of our everyday life. Statistics show that a prevalent percentage of Dutch population use email communication (87.7%), buy products online (65.8%), and use text editing software (69.4%) [44]. Many software platforms allow people to create customized software for their personal and professional needs. When people do so, they are categorized as end-user programmers. For example, an administrative assistant creating a spreadsheet to manage meeting-rooms availability, a business analyst writing queries to retrieve data from a database, a web designer creating a template for a website, etc. Estimates show that there are millions of end-user programmers nowadays [99, 194].

Who are end-user programmers? Research on end-user programming has shown progress during the past years. Nevertheless, there seems to be vagueness on whom we can call an end-user programmer. An end-user programmer is a *"programmer"* in the sense that they create, extend and modify programs. The program is defined under the end-user programming paradigm as broadly as a *"collection of specifications that may take variable inputs, and that can be executed (or interpreted) by a device with computational capabilities"* [124, p. 4].

By using definitions from previous research, we start by highlighting the main two characteristics that distinguish end-user programmers from other software users and professional developers:

End-user programmers program for their *personal use*.

This intent is the most distinctive aspect of end-user programming [124]. End-user programs are created for personal use, with no aim to share it publicly or with other people to use or extend. Ko *et al.* [124] stress that when those programs are eventually shared, they grow gradually out of the end-user programming definition. When a program gets shared, it automatically requires additional measures to ensure its correctness and to ensure a minimum level of quality.

End-user programmers program as a means to an end.

They do so to achieve another goal and not for the sake of programming [99, 114].

An in-depth look into the end-user programmers' motivations, Blackwell [28] highlights that there exist *intrinsic* and *extrinsic* factors that play a role into why end-user programmers program.

> The intrinsic factors depend on the personality traits of people which can be modeled using the five-factor or *"Big Five"* theory [113]: openness to experience, conscientiousness, extraversion, agreeableness, and neuroticism. When studying the personal intrinsic motivation of end-user programmers Aghaee *et al.* [3] divide them into three personal categories: *bricoleurs* who enjoy building software as part of their joy in making things in general, the *artists* who believe making software is an outlet of their creativity, and the *technophilia* who like programming as part of their affection for technology.
>
> The *extrinsic* motivation stems from the perceived usefulness (this will help me do something). Previous research considers the perceived usefulness as one of the main reasons why end-user programmers program for themselves. They create programs that help to perform better in their paid jobs by completing a task more efficiently or with less human intervention so that they could focus on another task. However, these benefits come with some cost, namely the investment end-users need to put into creating the programs. As a result, en d-user programmers are always in a state of cost-benefit evaluation to decide whether they do a programming activity or not. Intrinsic factors, such as self-efficacy, affect the cost-benefit evaluation. Self-efficacy is the perceived belief that one will succeed in doing something before actually doing it. Therefore, if you believe you will fail in doing something, then it is less likely that you invest time and attention in it. Self-efficacy has its roots in social and gender contexts [36, 80, 111] and thus differ accordingly between people.

Looking at the definition of end-users and the motivations behind their programming activities helps us understand why end-user programming is diverse, broad and widespread: end-user programmers vary in their motivations to program, goals to achieve, and tools they use. End-user programmers can be realistically anyone: financial modelers, secretaries, teachers, scientists, smart-home owners, musicians, and students [114, 124, 172].

The definition is not limited to adults: children can also be considered end-user programmers. Many software platforms and environments are dedicated nowadays to children allowing them to create artifacts such as games, animations and visual stories [59, 65, 162, 172]. Children are one group of end-user programmers worth special attention. Experiences with computing could be a starting point for some, while they build a barrier for others. Therefore in addition to studying how to improve the children end-user programmers experience now, we should investigate how the current experiences would potentially affect the future.

In this dissertation, we seek to analyze and address the difficulties and challenges that end-user programmers face. We investigate two groups of end-user programmers: spreadsheet programmers and school-age children.

1.1 CHALLENGES IN END-USER PROGRAMMING

While end-user programmers and professional developers differ in goals, they perform similar programming activities to build, extend or maintain programs [99, 114, 124]. For example, both end-users and professional programmers seek to find the best function, library or API for their programs to perform certain functions. They also aim at verifying and testing that their program works according to some specifications, and they debug their programs when it generates any unexpected results.

End-user programmers, however, are more vulnerable to difficulties and challenges compared to the professional developers. There are three overarching sources of the struggles and challenges that end-user programmers face when creating and maintaining software artifacts.

- **System support:** The end-user systems provide little support to the end-user programmers in comparison to professional developers who benefit from a variety of tool-support within the software development IDEs.
- **Peer support:** There are few collaborative software development processes which facilitate multiple developers contributing to one feature and reviewing other developers work. On the contrary, end-user programmers environments and culture are missing out on the benefits that collaboration bring.
- **CS Education:** The end-user programmers lack the necessary programming education and training before building the software artifacts.

Ko *et al.* [125] identify six *"barriers"*, or challenges, that make it difficult for the end-users to learn-on-the-job how to create and maintain their software artifacts. These challenges are:

1. The **Design barrier** which groups the difficulties in using programming problem-solving methods and concepts.
2. The **Selection barrier** which stems from the end-users knowing what they want to do but they do not know what software, library or what function to use.
3. The **Coordination barrier** that stems from the difficulty to understand how various programming components can be combined to achieve one higher-level goal.
4. The **Use barrier** which stems when the use of some parts of the user interface (UI) are ambiguous and vague.
5. The **Understanding barrier** which refers to the invalid assumptions, or misconceptions, that the end-users could hold before working on a program.
6. The **Information barrier** which hinders the end user's ability to find useful information that could help in verifying or writing programs.

With that in mind, and because end-user programmers are goal-driven and create programs with private-use intention, there tends to be less emphasis on the quality of the final product as long as it delivers the required functionality. As a result, end-user programming artifacts inevitably suffer from quality issues. This is confirmed by previous research, which shows that end-user programs often contain errors or have design issues that affect their correctness, understandability, and maintainability on the long run [45–47, 99, 124, 164].

These barriers will recur in various of the chapters in this dissertation. We will encounter them in our analysis of spreadsheet programmers and school learners. They offer a framework that we will use at the end of the dissertation to organize and contrast several of our key findings.

Table 1.1. A 2011 List of End-User Programmers and the Kinds of Programs from Ko et al. [124]

Class of people	**Activities of programming and tools and languages used**
System administrators	Write scripts to glue systems together, using text editors and scripting languages
Interaction designers	Prototype user interfaces with tools like Visual Basic and Flash
Artists	Create interactive art with languages like Processing (http://processing.org)
Teachers	Teach science and math with spreadsheets[159]
Accountants	Tabulate and summarize financial data with spreadsheets
Actuaries	Calculate and assess risks using financial simulation tools like MATLAB
Architects	Model and design structures using FormZ and other 3D modelers
Children	Create animations and games with Alice [67] and Scratch
Middle school girls	Use Alice to tell stories [119, 120]
Webmasters	Manage databases and websites using Access, FrontPage, HTML, Javascript
Health care workers	Write specifications to generate medical report forms
Scientists/engineers	Use MATLAB and Prograph [Cox et al. 1989] to perform tests and simulations
E-mail users	Write e-mail rules to manage, sort and filter e-mail
Video game players	Author "mods" for first person shooters, online multiplayer games, and The Sims
Musicians	Create digital music with synthesizers and musical dataflow languages
VCR and TiVo users	Record television programs in advance by specifying parameters and schedules
Home owners	Write control schedules for heating and lighting systems with X10
Apple OS X users	Automate workflow using AppleScript and Automator
Calculator users	Process and graph mathematical data with calculator scripting languages
Managers	Author and generate data-base backed reports with Crystal Reports

1.2 TYPES OF END-USER PROGRAMMERS

End-user programming is widespread across all aspects of our lives and especially in professional domains. End-user programming has many applications in various domains. Table 1.1 shows a partial list, from 2011, of end-user programmers and some of their activities taken from Ko et al. [124]. For example, in the financial and insurance sectors we see spreadsheets as dominant in tasks related to modeling and data analysis. In engineering and research we see Matlab and LabView used to create simulations and designs. In education, teachers and students use the end-user programming environments as an education tool as part of learning other disciplines. For example, the use of AgentSheets and AquaMOOSE to teach mathematics [76, 202], the use of Visual Basic for Applications (VBA) to teach Physics [54], the use of StarLogo to tech biological modeling to novices [123], and the use of Nuterpea, a domain-specific language, to teach music [171]. End-user programming, however, keeps expanding. Since the collection of the list in Table 1.1, new forms of end-user programming have emerged. For example, data scientists'use of open source literate programming tools like Jupyter notebooks and knitr [122].

In most cases, the software systems for end-user programmers are visual. With the assumptions that end-users are goal-driven and lack prior programming knowledge, end-user systems aim to lower the barrier to programming by eliminating the need to write or remember complex syntax. Visual software environments aim to provide a *"low-floor"*: easy to step in by people with little to no programming knowledge, and *"high-ceiling"*: allow the creation of complex programs that satisfy all the needs of the end-user.

In this dissertation, we will focus on spreadsheet programmers and children. We decided to study spreadsheets because they are widespread across various domains and users. Additionally, spreadsheets require little prior knowledge in programming yet they are used widely in building data-intensive models professionally.

We furthermore studied children programming since this adds the perspective of age and its effect on developing programming skills while at the same time developing the basic cognitive skills as a child. We believe that those skills learned during programming at young age could translate well to other settings. However, to do so those students should get explicit instructions and training on using those skills within in the target contexts.

1.3 End-user Programming: Spreadsheet Programmers

Spreadsheets are by far the most popular end-user platform. Microsoft Excel, the most-used spreadsheet software, has an estimate of 500 to 800 million users worldwide [132]. The Dutch Central Bureau of Statistics [44] reports that in 2018 49.2% of the Dutch population use spreadsheets as a tool, while 30.8% use spreadsheets with calculation formulas. In comparison, 7.4% of the population reported that they wrote software programs in the same period. From an organizational perspective, Panko [163] estimates that 95% of US companies use spreadsheets for financial reporting while Winston [235] suggests that *"around 90 percent"* of auditors' analysis is done using spreadsheets.

End-user programmers appreciate the powerful calculation functionalities in spreadsheets. But most importantly, they appreciate the flexibility and freedom that spreadsheets provide them with, to convey whatever ideas they have into the intuitive User Interface (UI). Baxter [18] argues that end-users are being limited by typical Information Systems with their strict routines and functionalities. Business processes, on the contrary, strive to fulfill the needs of customers in a competitive, time-demanding and continuously-changing environments [61, 200]. With spreadsheets, the end-user can rely on the result of a series of calculations being shown immediately without sending detailed specifications, discussing scenarios or waiting for buggy software to be fixed. Flexibility, freedom, and immediateness, as a result, are the most powerful features that make spreadsheets popular among business professionals according to Hermans [95].

Spreadsheets are used in businesses in a similar way to software: they perform calculations on some input data, and the results afterward become a part of decision-making processes; do we buy stocks for product A or product B? To what extent do we increase the student acceptance rate next academic year? Do we have enough budget for event A or event B?

With the prevalent use of spreadsheets in various businesses and their important role in decision making in organizations, concerns arise over the quality of created spreadsheets and their erroneous nature. According to Rahalingham et al. [180], the chance of finding errors in a spreadsheet exceeds 90% in many reported cases. They call it the *"spreadsheet error phenomenon"* to indicate that spreadsheet errors are prevalent and widespread. Panko stresses that *"spreadsheet error rates are unacceptable in corporations today"* [164, p. 1]. Errors in spreadsheets are, however, the tangible effects of deeper quality issues which failed in preventing the error from occurring in the first place. These errors in spreadsheets often lead to critical consequences that include direct financial losses and reputation damages to companies and employees alike. The European Spreadsheet Interest Group EuSpRIG [77] dedicates a continuously updated page of spreadsheet *"horror stories"* which show the damaging effects of some reported spreadsheet errors. In March 2018, for example, a drinks company filed bankruptcy after financial mismanagements, one of which is an arithmetic error in a spreadsheet that caused a deficiency of £5.2M. Earlier in 2017, an employee in the aircraft company Boeing emailed a spreadsheet template to his spouse. The spreadsheet turned out to contain the personal details of 26K workers in hidden columns, which is unauthorized access to their private data.

Various research aims at reducing the errors and minimizing the quality issues in spreadsheets. We can summarize these efforts into two main lines of research:

1. To provide end-users with more system- and tool-support: Spreadsheets are software; they contain errors and have quality issues in the same way software artifacts created by professional developers do [99]. The powerful features of flexibility, freedom, and liveness, are the same features that, according to researchers [99] make it easier for spreadsheet users to introduce mistakes. Therefore, the best way to improve the quality of spreadsheets is by providing the end-users with a comparable level of software tool-support as the developers in their IDEs. As a result, the challenges that spreadsheet end-users face in creating and maintaining spreadsheets are eliminated.

2. To provide end-users with more knowledge and context support: While creating spreadsheets is indeed very similar to creating programs, the users differ. Software programmers, on one hand, are mostly skilled professionals who are educated and trained to build software. Spreadsheet end-users, on the other hand, are typically educated and trained to perform other types of jobs and, while they do programming activities in their jobs, they do it for a partial amount of time and as a way to perform their original job better and more efficiently. Grossman [91] stresses that a spreadsheet professional cannot be a programmer even after gaining considerable experience in spreadsheets. To mitigate the spreadsheet issues here, we should follow other measures that consider the human, not the tool, in their center. This includes among others investing in extra training for spreadsheet professionals, and applying organizational policies and procedures that regulate the use of spreadsheets.

RELATED WORK

Looking at the literature of spreadsheet research we observe that previous research is more focused on the first line of research we mentioned earlier: providing system- and tool-support for end-users. Inspired by software engineering methods and practices [99, 124], End-User Software Engineering (EUSE) emerged to support spreadsheet end-user programmers in areas such as debugging, testing, version control, and comprehension. One of the early adaptations from software engineering to spreadsheets was What You See Is What You Test (WYSIWYT) [189]. As a formal testing approach, the end-user requires no knowledge in testing theories, and all they have to do is to validate cells as being correct, the system then visually present the *"testedness"* of the spreadsheet.

Related is the work of Hermans et al. [103] in identifying smells in spreadsheets. A code smell is one characteristic of the source code that indicates a deeper problem [83]. Code smells function perfectly as they are supposed to but are typically poor implementation choices that are opposite to design patterns. One example of a code smell is a long method containing dozens of lines of code. The long method, in this case, is not a problem on its own, however, it will be difficult to understand and maintain in the long run. The identification of code smells was a keystone on developing more automatic tools that highlight smelly components and formulas, in addition to suggesting fixes [98, 109].

Visualizing the content of a spreadsheet using class diagrams and data flows [63, 95, 102] is another area of research that focused on improving the spreadsheet comprehension by adding a higher level visual representation of the spreadsheet, clarifying the relation between their elements.

There is little focus on the second line of research, concerning the providing of knowledge and context support for spreadsheet end-users. While tool and system support is important, spreadsheet end-users need a broader organizational support. One aspect of this support is the formal training, education and learning-on-the-job for end-user programmers. Brancheau and Brown [33] provided an early model for End User Computing (EUC) management. The model consists of two core components namely the organization and the end-user. The organization factors include strategy, technology and management action. Formal training is one of the support services an organization should provide as part of its management actions within the EUC context.

Training of end-user programmers is directly related to their acquiring of a higher level of competency. A survey of end-user programmers [239] showed that the *"completion of domestic or overseas education and training related to computing"* is a significant factor in determining the competency of a spreadsheet end-user. Older studies [33, 53, 157, 211] have shown that end-user training in general raises end-users' computer-abilities but only help them in using the computer tools efficiently. A more recent study [130] concludes that providing training to end-user programmers raises the logical reasoning cognitive level and, as a result, leads to creating better spreadsheets with fewer errors.

Although training and learning-on-the-job should be part of the organizational management actions, there is a lack in providing such support in reality. In their early model, Brancheau and Brown report an overall lack of EUC support services in the field. Many years later other studies raised the same concern: Lawson et al. [134] point out that *"training programs are an exception rather than the rule, with no more than a few days of training each year generally offered in most organizations"*. While Coster et al. [60] report that *"formal policies and procedures in most companies are still lacking for most of the stages of spreadsheets"*. In the latter study, the survey showed that policies related to developer training, for example, are in place in fewer than 25% of the companies.

In this state of the art, we miss knowledge on how spreadsheet users should handle and overcome challenges in emerging and unconventional areas of importance to them such as performance and data extraction of unstructured and semi-structured data, which we address in this dissertation.

1.4 END-USER PROGRAMMING: SCHOOL-AGE CHILDREN

End-user programming is not limited to adults. Software-based technologies have reached almost all aspects of our daily life. Children, the *"web-generation"* [172], have become active users of software products either at schools or household environments. As Table 1.1 shows, Ko et. al. have identified children in multiple contexts as end-user programmers.

We see that many software and programming environments are dedicated to children as their primary audience group. In this regard, Kelleher et al. [120] categorize these software environments into three categories based on the motivation behind creating them: one, as a way for children to explore and develop their thinking in areas not related to programming such as music, language, mathematics, and science. Two, as a way to self-expression via the creation of games, simulations and virtual characters. Third, as a way to prepare for a computing-based career. Software environments which fall under the first two categories are the most popular among children. Examples of these environments include Logo, KidSim, EToys, Scratch[1], Storytelling Alice[2], and GameMaker[3]. In October 2017, Scratch was rated number 14 on the TIOBE programming languages popularity index. Currently[4], Scratch is rated 29th on the index, ahead of programming languages such as Lisp, Scala, Fortran, and Prolog. Scratch online platform has over 45 million registered users with more than 43 million shared projects[5].

Most of the current programming environments for children are inspired by the concept of constructionism that was introduced by Papert [165, 166]. Constructionism is a learning theory that states that the building of knowledge, in any context, happens best when the learner is involved in making a tangible and observable thing [155, 166, 199]. Piaget's theory of children's cognitive development [173] is one of the major contributors to constructionism. According to Piaget, the child's personal experience and social relations drive the mental processes that eventually lead to the cognitive development. As a result, the environment where the child finds the opportunities to build and expand their experience and peer-relations, is an essential factor [155].

Within constructionism, programming becomes more of a means to a goal than a goal in itself: to develop new ways of thinking and learning [155]. It is no surprise then that children's programming environments often stress this learning outcome when, or after, using the environment. For example, Scratch suggests that it *“helps young people learn to think creatively, reason systematically and work collaboratively”*. GameMaker claims to be *“the perfect tool for teaching students of all ages how to make games”*.

One major new area of thinking that programming helps to develop is Computational Thinking (CT). Computational Thinking (CT) is the set of concepts that help us as humans to understand the language of computer-based technologies. Wing [234] is one of the first researchers who defined the term of Computational Thinking as the use of abstraction, automation, and analysis in problem-solving. Manovich et al. [142] put CT as being about numerical representation, modularity, automation, variability, and transcoding. In any case, practicing programming is at the core of CT: it is the tool that supports the cognitive tasks involved in computational thinking, and it is the way for individuals to demonstrate computation competencies [191]. The goal of introducing programming at this young age is the development of computational thinking. These programming environments provide a very rich environment for children where CT skills are exercised and developed in natural contexts to them: playing, making, and having fun [155].

With our increasing dependence on computational technologies, many believe that computational thinking is essential for all kinds of jobs from now on [16, 234]. Wing believes that CT is going to be a skill used by everyone by the middle of the 21st century [234].

[1]https://scratch.mit.edu
[2]https://www.alice.org
[3]https://www.yoyogames.com/gamemaker
[4]Retrieved August 21, 2019, from https://www.tiobe.com/tiobe-index/
[5]Retrieved August 21, 2019, from https://scratch.mit.edu/statistics/

This broad range of needs come as Computational Thinking is a transferable skill when certains measures and conditions are met, including for example the use explicit instruction within the targeted context [169]. Taking those requirements into account, developing Computational Thinking in children via the use of programming environments could be beneficial to developing the skills and knowledge of other domains away from programming itself [16, 89, 230, 238]. One study, for example, found that using Scratch in teaching mathematics helped students in improving their performance in mathematics processes such as modeling, reasoning, and problem solving [39]. Research stresses the importance and criticality of experiences in school-age on future interests in general, and the engagement with Science, Technology, Engineering and Mathematics (STEM) fields, in particular, [92]. Overall, developing children's CT skills in their school-years opens the doors for them to a broader range of future opportunities.

Just like spreadsheet programmers, children as school students have to deal with barriers and challenges all end users face. In particular, there are two aspects of these challenges. The first aspect is very similar to what spreadsheet end-users face: namely the lack of system and tool-support. We pointed out that end-user programming environments are historically more visual, with the aim of lowering programming barriers for the end-users. Programming environments for children have the same characteristics. In the visual environments, there is a level of abstraction for concepts into visual shapes and colors. However, children, as we discussed earlier, are still developing their computational thinking skills which among others involves abstraction. This requirement means that children find it more difficult to navigate in a visual block-based environment without support from the environment itself. We find that the support from these environments is lacking.

The second aspect is unique to the children as a group of end-users. As noted earlier, there is an assumption, boosted by the programming environments, that children develop theri computational thinking skills when using programming environments to construct and build. We still, however, lack sufficient knowledge about how, at what age, and by what tools children develop their conceptual knowledge that link to computational thinking. Papert stresses that when dealing with children programmers, research suffered from focusing on *"technocentric questions"* thereby overlooking how programming was used and at whom [165]. While it sounds beneficial for our societies to invest in empowering the children at the school-age, failures due to unaccountable challenges could alienate children from computing technologies and cause other issues. For instance, we know from previous research that children at this age are vulnerable to misconceptions and inaccurate beliefs about science in general [58]. Those beliefs and incomplete knowledge can grow with the children to build the barriers we mentioned earlier (see Section 1.1) especially understanding and information barriers.

Related Work

Looking at the literature of challenges faced by children learning to program, two lines of research appear:

One line of research involves **studying the software quality attributes of children-created artifacts**. This provides children with tools that allow them, similar to spreadsheet end-users, to create better quality programs with fewer errors. For example, Aivaloglou and Hermans [4] analyzed a repository of 250,000 Scratch programs to check for code smells that exist in Scratch programs. They conclude that Scratch programs do suffer from code smells including large scripts and unmatched broadcast signals. Moreno and Robles [150] did a preliminary study with school students to identify bad programming habits in Scratch projects. They found that the habits of keeping the default name of Scratch objects and repeating scripts are the two most occurring bad habits. Automatic assessment tools have been created to support this process of automatic assessment, for instance Hairball [31], Dr.Scratch [152] and Quality Hound [218]. Those tools aim to detect almost the same problematics issues in scratch programs referred to as code smells, bad habits or unsafe practices, respectively. In general, research in this area argues that such issues could stick with the children and be a cause for software quality issues in the future [96]. A slightly different approach is the tool iSnap [176] aimed at helping children complete their programs when they are stuck by providing automatic hints and feedback. iSnap specifically focuses on blocks that are missing from a correct solution, or blocks that are misplaced.

The second line of research involves **the identification and analysis of the challenges that children face in developing programming concepts during their programming activities**. Dr.Scratch [152] analyzes Computational Thinking levels of Scratch programs. For that, the tool assesses seven aspects of a program: abstraction, logical thinking, synchronization, parallelism, flow control, user interactivity, and data representation. A preliminary study analyzing 100 random Scratch projects using Dr.Scratch [151] found that the programs score higher on flow control, abstraction, parallelism, and synchronization, while they score lower on user interactivity and data representation. A study on 250,000 Scratch projects [5] found that Dr.Scratch mean score for the programs in the dataset is 8.9 out of 21, which is a relatively low score. However, the scores in those studies do not take into account the type of a Scratch program. For instance, storytelling projects tend to score low in components of logical thinking, while games would score higher. Wilson *et al.* [233] focused on analyzing games developed by primary school students in Scratch. They evaluated and coded 29 games developed by 60 students after following an 8-week Scratch course. The evaluation was done following a scheme that is split into three main categories: programming concepts, code organization and designing for usability. The study found that the mostly-used programming concepts are sequencing, event handling, and conditional statements. Variables, coordination, and iteration were less used while keyboard input and random numbers were rarely used. Seiter and Foreman [198] studied the progression of Computational Thinking of primary school students across multiple grades. They analyzed 150 Scratch projects for those students assessing what they called Evidence Variables, which include programming concepts such as variables, sequence & looping, and conditions, and Design Pattern Variables which include components such as animating motion and looks, colliding and user interactions. Each component has a scale from one to three or Basic, Developing, or Proficient. Their findings indicate that Data Representation (variable referencing and assignment) is not present in the analyzed programs until later grades (older students). For the Collide design pattern, students at a later grade were able to show basic-level mastery while the advanced two levels were only observed for a few students. It indicates that the Colliding design pattern is difficult as it requires algorithmic thinking, modularization, and synchronization for the three scales.

In this state of the art, we miss knowledge on how school-age children, as a special group of end-users, handle and overcome challenges that relate to developing programming and computer science concepts, which we address in this dissertation.

1.5 Research Questions

In the previous sections, we have seen how both spreadsheet and children end-users have to deal with challenges and barriers related to the lack of programming environment support for the end-user in general, and the lack of solid knowledge on how programming affects the development of computational thinking in children in particular.

The main goal of this dissertation is to explore how end-user programmers deal with the challenges they face while building their software artifacts. Considering more than one group of end-user programmers is essential to gain a broader understanding of this diverse community [28]. In this dissertation, we look at two groups namely spreadsheet professionals and school-age children.

1.5.1 Spreadsheet Users Challenges

To obtain a thorough understanding of the challenges spreadsheet users face, we focus on two aspects. First, we address spreadsheet performance. This is an often overlooked aspect in spreadsheet end-user programming research. However, with the large amount of data involved in spreadsheets and the need for more accurate models, the execution time of a spreadsheet becomes increasingly important.

Second, we look at data extraction. This is an important problem since some of the critical organization data are only available in spreadsheets. Extracting data from spreadsheets has many applications in areas related to auditing, version control and automatic migration away from spreadsheets. While extracting data from spreadsheets has been well covered in spreadsheet research, little has been done so far on cross-table data structures in spreadsheets.

1.5.2 School-age Children Challenges

Our target group is school-age students as young as 4 years old. When investigating challenges that children face while learning to program, we follow a different approach that is focused on the exploration and investigation of challenges and difficulties our target group face, without the rush to provide solutions. This is because children are still developing their basic cognitive skills while learning to program. Therefore, challenges that they face could be so complex that you need more time to explore and investigate rather than apply remedies directly.

We focus on the Scratch programming environment since it is one of the most popular environments for this age group, with much data available for research. However, we also investigate other physical and robotic programming environments that target very young children. In addition, we study student experiences in Python as one mainstream textual language that gives, opposite to visual programming languages, more importance to syntax.

A particularly interesting group of children is that of young children or pre-schoolers, who have yet to develop basic cognitive skills related to language and directions for example. We know from theories of children's cognitive development, Piaget's theory and its derivatives for instance, that children pass through four phases when developing a certain skill. There is need, however, for more work to understand how children learning to program develop their pathways and move from one phase to another. Yet, many puzzles, games and robotics-based environments target those children as their primary audience, promising them to learn programming.

For older children, we investigate the naming patterns that Scratch programmers follow. In programming, identifiers naming is an important aspect of a program's quality. Better and meaningful names signify easier future comprehension and maintenance of the programs by other programmers. Another aspect that is unique to the case of Scratch and children is that naming patterns could help us understand and resolve challenges Scratch programmers might face when they transition to mainstream textual languages. This is important to follow as Scratch and other block-based languages have language-specific features for naming variables and as well as specific usage patterns.

Many end-user programming barriers can be traced back to misconceptions; understanding and information barriers for example. We, therefore, investigated what programming misconceptions school-age children hold. A programming misconception is having an incorrect understanding of a programming concept or a set of related concepts, typically affected by prior knowledge from domains other than programming such as mathematics and natural languages [208]. There is a lot of research on misconceptions for adult students in universities. However, little is known on children's misconceptions, especially with the considerations regarding children's cognitive development.

To obtain in-depth insight into challenges and barriers, we finally explore how children vocalize textual code and how reading code aloud affects their comprehension. Transitioning from block-based languages, where the focus is on the visual attributes (color and shape for example), to textual languages can be challenging for novice programmers. Taking a look at natural language's education, young children tend to start learning by reading words aloud, repeating after the teacher or on their own. Phonetics of words are taught and reading aloud passages is encouraged. Reading code in programming lessons with children, however, is not practiced as systematic as when learning the natural language, even though for some children classroom lessons can be the first place where they see textual code.

1.5.3 RESEARCH QUESTIONS

In this dissertation we aim at answering the following research questions:

RQ1.A How do spreadsheet users deal with challenges they face when working with spreadsheets?

RQ1.B How can these challenges be addressed?

RQ2.A How do school-age children deal with challenges they face when learning to program?

RQ2.B How can these challenges be addressed?

For the first group, the spreadsheet users, our exploration is a continuation of research efforts in understanding technical challenges, from the end user's point of view, and providing software-based solutions that mitigate these challenges based on Software Engineering principles.

In Chapter 2, we report a case study on a spreadsheet that takes tens of hours to execute. We consequently provide a solution based on parallelizing a spreadsheet workload on multiple computing nodes, thus offering spreadsheet users new ways to do iterative financial modeling in a significantly reduced amount of time.

In Chapter 3, we propose a new technique that enables us to identify, extract and migrate cross-table data from a set of financial spreadsheets into a relational database format.

Our research in performance and data extraction offers insights into how spreadsheet users deal with the challenges of lacking software's environment support and the lack of computational knowledge and capacity required to make effective changes to real-world problems. In addition to highlighting these challenges, our research offers novel ways to address them.

For the second group, the school-age children, we believe that most of these children will become end-user programmers in their professional careers. The schooling phase, thus, becomes of increasing importance since it needs to build the required foundational knowledge in Computational Thinking and software programming for their future. *"The future begins now"* reflects how we should look at the children learning to program. As we identify and investigate possible fixes to the issues young students face right now, we can help them become better end-user programmers in the future.

In Chapter 4, we conduct a study on naming patterns in 250,000 Scratch projects, in order to identify how children name identifiers in Scratch programming. We subsequently compare the found patterns in Scratch to those known in mainstream programming languages.

In Chapter 5, we conduct a field study where we observe how preschoolers learn basic programming concepts through robotics and games. We highlight the nature of challenges they face and link them to Piaget's theory of cognitive development.

In Chapter 6, we perform a study to identify programming misconceptions held by school-age children. We ask programming questions based on the misconception known from research on university-level students. We use the children's answers to indicate whether they hold a certain misconception and to understand where the misconception comes from.

Finally, in Chapter 7 we assess an approach to read code aloud in Python. We provide programming lessons to primary school children aiming to train them on reading code. We subsequently assess the effect of reading code aloud on their code comprehension.

Our research with school-age children offers insights into various aspects related to the challenges that they face when learning to program. We looked at the difficulties of starting early with programming environments and how it related to the still-developing cognitive skills of children. We subsequently showed how children develop naming patterns in Scratch and what misconceptions they could hold about programming in early years. With an eye on the transition to mainstream textual languages we finally investigate how children read textual code. Besides the insights into challenges, our research provides a novel way to overcome the difficulty to read code by adopting a read aloud approach in classrooms.

1.6 METHODOLOGY

The study of end-user programming is a multi-disciplinary topic, requiring a series of research methods taken from the social as well as the technical sciences. The most important methods we used include:

CASE STUDIES

We followed the case study research method when explored the challenges of spreadsheet end-user programmers concerning performance issues in their spreadsheet models. Case studies are often used in the area of Empirical Software Engineering and End-user Engineering [124].

The case study methodology is reported in Chapter 2. In this case study, we analyzed one large-scale spreadsheet model used, investigated the problem of long execution times and proposed a solution in discussion with the end-users. We applied the solution and evaluated it afterward.

We followed a field study research methodology, a special type of case studies, in Chapter 5, where we investigate how preschoolers learn programming. In field studies, observation is the predominant method of data collection [13, 112]. Therefore, we designed and provided programming sessions to these students. We observed the children's behavior during these sessions and reported the observations in the aforementioned chapter.

STATIC CODE ANALYSIS

We used static source code analysis on a repository of Scratch projects to explore how Scratch programmers name variables and procedures. Naming is an important indication of programs understandability by programmers other than the original creator. This study partially replicates another on Java programs and is reported in Chapter 4. In this study, we analyze the dataset created by Aivaloglou and Hermans [4], consisting of 250,000 Scratch projects.

CONTROLLED EXPERIMENT

We study the effect of vocalizing code snippets in classrooms on comprehension in a controlled experimental setup. We provided the same lessons to both student groups, aged between 9 and 13 years, with the only difference being reading the code aloud in the experimental group. This study is presented in Chapter 7.

MIXED-METHODS

Combining quantitative and qualitative research methods is widely known as the mixed-methods approach. We followed this approach in studying the difficulties that lie in identifying and extracting semi-structured data, in the form of a cross-table, from a set of spreadsheets in Chapter 3. In this study, we first applied a static code analysis on the public Enron spreadsheet dataset with the aim of identifying the frequency of using the cross-table data structure. We followed the analysis with a case study in a company in which cross-table data must be extracted from their financial spreadsheets. The study included in-depth interviews with the end-users involved. This study is found in Chapter 3. Another study that uses the mixed-methods approach is the research on programming misconceptions held by school-age students between 7 and 17 years old. We used a questionnaire setup with close and open-ended questions. We designed the questions in Scratch blocks based on a set of programming misconceptions well-known from previous studies on older, university-level students. This study can be found in Chapter 6.

1.6.1 ETHICAL RESEARCH AND DATA COLLECTION

Research impact on humans who take part in experiments and other research activities has driven the development of ethical, social and legal guidelines for researchers to follow [201]. These guidelines concern the protection of dignity, rights and welfare of all human subjects involved in a study. For example, this involves adequately informing the participants about the research goals and impact, acquiring their free consent, guaranteeing privacy and anonymity of the participants and abiding by the general data protection regulations. The TU Delft Human Research Ethics Committee [6] is responsible for setting university-level guidelines that conforms to the general principles in addition to EU regulations. The committee requires that each research involving humans be approved by the committee ahead of performing the research.

In this dissertation we endeavored to adhere to the general ethical principles as well as the university level guidelines. In particular, we highlight the performing of interviews with spreadsheet users in Chapter 3 and the collection of children demographical and programming-assessment data in Chapters 6 and 7 in relation to the following aspects:

Unambiguous and adequately informed Consent: As part of our agreement with the industrial partner, we performed the research on their premises and were in contact with the spreadsheet users directly. We discussed the problem and the solution provided, thus they were aware of the research goal and its impact.

The work with children, as a specially vulnerable group of participants involved taking extra measures. We provided written consent forms to be approved by one of the parents/guardians of the child. In these forms we provided information about the research, its impact and goals. We additionally provided the contact information of the research team if parents needed further details. For the experiment in Chapter 6, children whose parents/guardians did not give consent were not allowed to start the experiment at Nemo[7], the science Museum in Amsterdam, since it was a prerequisite condition according to the museum's guidelines. In the case of the school children in Chapter 7, children who did not provide a signed consent form took part in the lessons, but their data was not collected.

Anonymity: Minimal information was gathered during the experiments. We especially targeted data that relate to programming experience and education. In the research papers we kept the identity of the participants anonymous.

Storing data: We follow the guidelines of TU Delft, which are in accordance with the EU General Data Protection Regulation and the Dutch Code of Conduct for Academic Practice. This means that our collected data, including the consent forms, are stored and archived for at least ten years for the sake of transparency and auditability.

Lastly, the research we performed in Chapter 6 and 7 was approved by the Human Research Committee prior to the start of the experiments.

DATASETS

In this dissertation, we do not generate new datasets. However, we used public and private repositories in our research. For the research on spreadsheet users' challenges, we used the Enron dataset [100] which provides researchers with the opportunity to study a large number of spreadsheets (16,160) that were developed and used in an industrial context. We additionally worked with a smaller set of private spreadsheets from our industrial partner, Solvinity [8].

[6]https://www.tudelft.nl/over-tu-delft/strategie/strategiedocumenten-tu-delft/integriteitsbeleid/human-research-ethics/
[7]https://www.nemosciencemuseum.nl/nl/
[8]https://www.solvinity.com/ (previously known as BitBrains)

For the research on school-age children, we used, in Chapter 4, the public repository created by Aivaloglou and Hermans [4] which includes 250,000 Scratch programs.

1.7 CHAPTERS AND PUBLICATIONS

The chapters of this dissertation are all based on peer-reviewed publications in conferences related to End-user programming, Software Engineering, and Computer Science Education. Each chapter is self-contained with its contributions. You may, therefore, notice some repetitions in the introductions of the chapters. All chapters are co-authored with Felienne Hermans, one chapter is co-authored with Alexander Serebrenik, and another is co-authored with Rubin Koesoemowidjojo. The dissertation consists of the following chapters:

Chapter 2: Spreadsheet performance Based on our paper *"Improving the Performance of a Large Scale Spreadsheet: A Case Study"* (Alaaeddin Swidan, Felienne Hermans and Ruben Koesoemowidjojo) which appears in the proceedings of the IEEE Software Analysis, Evolution, and Reengineering conference, SANER 2016.

Chapter 3: Spreadsheet data extraction Based on our paper *"Semi-automatic extraction of cross-table data from a set of spreadsheets"* (Alaaeddin Swidan and Felienne Hermans) which appears in the proceeding of the The International Symposium on End-User Development, IS-EUD 2017.

Chapter 4: Identifiers naming practices in Scratch Based on our paper *"How do Scratch Programmers Name Variables and Procedures?"* (Alaaeddin Swidan, Alexander Serebrenik and Felienne Hermans) which appears in the proceedings of IEEE 17th International Working Conference on Source Code Analysis and Manipulation, SCAM 2017.

Chapter 5: Programming education to preschoolers Based on our paper *"Programming Education to Preschoolers: Reflections and Observations from a Field Study"* (Alaaeddin Swidan and Felienne Hermans) which appears in the 28th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2017.

Chapter 6: Programming misconceptions for school students Based on our paper *"Programming Misconceptions for School Students"* (Alaaeddin Swidan, Felienne Hermans and Marileen Smit) which appears in the proceedings of the 2018 ACM Conference on International Computing Education Research, ICER 2018.

Chapter 7: Effects of vocalizing code on comprehension Based on our paper *"The Effects of Reading Code Aloud on Comprehension: An Empirical Study"* (Alaaeddin Swidan and Felienne Hermans) which appears in the proceedings of the 1st Global conference on Computer Education, CompEd 2019.

Chapter 8: Conclusions In this chapter, we summarize the contributions of this dissertation and revisit the research questions.

Additional Publications

The author has been involved in other related publications which are not directly included in this dissertation:

- Spreadsheets are code: An overview of software engineering approaches applied to spreadsheets (Felienne Hermans, Bas Jansen, Sohon Roy, Efthimia Aivaloglou, Alaaeddin Swidan, David Hoepelman) which appears in the proceedings of the IEEE Software Analysis, Evolution, and Reengineering conference, SANER 2016.
- Thinking out of the box: comparing metaphors for variables in programming education (Felienne Hermans, Alaaeddin Swidan, Efthimia Aivaloglou, Marileen Smit) which appears in the proceedings of the 13th Workshop in Primary and Secondary Computing Education, WiPSCE '18.
- Code Phonology: an exploration into the vocalization of code (Felienne Hermans, Alaaeddin Swidan, Efthimia Aivaloglou) which appears in the proceedings of the 26th Conference on Program Comprehension, ICPC '18.

Open Science

Open science is the idea that *“scientific knowledge of all kinds should be openly shared as early as is practical in the discovery process”*[9]. This involves six main principles of openly sharing: methodology, source, data, access, peer review and educational resources.

Open science is increasingly looked at as an essential part of successful research, and an accelerator to the process of discovery [237]. Therefore, funding agencies on the national (Netherlands Organization for Scientific Research [160]) and European levels[10], as well as many universities are demanding that all publications resulting from their projects or employees are available in open access.

Our contribution to the open science initiative is through providing open access to the research output in this dissertation. Open access is one principle among the open science initiative and focuses on *“publishing in an open manner and making it accessible and accessible to everyone”*[11]. This includes in addition to the publications, the access to the materials and assessment questions used in experiments especially in Chapter 6 and Chapter 7. All of the papers are thus accessible via the research repository of TU Delft[12].

[9]http://opensciencesap.org
[10]https://ec.europa.eu/research/openscience
[11]https://www.budapestopenaccessinitiative.org/read
[12]https://pure.tudelft.nl

2 Spreadsheet Performance

Spreadsheets are used extensively for calculations in several domains, especially in finance and insurance. Spreadsheets offer a clear benefit to their users: they are an easy to learn application in which to express their business needs, however, there are downsides too. Like software, spreadsheets can have a long life span in which they are used by several people. This leads to maintainability issues, including errors, but also often to issues with performance. In this chapter, we present a case study in which a model for shortfall calculations, *originally implemented in a spreadsheet, was adapted to run on an High Performance Cluster (HPC). We present the design, analysis and implementation of the solution which clearly improved the performance of the spreadsheet, with a factor of 50 in some cases. We subsequently reflect on challenges related to reverse engineering, testing and scalability. Finally, we identify opportunities that would provide automatic support to refactoring, dependency recognition and performance profiling in future spreadsheet optimization projects*[1].

[1]This chapter is based on our paper: Improving the Performance of a Large Scale Spreadsheet: A Case Study (Alaaeddin Swidan, Felienne Hermans, and Ruben Koesoemowidjojo) published in the proceedings of the IEEE Software Analysis, Evolution, and Reengineering conference, SANER 2016

2.1 INTRODUCTION

Research shows the prevalent use of spreadsheets in various business domains in general, and financial services in particular [18, 62, 95]. Spreadsheet usage ranges from trivial data manipulation to complex modeling and simulation [62]. A spreadsheet model in principle is easy to create: domain experts convey their ideas into the intuitive UI, allowing for immediate results to show, and thus providing the ability to react with a business decision such as buying or selling certain stock. Within organizations, spreadsheets are used for long periods of time [95], which could lead to a spreadsheet model growing in size and complexity. Increased size and complexity could affect the performance of the spreadsheet, hampering immediate output and thus inhibiting one of the strongest features of spreadsheets [95].

To investigate the issues around spreadsheet performance in practice, we present a case study of one large-scale spreadsheet model which suffered from performance problems. The spreadsheet was developed at a major insurance company in the Netherlands. It was used to support risk analysts in comparing pension investments and obligations. The spreadsheet suffered from severe performance issues, as the execution regularly exceeded 10 hours. This situation was inconvenient to the business operations, as the time-consuming runs of the spreadsheet caused additional difficulties in its maintenance and usability for the analysts. To address these problems, a parallel-based solution was applied to the spreadsheet, based on a Microsoft High Performance Cluster (HPC). In this chapter, we highlight the steps followed to improve the performance of the spreadsheet.

The contributions of this chapter are: (i) providing an industrial investigation into addressing performance issues related to simulation models in spreadsheets, (ii) identifying the challenges that we faced throughout the project, and how they were overcome. Finally, (iii) we introduce possible enhancements, which we foresee could mitigate the risks in similar projects.

2.2 BACKGROUND AND MOTIVATION

Simulation models in general can be either deterministic or stochastic. In deterministic models, input values are set before the run, and calculations are built to produce an output accordingly. Running a deterministic model for many times will produce the same result in each run. Opposite to that is the stochastic modeling, where the knowledge of the inputs is neither complete nor certain. In a stochastic model, some of the inputs are chosen randomly, and usually independently, from within a range of distributed values. This is a common modeling approach used in the financial service-providers, banks and insurance companies that highly depend on live information of stocks. In each run of the model, a different set of inputs is used, leading to unique and discrete results. One of the most popular stochastic modeling techniques for numerical analysis is the Monte Carlo simulation [236]. As illustrated in Figure 2.1, Monte Carlo simulations aim to evaluate a specific calculation, a function, depending on a group of input parameters that are discrete, independent and randomly evaluated from a range of possible values. In Monte Carlo, the main calculation function is executed for a predefined number of iterations $\boldsymbol{n}$. Each iteration's result is recorded, and participates in presenting the final outcome to the user in the shape of graphs, aggregations and probability analysis.

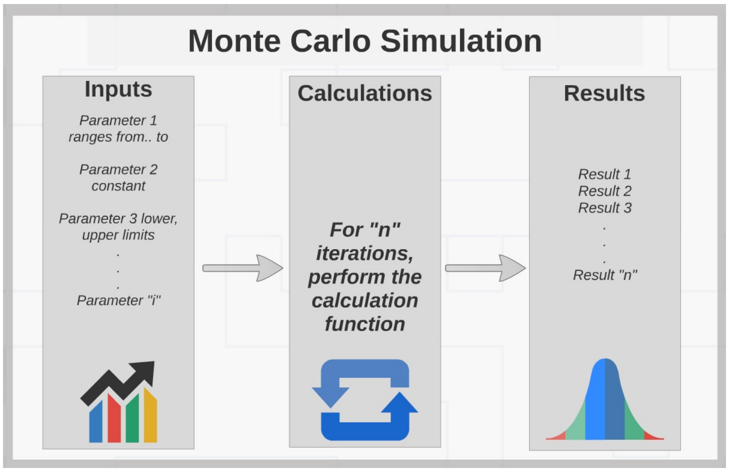


Figure 2.1. Monte Carlo Simulation Summary

2.2.1 MOTIVATION

Monte Carlo simulations by nature suffer from performance issues. This is usually the case since analysts and domain experts prefer to run their simulations for as many iterations as possible. By this, they aim to produce more precise and reliable results. However, this leads to a high utilization of computing resources, and thus a growth in the runtime of the model relative to the number of iterations. The overall health of the model is also affected, because its maintenance becomes an issue. In our case for instance, modifying the model meant that a simple debugging, of any change, would take tens of hours to finalize.

Table 2.1. VBA Code Metrics Extracted from the Spreadsheet Model

VBA Code Analysis		
Size	Number of modules	23
	Number of methods	146
	Number of types	58
	Lines of code (total)	7,372
	Comments coverage	12.79% (1081 LoC)
Complexity	Cyclomatic Complexity for Methods	Max=386, Average=14.36
	Cyclomatic Complexity for Types	Max=546, Average=69.3
	Methods too complex (Cyclomatic Complexity >20)	17 (11.6% coverage)
	Methods too big (LoC >= 30)	53 (36.3% coverage)
	Types with too many fields (fields>20 and not enumerated)	5
External Dependency	Number of projects invoked	2
	Number of methods invoked	65
	Number of types used	14

2.3 CONTEXT

The company at which the spreadsheet was developed, is one of the largest financial service providers in the Netherlands, with millions of customers and thousands of employees. In the next subsections we provide information about the use case and aspects of the spreadsheet model. From now on, the company will be referred to as the insurance company.

2.3.1 SHORTFALL ANALYSIS

The investigated spreadsheet model is called the shortfall risk calculator (SFC), and contains an application of a Monte Carlo simulation. A shortfall analysis aims to numerically predict the deficit of the pension liabilities of a company, compared to their stock market capitalization [161]. This is implemented through a stochastic evaluation of the expected returns of the company's investments in equity markets for example, while simultaneously comparing it to the probabilities of the pension obligations at a certain point in the future. An SFC model usually involves a high level of uncertainty and deals with continuously changing data, such as the interest rate, stock compounded market rate, and mortality intensity of a person in a certain age [161].

In our case, the SFC model was implemented using Excel, and used to run it on the common enterprise PCs. The problem is that the model usually took many hours to complete, sometimes days. Worse case, the model would crash unexpectedly and the analysts had to run it again. The performance of the model caused a major problem to the risk analysis department by delaying the decision making process. Moreover, this made it difficult to maintain the spreadsheet model. For example to add a new functionality, debug or test any modification, meant that a new run will start, which then will take another tens of hours to complete.

2.3.2 FEATURES OF THE MODEL

In this section we present the metrics of the original spreadsheet model and the contained VBA code. For that, we used our research spreadsheet analyzer [100], and a leading software tool for VBA code quality [228]. These metrics are shown here to illustrate the model features, but were not used by the developers in the case.

The SFC model in our case features one core spreadsheet which is the shortfall calculator. The core spreadsheet depends on three external spreadsheets for providing input data and configurations. It produces data into three other spreadsheets. The core spreadsheet contains 12 worksheets, and a total of 3,329 nonempty cells. Within these cells, 329 contain formulas of which 72 are unique. The formulas used have a low level of complexity, which is indicated by the number of cells (16) containing functions such as VLOOKUP, OFFSET, and IF.

The VBA project in the core spreadsheet has 37 source files with more than 7,000 LoC. The code contains 23 modules, 146 methods, and 58 user-defined types. We generated the standard complexity metrics for methods and types, finding the maximum Cyclometic Complexity (CC) of 386 paths for the `main()` method. Another 16 methods are labeled as too complex by the software tool we used, since they have CC value above 20 paths.

2.4 THE CASE STUDY

As an HPC solution provider, our industry partner BitBrains[2] [27] was approached by the insurance company to improve the spreadsheet's performance. In collaboration with GridDynamics [90], the solution we provided is based on a framework called the HPC for Excel Acceleration Toolkit (HEAT). HEAT, developed by GridDynamics, is built on top of a standard Microsoft HPC cluster [219], and allows for the spreadsheet calculations to be offloaded into the HPC nodes (Figure 2.2). The project was carried out in three phases that included: setting up the design based on parallelism (Section 2.4.1)), followed by analyzing the spreadsheet model to decompose its components (Section 2.4.2), and implementing needed changes to the original model (Section 2.4.3), leading to clear improvements in the model performance as presented in the evaluation (Section 2.4.4).

2.4.1 DESIGN

In determining the design, the aim was to handle two major requirements: (i) the parallelization of the model, and (ii) the scalability of the provided solution.

The parallelization of the model was achieved through a map-reduce derived algorithm. However, a limitation in Excel prevented user-defined types from being communicated between HPC nodes. In VBA, a user-defined type is a data structure that is defined by the keyword `Type`, and contains a combination of built-in types. Since the model contained more than 50 user-defined types (Table 2.1), the solution was to serialize them into built-in types before sending to the cluster nodes, and to deserialize them back to the original types on receiving.

The second requirement considered in the design was the scalability. The insurance company's aim was to evaluate as large as 30K scenarios in the future, and this would bring up two issues: a high memory utilization, and an increased time in splitting the input data. Consequently, the original model was redesigned, and that included some code refactoring, such as minimizing the creation of temporary arrays inside the loops.

[2]Now known as Solvinity (https://www.solvinity.com/nl), with whom we also collaborate in Chapter 3

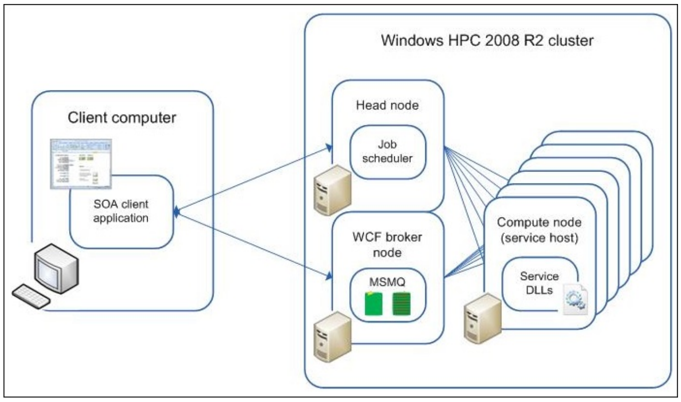


Figure 2.2. Microsoft HPC Standard Package [219]

2.4.2 Analysis

In the analysis phase, the spreadsheet model was analyzed to determine what methods to parallelize and how. The analysis consisted of three operations:

Performance profiling The profiling was done by instrumentation, where a time-logging library was called on each VBA method's entrance and exit. Three methods were the most time-consuming, one of them was the `main()` method.

Complexity evaluation Through standard tools, the complexity and size of the VBA project were measured. The `main()` method was the largest and most complex.

Dependency analysis This was done by studying the input and output data flows of the methods, with the aim of avoiding parallelizing a method with high level of data coupling. The original VBA code was written in a way that introduces a considerable amount of data coupling between the methods, where the `main()` method exhibited the most dependency on other methods.

As a direct outcome of the analysis, various loops in a number of VBA methods were candidates for parallelization. Of which, the `main()` method was the most feasible since it contained the most time-consuming code. Important to note here, the analysis process was difficult, since the software engineers lacked the needed domain knowledge of the model. As such, information were regularly required from the model owners at the insurance company (discussed further in the Challenges, Section 2.6.1).

2.4.3 Implementation

The implementation comprised rewriting parts of the VBA code, and adding other parts (i) to apply parallelization and (ii) to deliver a scalable solution.

The parallelization targeted various loops in the `main` method, and the focus was on splitting the arrays used in these loops' iterations. The mapping for an array was done by dividing the total number of scenarios by the number of cluster cores, consequently creating split copies of that arrays. On the other hand, the reduce process was performed based on each function's requirements: some arrays were averaged or summed, others were concatenated. To ensure successful parallelization, dependencies between the methods were minimized. In this manner we refactored, when applicable, code parts that include passing parameters by reference, or modifying the passed parameters inside a function's body.

Because of the parallel approach, serialization and deserialization code was developed to allow the communication of each user-defined type in the model. This is due to a limitation in Excel, which forbade the proper transfer of user-defined types between the cluster nodes (described in Section 2.4.1).

Finally, to ensure scalability both in terms of the number of scenarios and the number of cores, re-design of the model in some loops was implemented. For instance the creation of temporary arrays inside the loops was not allowed. In addition, the array splitting and copying were moved to the cluster instead of the client for some loops, and some data structures were made smaller through using data types with smaller sizes.

2.4.4 Evaluation

As a result of the parallelization approach and the modifications to the model, the run times showed clear improvements. In this section we present (i) the performance figures of the various runs of the model, (ii) the verification of the model's health throughout the migration process.

Performance Results There were two factors that directly affected the performance: the number of scenarios in the model and the number of cores utilized. First, Figure 2.3 shows the effect of increasing the scenarios on the model's performance, while the number of cores was fixed: a single core for the sequential original model, versus a 24-core cluster for the parallel modified model. When the original model was considered, the runtime increased exponentially as we increased the number of scenarios. In the meanwhile, the modified model runtime increased as well, but in an almost linear manner. In Figure 2.3-b, we see that the modified model in parallel performed 39 and 52 times faster than the original model, for 480 and 960 scenarios respectively. To run 960 scenarios for instance, the original model required 729 minutes (more than 12 hours), while the modified model completed the run in 14 minutes on the 24-core cluster. Figure 2.4 illustrates the effect of increasing the number of cores on performance, while the number of scenarios is fixed. For 1008 scenarios, the original model needed more than 15 hours to complete. For the parallel runs, started by 8-cores and increased gradually up-to 48 cores, the runtime goes down almost linearly, from 105 minutes to 33 minutes respectively.

Model Accuracy The accuracy of the obtained results was verified throughout the migration of the model. The verifications were carried out by the insurance company's team, who manually compared the outcomes of the calculations between the original and the modified models. In one case for example, the results were slightly different, and an investigation showed that a conditional statement was not migrated correctly. However, the outcome of the modified model was more accurate than before, due to running more simulations, which is expected in a Monte Carlo simulation.

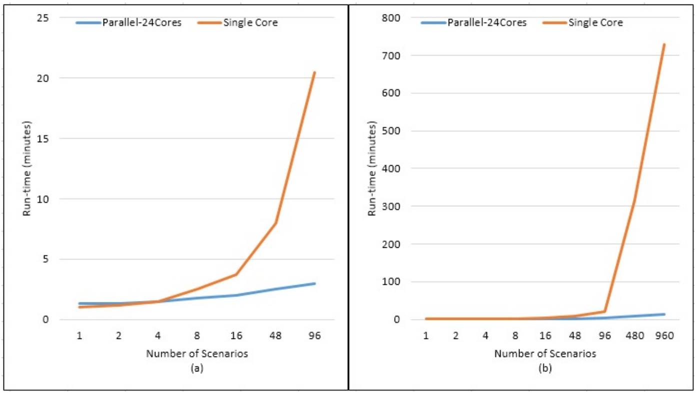


Figure 2.3. Results: Effect of increasing the number of scenarios on the performance of the original model and the modified model. (a) On a small scale: up to 96 scenarios. (b) On a larger scale: up to 960 scenarios

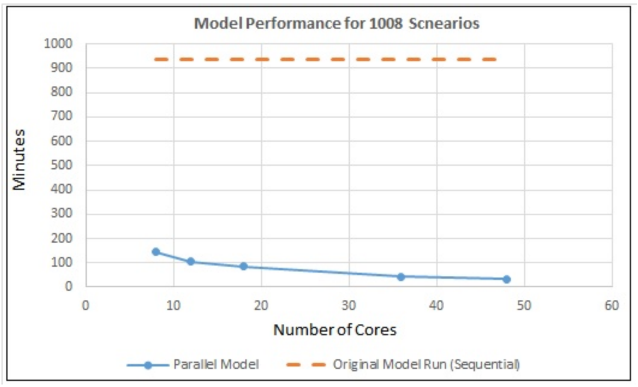


Figure 2.4. Results: Effect of increasing the number of cores on the performance of the parallel model

Migration Cost According to the project plan sheet, the migration was finalized in 431 working hours. The recorded work included development of the new model and changes to the HEAT, testing and verification, bug fixing, and writing documentation. 93 hours of the recorded hours were related to providing support for the insurance company's employees, helping them through the running of the model.

2.4.5 Maintainability of the Modified Model

The insurance company wanted to ensure business continuity. They required that the modified model is easily changed and maintained by their end-users, without continuously referring to the engineers in BitBrains. To achieve this, the HEAT provides an option for the end-users to run the model sequentially on a local machine. This is important to test the introduced changes, before the model is run on the cluster. Furthermore, the engineering team performed several hand-over sessions, and provided thorough documentation materials, that detailed how to modify the core parallel part of the code.

2.5 Related Work

Related to our study is the work of Pichitlamken *et al.* [174], where they provided a prototype that is similar to the one we presented. The prototype, presented as an Excel add-in, offloads the calculations of a spreadsheet model to a computer grid. However, the two solutions still differ. The solution in [174] built its own resource management and scheduling unit, based on FCFS algorithm. In our solution, HEAT exploits the well-established Microsoft HPC services for these functionalities. Additionally, the solution they provided as a prototype, was not validated on a large-scale spreadsheet, or in an industrial setup. In another related work, Abramson *et al.* in [2] presented ActiveSheets, a solution designed to evaluate the parallel user-defined functions of a spreadsheet in the background. ActiveSheets is different to our solution by targeting independent user-defined functions, and the lack of an industrial application. Another related approach is ExcelGrid [156], in which a middle ware tool was designed and built to connect an Excel spreadsheet with either a private or a public cluster. Even though ExcelGrid applies a parallel solution to distribute work on a cluster, it does not target spreadsheet models built completely inside Excel, as it uses the spreadsheet as a means for providing input data to another software. To the best of our knowledge, our study presents the first industrial and large-scale case, involving the parallelization of a spreadsheet model.

2.6 Lessons Learned

In the above, we described the steps used by BitBrains and GridDynamics, to improve the performance of a spreadsheet-based model. This was done by creating a design based on a software framework, that allows Excel spreadsheets to run in parallel on an HPC cluster. Subsequently, the model was analyzed by the software developers to identify candidate methods for parallelization. Following that, the solution was implemented by modifying parts of the the methods, and adding other code parts to ensure a successful integration between the spreadsheet and the HPC cluster. In this section, we reflect on the project, identifying challenges that arose along the way to the final implementation. Subsequently, we introduce possible opportunities that can be addressed in future work, to minimize the efforts in similar spreadsheets projects.

2.6.1 CHALLENGES

Throughout the phases of the solution, a number of issues proved to be challenging. We foresee that these challenges will also be faced in future projects that optimize other spreadsheet models. Therefore, the followings are our insights into some of the challenging issues.

Reverse Engineering and Restructuring The analysis of the spreadsheet model was a difficult process. Four weeks were needed to reverse engineer the data flows and the algorithm of the original spreadsheet model. The difficulty originated from the fact that the developers lacked financial domain knowledge, which required holding several meetings with the model owners. These meetings hampered a speedy development process. On the other hand, the implementation phase involved applying several refactorings to the VBA code, as well as rewriting and adding some components to the HEAT framework, for the sake of fitting all the parts together. These recursive and continuous processes of reverse engineering, restructuring and integration development were vital to prepare the spreadsheet model for the parallel execution, without any unwanted result. These preparations, however, were done manually, neither planned nor systematic, which was time consuming.

Preserving the Model's Functionality The transformation process is risky; each modification applied to the original model compromises the accuracy of the output results, thus forcing continuous validations throughout the process. These validations, in our case, were performed manually and remotely by the insurance company's team. With the lack of domain knowledge added, It was difficult for the developers in some cases to track the root causes of the variances, causing some delays to the implementation process.

Scale of Improvement When to consider the performance improvement *enough* in a similar project? This question was not answered in our case. The computational power was increased gradually to see the reflection on performance. However, we believe that a turnover point exists, at which the model's run may take longer time. This would happen since the increase of the computation nodes means the increase of the number of intermediate results to be reduced. Depending on the size of the data structures used, the intermediate data may grow rapidly to utilize more memory, increasing the processing time. Therefore, the computing resources that would achieve the optimal performance is not previously defined, and is challenging to measure.

2.6.2 OPPORTUNITIES

In this section we present areas for researchers to investigate in future. The aim is to provide tools and solutions to issues that could hinder the implementation of similar spreadsheets projects. We identify the following opportunities, categorized per phase of the solution:

DESIGN

The solution was primarily based on parallelization of the spreadsheet model, by offloading the core computational parts to the underlying cluster. An improvement to the design is possible by accompanying the core parallel solution with a concurrent approach. For example, we envision a new design that includes a step related to detect and refactor spreadsheet smells that may cause performance issues, such as long calculation chains [103].

Analysis

In analyzing a spreadsheet model, software engineers could benefit from having tools in the following areas:

Dependency Recognition To analyze the dependency between the VBA methods, the developers carried manual checks to the data flows and algorithms of the model. A systematic approach, through a tool for example, is needed to identify dependencies inside a spreadsheet model. This would allow the developers and spreadsheet users to decide possible parallel sections in their spreadsheet easily, and more quickly.

Performance Profiling and Hot-spot Analysis In the same manner to the dependency checks, profiling of the VBA methods was done manually by writing code to log the run times, and upon that deciding the major functions. A spreadsheet model consists of different parts (both formulas and VBA code), that can consume time and resources. The manual process is error-prone and time consuming, thus a tool that detects code runtime , and links it directly to the associated formula or VBA method, would produce more assured results.

Implementation

The development cycles in the solution involve repetitive work that has room for improvement, in the form of automation or semi-automation of some processes. The developers identified a predefined set of code behaviors that lead to heavy dependencies between functions, such as the passing parameters by reference, and the modification of parameters inside the function body. A tool can be developed to detect and refactor the code to eventually remove, or reduce, the dependencies.

Finally, the solution presented is fully customized towards a specific spreadsheet model. This suggests that for each new spreadsheet model (new in terms of the simulation type, or the used data structures and algorithms) the same procedure shall be followed again. An opportunity to improve would be widening the targeted models, by specializing in a specific category of financial simulations, such as Monte Carlo. The solution would automatically analyze any model that fits the simulation criteria, based on a predefined set of rules and steps, reducing the amount of repetitive work in future projects.

3 Spreadsheet Data Extraction

Spreadsheets are widely used in companies. End-users often value the high degree of flexibility and freedom spreadsheets provide. However, these features lead to the development of a variety of data forms inside spreadsheets. A cross-table is one of these forms of data. A cross-table is defined as a rectangular form of data, which expresses the relations between a set of objects and a set of attributes. Cross-tables are common in spreadsheets: our exploratory analysis found that more than 3.42% of spreadsheets in an industrial open dataset include at least one cross-table. However, current software tools provide no support to analyze data in cross-tables. To address this, we present a semi-automatic approach to extract cross-table data from a set of spreadsheets, and transform them to a relational table form. We evaluate our approach in a case study, on a set of 333 spreadsheets with 2,801 worksheets. The results show that the approach is successful in extracting over 92% of the data inside the targeted cross-tables. Further, we interview two users of the spreadsheets working in the company; they confirmed the approach is beneficial and provides correct results.[1]

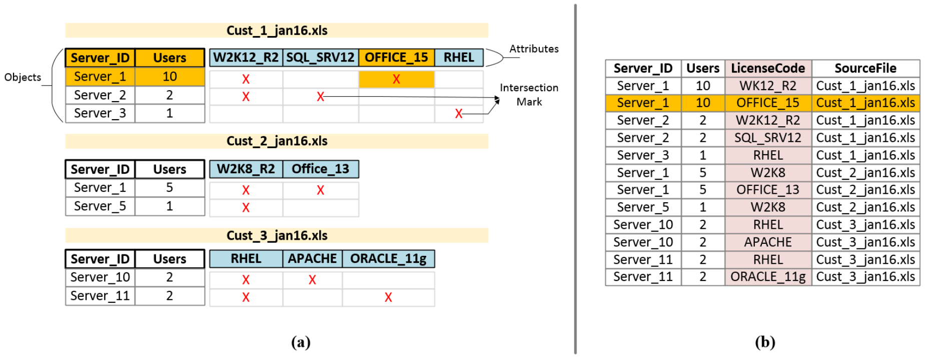
[1]This chapter is based on our paper: Semi-automatic Extraction of Cross-table Data from a Set of Spreadsheets (Alaaeddin Swidan and Felienne Hermans) published in the proceeding of the The International Symposium on End-User Development, IS-EUD 2017

3.1 INTRODUCTION

Spreadsheets are used extensively across various domains of expertise, to perform a wide range of tasks[48, 55, 56, 61]. In particular, spreadsheets are often used for data analysis and management [49]. In addition to the powerful set of functionalities, the end-users appreciate the flexibility and freedom provided by spreadsheets [95]. This leads, however, to a variety of forms in which data is represented inside a worksheet. One of the special forms of data found in spreadsheets is the *cross-table*. A cross-table in general aims to represent a relation I. It consists of G rows and M columns, and can be defined as:

A rectangular table with one row for each object and one column for each attribute, having a cross in the intersection of row g with column m iff (g,m) belongs to I. [26]

Cross-tables, an example of which is shown in Figure 3.1, are common in spreadsheets. To explore this, we manually investigated more than 1,500 spreadsheets from the industrial dataset Enron [100]. Our investigation revealed that 552 spreadsheets, or 3.42% of the dataset, include at least one cross-table (Section 3.3). As an example of a cross-table in a



Cust_1_jan16.xls

Server_ID	Users	W2K12_R2	SQL_SRV12	OFFICE_15	RHEL
Server_1	10	X		X	
Server_2	2	X	X		
Server_3	1				X

Cust_2_jan16.xls

Server_ID	Users	W2K8_R2	Office_13
Server_1	5	X	X
Server_5	1	X	

Cust_3_jan16.xls

Server_ID	Users	RHEL	APACHE	ORACLE_11g
Server_10	2	X	X	
Server_11	2	X		X

(a)

Server_ID	Users	LicenseCode	SourceFile
Server_1	10	WK12_R2	Cust_1_jan16.xls
Server_1	10	OFFICE_15	Cust_1_jan16.xls
Server_2	2	W2K12_R2	Cust_1_jan16.xls
Server_2	2	SQL_SRV12	Cust_1_jan16.xls
Server_3	1	RHEL	Cust_1_jan16.xls
Server_1	5	W2K8	Cust_2_jan16.xls
Server_1	5	OFFICE_13	Cust_2_jan16.xls
Server_5	1	W2K8	Cust_2_jan16.xls
Server_10	2	RHEL	Cust_3_jan16.xls
Server_10	2	APACHE	Cust_3_jan16.xls
Server_11	2	RHEL	Cust_3_jan16.xls
Server_11	2	ORACLE_11g	Cust_3_jan16.xls

(b)

Figure 3.1. An example showing related data presented in three separate spreadsheets using cross-tables in (a). (b) shows the equivalent data migrated into one relational table.

context, consider Figure 3.1a. The example is based on our case study at Solvinity[2], which provides virtual environments and IT services. In the example, the finance department uses spreadsheets to record the data related to third-party licenses sold as part of each virtual environment, in a cross-table. In one scenario, an internal auditor, let us call him Bas, aims to answer management questions such as: do actual billed licenses conform to the specification of the providing third-party? do the actual-billing data on licenses conform to the actual software license-configuration (eg. number of users) on the servers? To answer these, and similar related questions, data analysis on the related cross-tables, which are included in separate files should be carried out.

As a human, Bas can easily read a cross-table. For example, in Figure 3.1a, he can read that for *"Cust_1_jan16.xls"*, *"Licence: OFFICE_15"* is related to *"Server_ID:Server_1, Users:10"*. Bas does this *mapping* through a sequence of mental operations, which include

[2]https://www.solvinity.com/

linking and relating the two-separate data sets: the objects and the attributes. In addition, Bas incorporates his domain knowledge to recognize that the attribute *“OFFICE_15”* is a data value of type *“License”*.

In reality, Bas is *transforming* each X-marked relation in the cross-table, into one *“record”* in the relational table in Figure 3.1b. Therefore, if the data was originally formed in a relational table form, the user would be spared from performing the conversion steps, and focuses on the analysis part of the task. In addition, data in a relational table allows the user to leverage a wide-set of analysis tools and visualizations, including Excel itself. These tools are designed to work best with strictly-formed relational tables. In our case, the user can perform the transformation from a cross-table to a relational-table form manually. However, as the number of cross-tables increases, the task becomes time-consuming, tedious and unrealistic. The previous scenario describes a twofold problem for the end-users:

a. The data has a special form that is not supported by current software tools.

b. The data should be consolidated from multiple spreadsheets into one relational table, to be used in other analysis software.

To address this problem, we propose a semi-automatic approach that extracts cross-table data from a set of spreadsheets, and converts them into one (denormalized) relational table. Our approach expands upon previous studies that managed to extract data and hierarchy of relational tables in spreadsheets [102]. In addition to the user input, we incorporate an automatic transformation algorithm that is suitable for a cross-table. Our approach aggregates the data from all the transformed cross-tables, and generates one relational table. The final output is the structure and data of the desired table, encoded in an SQL script.

We perform a mixed method evaluation of our approach on a set of 333 spreadsheets with 2,801 worksheets found in the company dataset. First, we quantitatively analyze the approach performance in extracting cross-table data. Subsequently, we interview two frequent users of the spreadsheets, to manually validate the approach generated data on a subset of 30 spreadsheets.

The contributions of this chapter are:

1. An exploratory study on the incidence of cross-table in spreadsheets. For this, we manually examined a subset of more than 1,500 Enron spreadsheets.

2. A semi-automatic approach that identifies, transforms cross-table data from multiple spreadsheets, aggregating them into one relational table.

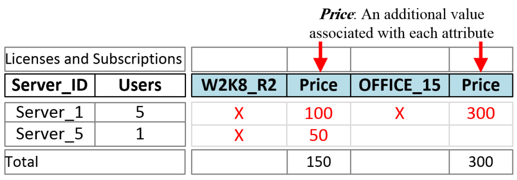
3. An industrial evaluation through a case study.

3.2 BACKGROUND

Before describing the extraction approach, we provide a brief overview of the preliminaries this chapter builds upon.

3.2.1 CROSS-TABLES

There are two types of cross-table: single-valued and many-valued [26]. A single-valued cross-table, shown in Figure 3.1a, follows the general definition, where a binary relation


Price: An additional value associated with each attribute

Licenses and Subscriptions					
Server_ID	**Users**	**W2K8_R2**	**Price**	**OFFICE_15**	**Price**
Server_1	5	X	100	X	300
Server_5	1	X	50		
Total			150		300

Figure 3.2. An example of a many-valued cross-table. The column *Price* represents an additional data value associated with each attribute in the adjacent column.

I between a row-based object g, and a column-based attribute m is marked using the *intersection cell*. A many-valued cross-table, shown in Figure3.2, adds a fourth dimension to the definition, which is the set of data values W_m that are associated with each attribute m.

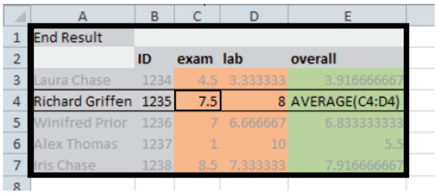
Cross-tables are sometimes called *"matrices"* [23, 192]. We find that *"cross-table"* is the frequently used term [21, 26, 221], so we use it throughout this chapter. A cross-table, as a word, suggests the usage of a *"cross"*, or the *"X"* character in *the intersection cell* which indicates the relationship between an object and its attributes. In practice, however, users are not restricted to use the cross, and they can use other marks in the intersection cell, which can be a specific character, a digit or a shape such as a circle or a triangle.

In spreadsheets, cross-tables may be developed for various tasks. For a simple example, a spreadsheet with a cross-table can be used by employees (objects) to reserve meeting rooms (attributes). A more complex example is used in *"software requirement traceability"* [229],where the user follows up the progress of customer requirements (objects) against system components under development (attributes). In Section 3.3, we quantify the popularity of using cross-tables in spreadsheets in a business context.

3.2.2 EXTRACTION AND TRANSFORMATION OF SPREADSHEET DATA

In this subsection, we present an overview of previous research work, and software products which targeted the extraction of data from spreadsheets for various goals. One example from previous research is the GyroSAT algorithm, which was developed by Hermans *et al.* [102] to help end-users comprehend the structure of spreadsheets. For that purpose, the GyroSAT visualized the hierarchy and relations between blocks of cells, and worksheets, in a spreadsheet. Another research work is UCheck [1], which extracts header and unit information from all cells, as a means to detecting potential errors in spreadsheets. In software, database and data analysis systems, such as Microsoft SQL Server, consider spreadsheets as a data source, and the user is allowed to import data from a spreadsheet. The aforementioned tools and software products, however, work best with relational tables in spreadsheets. For cross-table data, the user should transform the data inside the spreadsheet into a relational table before being able to use these software tools.

Despite being designed to work on relational tables, some components from previous research can be re-used to extract other forms of data, specifically the cross-table. In our case, our approach expands upon the GyroSAT algorithm, which we choose because: First, the GyroSAT is able to identify a rectangle of adjacent cells, which is called a data block. A cross-table, in essence, is a rectangular area of cells, which makes the identification of a data block is the logical first step in extracting data from a cross-table. Second, a recent



	A	B	C	D	E
1	End Result				
2		ID	exam	lab	overall
3	Laura Chase	1234	4.5	3.333333	3.916666667
4	Richard Griffen	1235	7.5	8	AVERAGE(C4:D4)
5	Winifred Prior	1236	7	6.666667	6.833333333
6	Alex Thomas	1237	1	10	5.5
7	Iris Chase	1238	8.5	7.333333	7.916666667
8					

Figure 3.3. A Data Block Including the Cells A1:E7 [95]

study benchmarked the GyroSAT and UCheck performances against the selection made by human users of spreadsheets [190]. The study reveals that GyroSAT has better performance in identifying cell types, which is a prerequisite for the data block identification. Following is the description of the two components that are implemented in GyroSAT, and are used in our approach.

- **Cell Classification**: The GyroSAT approach has four classifications of a cell depending on its content and relations. A *"formula"* type is given to a cell which contains a formula inside. The cells which are referred by the formulas are considered *"data"*. If the cell has an empty content, it is given the *"empty"* type, otherwise the cell is given the *"label"* type. Label cells contain data that are not part of any calculation, thus it is supposed they label other data.
- **Data Blocks:** Using the cell classification, and through a cell-to-cell search algorithm, the approach identifies a rectangular area which includes physically-adjacent cells. The data block, as shown in Figure 3.3 is not expanded when each of the current four corners is surrounded by empty cells, from the outside of the block, in the diagonal, vertical and horizontal directions. In Figure 3.3 for example, the corner cell A7 is not expanded since the outside neighbor cells A8 and B8 are both empty.

3.2.3 MOTIVATION

As the spreadsheet usage grows in organizations, the data in spreadsheets becomes more important for decision making processes [56]. When valuable data are stored in spreadsheets, users such as executives, managers or auditors, become more interested in acquiring knowledge out of the spreadsheets. Many software tools might aid these users in understanding their business, and taking actions accordingly. For instance, Business Intelligence (BI) tools, such as Tableau[3], provide reports and visuals. Traditional database systems provide the ability to perform user-specified queries. In addition, data integration tools, such as Microsoft SSIS[4], allow to compare and aggregate data from different sources.

Despite considering spreadsheets as a potential data source, these software tools require the data inside these spreadsheets to be in a strict relational table form [49]. If this is the case, the user can directly leverage the powerful analysis features provided by these tools. However, when the data is in other forms, such as a cross-table, the users are expected to transform the data beforehand. The transformation can be done manually, though it is tedious and time-consuming especially when the data involved is large in scale. In fact,

[3]https://www.tableau.com/
[4]https://msdn.microsoft.com/en-us/library/ms141026.aspx

when working on spreadsheet data, a recent survey reveals that users prefer less manual processing and more automatic solutions [56].

In this chapter, we aim at providing an approach that extracts and transforms data formed in cross-tables, from multiple spreadsheets, into one integrated relational table, with minimum user efforts. The output of our approach, encoded in standard SQL, can be used to further analyze the data by many analysis tools, not limited to the ones described earlier.

3.3 AN EXPLORATORY ANALYSIS OF CROSS-TABLES IN INDUSTRY

Before we address the extraction of cross-table data, it is important to realize how widespread cross-tables are in spreadsheets. To achieve this, we analyze the Enron spreadsheet dataset in search for the usage of cross-tables. The Enron dataset [100] provides researchers with the opportunity to study a large number of spreadsheets, compared to other known corpora, which were developed and used in an industrial context.

3.3.1 SETUP

We follow a two-step approach to identify a cross-table in Enron spreadsheets.

First, we automatically identify all of the spreadsheets which include at least one value of a particular intersection cell. Subsequently, the resulted subset includes the spreadsheet candidates which may have one form of a cross-table inside. To filter out the actual cross-tables from other structures of data, our **second step** consists of manually analyzing the resulted subset of spreadsheets. These two steps are based on self-defined criteria which we detail next.

3.3.2 CRITERIA

For the first step, criteria include the values of intersection cells that the automatic search will use. We recall from Section 3.2.1 that an intersection cell may include many possible values such as a single character, a digit or a shape. Since our analysis is exploratory, and is carried out without previous domain knowledge of the content of Enron spreadsheets, we decide to limit the values we search for to two values:

X: Mostly used by users in cross-tables.

Y: An abbreviation of Yes, indicating a positive selection of an attribute.

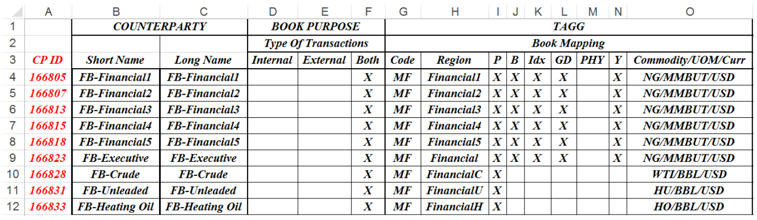
For the second step, the manual verification of the existence of a cross-table, our criteria follow the general definition of a cross-table, examples shown in Figure 3.1a. For this, an actual cross-table is identified if it has the following three criteria: (1) The *“objects”* and *“attributes”* are visually recognizable as two separate sets of data. (2) There are one or more intersection cells found in the area between objects and attributes. (3) The value of the intersection cell is used as an indicator to a relation, not as a data value by itself.

3.3.3 RESULTS

The summary of our analysis is presented in Table 3.1. Initially, Enron dataset consists of 16,160 spreadsheets. The first step of the analysis generated a subset of 1,524 spreadsheets. After the manual verification in the second step, we established that 552 spreadsheets in the subset contain at least one form of a cross-table. In other words, at least 3.42% of

Table 3.1. Results of the analysis performed on the Enron dataset (16,161 spreadsheets), to identify cross-tables structures in the spreadsheets.

Interaction Mark (Case Insensitive)	Spreadsheets with a *"candidate"* cross-table	Spreadsheets with a *"verified"* cross-table
X	1,177	542
Y	347	10
Total	1,524	552



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1		*COUNTERPARTY*		*BOOK PURPOSE*			*TAGG*								
2				*Type Of Transactions*			*Book Mapping*								
3	*CP ID*	*Short Name*	*Long Name*	*Internal*	*External*	*Both*	*Code*	*Region*	*P*	*B*	*Idx*	*GD*	*PHY*	*Y*	*Commodity/UOM/Curr*
4	*166805*	*FB-Financial1*	*FB-Financial1*			*X*	*MF*	*Financial1*	*X*	*X*	*X*	*X*		*X*	*NG/MMBUT/USD*
5	*166807*	*FB-Financial2*	*FB-Financial2*			*X*	*MF*	*Financial2*	*X*	*X*	*X*	*X*		*X*	*NG/MMBUT/USD*
6	*166813*	*FB-Financial3*	*FB-Financial3*			*X*	*MF*	*Financial3*	*X*	*X*	*X*	*X*		*X*	*NG/MMBUT/USD*
7	*166815*	*FB-Financial4*	*FB-Financial4*			*X*	*MF*	*Financial4*	*X*	*X*	*X*	*X*		*X*	*NG/MMBUT/USD*
8	*166818*	*FB-Financial5*	*FB-Financial5*			*X*	*MF*	*Financial5*	*X*	*X*	*X*	*X*		*X*	*NG/MMBUT/USD*
9	*166823*	*FB-Executive*	*FB-Executive*			*X*	*MF*	*Financial*	*X*	*X*	*X*	*X*		*X*	*NG/MMBUT/USD*
10	*166828*	*FB-Crude*	*FB-Crude*			*X*	*MF*	*FinancialC*	*X*						*WTI/BBL/USD*
11	*166831*	*FB-Unleaded*	*FB-Unleaded*			*X*	*MF*	*FinancialU*	*X*						*HU/BBL/USD*
12	*166833*	*FB-Heating Oil*	*FB-Heating Oil*			*X*	*MF*	*FinancialH*	*X*						*HO/BBL/USD*

Figure 3.4. A complex cross-table found in Enron dataset: objects are in two columns, attributes are in multiple (hierarchical) rows.

spreadsheets in the Enron dataset include one cross-table form. Figure 3.4 shows one of the cross-tables found in the Enron dataset.

In addition to identifying cross-tables, the analysis shows that particular cross-tables are context-related and found in a subset of spreadsheet files. For example, cross-tables similar to the one shown in Figure 3.4 are developed and used frequently in more than 100 spreadsheets in Enron dataset. The content of the cross-table in this example suggests that it was part of a business process of risk book requests. It mentions an information system called *"ERMS"*, standing for Enron Risk Management Services. However, identifying the exact business context remains difficult without the user domain knowledge.

3.4 APPROACH

Our approach aims to extract cross-tables data from a set of related spreadsheets. The approach follows four steps to achieve its aim, as summarized in Figure 3.5.

- Step 1 - Identify the dimensions of a *"potential"* cross-table: For a given worksheet, the approach uses an algorithm based on the GyroSAT, in addition to keywords and configuration parameters specified by the end-user, to decide three attributes of a cross-table rectangle: the minimum row and column, the maximum row and column, and the header row number.

- Step 2 - Transform the cross-table into key-value data tuples: For each detected cross-table, the approach perform transformations on the cross-table to construct the related key-value tuples.

- Step 3 - Decide the common keys from the transformed cross-tables: Our approach considers all the cross-tables identified and transformed from multiple spreadsheet files, and subsequently decides the common keys in the data tuples. These keys will be the column names in the target relational table.

- Step 4 - Generate the output file: The output file, encoded as an SQL script, contains the statements to create the structure, and fill the data of the target relational table.

Prior to the application of the approach, the user should provide the following parameters and textual keywords:

(a) Keywords which can be found in the upper and lower left corners of the cross-table. For example, in Figure 3.2, the user may supply {upper:*"licenses"*, upper: *"subscription"*, lower: *"total"*}.

(b) Intersection mark: the value used to indicate an intersection between a cross-table object and its attributes. In Figure 3.2, this value is *"X"*.

(c) Number of empty columns to skip: a parameter used in the identification of the cross-table dimensions.

(d) Has adjacent data: A boolean value indicating whether an attribute has related data in the adjacent column to its right. In fact, this value indicates whether the cross-table is of type single-valued or many-valued.

Following, we describe in details the algorithm and components of the four steps.

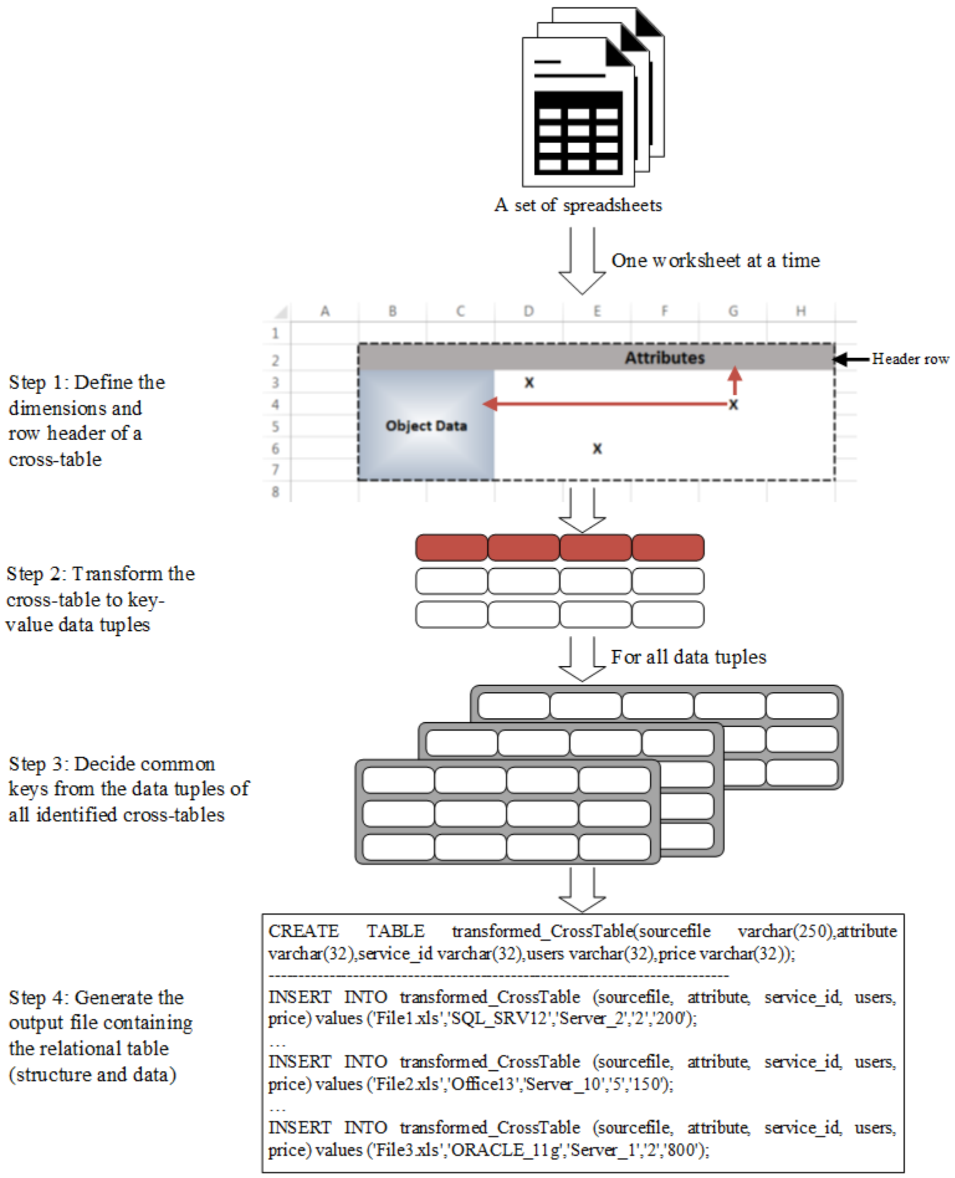
Step 1 - Identify the dimensions and header row of a *"potential"* cross-table:

(a) **Dimensions**: The user-specified keywords aid our approach to define the top left corner (minimum row, minimum column) and the bottom left corner (maximum row, minimum column) of a cross-table. To identify the maximum column of the cross-table, our approach performs a ***data-block-based*** search, taking advantage of the data block definition in GyroSAT [101]. First, an initial *"maximum column"* value is retrieved from the first identified data block. Thereafter, the maximum column value is recursively updated whenever a new data block is found under the condition that the data block position is within the values of the three known dimensions (minimum row and column, and maximum row).

Once no more data blocks are found, we force an extension of the rectangular area to find adjacent data blocks to the right. The extension allows to skip a number of columns, identified by the end-user. Thereafter, the search locates data blocks starting in the extended area. This is performed because cross-tables may contain empty columns, and an empty column forces the GyroSAT to stop expanding a data block horizontally. We apply the extension action, until no further data blocks are found even with the forced extension.

(b) **Header row**: The header row contains two types of data:

- Label values, which categorize and describe the data cells underneath.


A set of spreadsheets
One worksheet at a time
Step 1: Define the dimensions and row header of a cross-table
Attributes
Header row
Object Data
Step 2: Transform the cross-table to key-value data tuples
For all data tuples
Step 3: Decide common keys from the data tuples of all identified cross-tables
Step 4: Generate the output file containing the relational table (structure and data)

```
CREATE TABLE transformed_CrossTable(sourcefile varchar(250),attribute
varchar(32),service_id varchar(32),users varchar(32),price varchar(32));
------------------------------------------------------------------------
INSERT INTO transformed_CrossTable (sourcefile, attribute, service_id, users,
price) values ('File1.xls','SQL_SRV12','Server_2','2','200');
...
INSERT INTO transformed_CrossTable (sourcefile, attribute, service_id, users,
price) values ('File2.xls','Office13','Server_10','5','150');
...
INSERT INTO transformed_CrossTable (sourcefile, attribute, service_id, users,
price) values ('File3.xls','ORACLE_11g','Server_1','2','800');
```

Figure 3.5. Approach summary showing the steps to extract and transform cross-table data from multiple spreadsheets to one relational table encoded in SQL.

- Data values, when the cells below the header are the intersection cells.

Using the cell classification concept of the GyroSAT, the header row is found as *the row with the largest number of cells of type label, within the dimensions already found*. The search is performed from top to bottom.

The output of this step is the dimensions and header row of a cross-table. The cross-table cannot be identified when the keywords are not matched or the intersection value is not found. In this case, the approach moves to another worksheet.

Step 2 - Transform a cross-table: After identifying the targeted cross-table, data transformations are performed on the cells which lie within the cross-table's dimensions. The transformation starts by identifying the intersection marks, *"X"* in our case. For each intersection mark, the transformation builds a key-value data tuple. Specific transformation actions are followed to generate a data tuple. Following is the description of these actions depending on the type of cell found in the intersection row.

- Intersection: For the intersection cell itself, the data is found in the header. The key which describes this data value is implicit. As a resolution, we introduce the word *"Attribute"* as the key for the intersection cell data. The key-value pair
  $\boxed{\textit{"Attribute", ValueIn(Intersection column, Header row)}}$
  For Figure 3.2, the top-left intersection mark has the key-value pair = (*"Attribute"*,*"W2K8_R2"*).

- Adjacent data: When the cross-table is a many-valued cross-table, a data cell is always adjacent to the right of an intersection cell. In Figure 3.2, the cells in the *"Price"* column are always adjacent to the right of the intersection cells. The user defines the boolean parameter *"HasAdjacentData"* to tell the approach whether or not it should consider these data cells in the transformation. In our example, the parameter is set to True and thus we build the key-value pair for the *adjacent data cell* as:
  $\boxed{\textit{ValueIn(Adjacent column, Header row), ValueIn(Adjacent column, Intersection row)}}$
  For Figure 3.2, the adjacent cell has the key-value pair = (*"Price"*,*"100"*).

- Related (object) data: According to the cross-table definition, an intersection mark relates objects data with their attributes. The object data cells are found to the left of an intersection mark. They should be both non-empty and do not fall in the ignored type of cells. The key-value pair for a *related object cell* is:
  $\boxed{\textit{\{ValueIn(Object column, Header row), ValueIn(Object column, Intersection row)\}}}$
  For Figure 3.2, the object cell describing the *"Server ID"* has the key-value pair = (*"Server_ID"*,*"Server_1"*).

- Ignored: Our approach ignores two types of cells. First, we ignore other intersection cells within the same row, since they will have their own key-value tuple. Secondly, we ignore cells adjacent to other intersection cells, since one adjacent cell is linked to each intersection cell.

Step 3 - Decide common keys from the transformed cross-tables: The keys used in the cross-table data tuples are going to be the column names of the target relational table. To decide these keys, from all the transformed cross-tables, we perform two actions. First, we identify the unique keys per cross-table's data tuples. Second, we measure the coverage for each unique key over the whole set of identified cross-tables. The coverage of a key is the devision of the count of its unique occurences over the total count of the identified cross-tables. Keys that have a coverage above 80% are included as columns in the target relational table. The integration method we follow may be considered simple, for this we provide further discussion in Section 3.7.

Step 4 - Generate the output file: In this step, the approach generates the target relational table, which is encoded in a standard SQL script. The SQL script includes a *"create"* statement representing the structure, and *"insert"* statements representing the data. The columns of the relational table are the common keys decided from the previous step, in addition to the mandatory columns: *"attribute"* and *"sourcefile"*. All columns are assigned a text data type, which is chosen to eliminate additional complexities of type detection and conversion. The generation of the SQL output file starts by making the create statement of the target table. Thereafter, the approach processes the collection of key-value data tuples for each cross-table. For each data tuple an insert statement is generated. Again, the insert statement includes the pairs whose keys are chosen in advance. When a data tuple does not include a pair for one of the chosen keys, we assign to its key the default value *"empty_by_approach"*.

3.5 EVALUATION: CASE STUDY

We evaluate our approach presented in the previous sections, through a case study. In this section, we assess the performance of our approach first quantitatively by answering the following:

RQ1 How many cross-tables and intersection marks were extracted by the approach?

RQ2 Did the approach fail to detect a targeted cross-table completely? If yes, how many?

RQ3 Did the approach fail to extract particular intersection marks from the successfully detected cross-tables? If yes, how many?

Following the quantitative analysis, we perform a qualitative assessment through an interview with two frequent spreadsheet users in the company.

3.5.1 CONTEXT AND DATASET

Solvinity[5] is an IT solution provider. Their finance department uses spreadsheets for calculating the monthly bills for each customer. Within a billing spreadsheet a cross-table, similar to the one in Figure 3.2, is used to record the sold third-party licenses. An internal audit required the verification of licenses from all available sources, including the spreadsheets. Collecting the required cross-table data was challenging to users, especially

[5]https://www.solvinity.com/

Table 3.2. Summary of the case study results, after applying our approach to extract and transform cross-table data. Further details in Section 3.5.2.

Result Class	Cross-table Class	Cross-table Count	Cross Marks Count	Performance (Identify Cross-table)	Performance (Transform Cross Mark)
Succeeds	Complete Cross-tables	1,442	14,099	99.17%	92.83%
	Partial Cross-tables	6	1,182	0.41%	
Misses	Complete Cross-tables	6	1,139	0.41	7.17%
	Partial Cross-tables	6	41	0.41%	
Total		1,454	16,461	100%	100%

with all the manual work needed. Thus, our approach was applied to extract license data in cross-tables, into one relational table. The output was subsequently used by the auditing team for further analysis. The spreadsheet dataset on which the approach operated includes 333 spreadsheets with 2,801 worksheets.

3.5.2 RESULTS

In a mixed method evaluation, we performed both quantitative and qualitative methods to analyze the results of our approach. Quantitative analysis assesses the performance of the extraction and transformation approach by answering the questions (RQ1,RQ2 and RQ3) presented earlier. In addition, we interview two frequent spreadsheet users, to asses the approach qualititively. Before the interviews, these users were asked to manually validate a selected subset of spreadsheets against their extracted data.

QUANTITATIVE EVALUATION

Among the 333 spreadsheets, the spreadsheet parser failed to read the contents of two spreadsheets. However, we checked these two files manually, and no cross-tables were detected. This leaves 331 spreadsheet files for the approach to analyze, with a total of 2,801 worksheets. Table 3.2 represents the summary of the approach performance in extracting and transforming cross-tables, and cross-table marks. To understand the statistics better, we answer the research questions:

RQ1: How many cross-tables and intersection marks were extracted and transformed by the approach?

The approach detected 1,448 cross-tables. Within these cross-tables, 15,281 intersection marks were extracted and transformed into one relational database table. SQL statements in the output file were syntactically correct. Out of the 2,801 worksheets, we found that no cross-table data were extracted from 1,353 worksheets (48.3%). The reasons for not extracting a cross-table may be that the worksheet did not include a cross-table in the first place, or that the approach detection failed due to wrong keywords for example. To understand the actual reason, we further analyze this result in RQ2 and RQ3.

RQ2: Did the approach fail to detect a targeted cross-table completely? If yes, how many?

Among the 1,353 worksheets from which no cross-tables were extracted, 1,347 worksheets did not include any cross-table inside. The remaining 6 worksheets included 6 cross-tables, one per worksheet. The approach failed to detect these six cross-tables, because it failed to handle the extra number of empty columns in these cross-tables. The six *"missing"* cross-tables includes 1,139 marks, which as a result, were not transformed to the target relational table. Nevertheless, in another run of the approach, we increased the parameter *"Empty Columns to Skip"* to 2, and these cross-tables were successfully extracted. We refrain from setting the *"Empty Columns to Skip"* parameter to a large value, since this will increase the risk of extracting other irrelevant data lying next to the cross-table form in the worksheet.

RQ3: Did the approach fail to extract particular intersection marks from successfully detected cross-tables? If yes, how many?

In total 1,448 cross-tables were detected by the approach, one cross-table per worksheet. To verify that all the marks in these cross-table were transformed, we performed a separate analysis. We calculated the counts of all cells with the mark value X, per worksheet. Subsequently, we validated these counts against the number of marks extracted by our approach for each cross-table. What we found is that 1,442 cross-tables were completely migrated into the relational table, while 6 cross-tables were partially transformed. The missing cross-table marks in this case are 41. The root cause for missing these cross-table marks is the same that caused the approach to miss the complete cross-tables: the large number of empty columns within the cross-tables structure.

QUALITATIVE EVALUATION

In addition to the quantitative evaluation, we interviewed two spreadsheet users. The users worked at the finance department, and their daily job is directly related to the spreadsheets under analysis. One user has been developing and maintaining these spreadsheets for six years within the accounting team. The other user has experience in accounting as a business controller, and he joined the team recently. Prior to the interviews, the end-users were provided with a subset of 30 spreadsheets, selected from both successful and failed cases of the approach. We additionally provided the relational table corresponding to the cross-table data from these 30 files. With these data, the end-users were asked to validate the completeness and contextual correctness of the extracted data by our approach.

Users perspective: When asked about the correctness of the information retrieved, according to the context, both users found them completely correct. One user highlights that *"no factual errors"* were identified in the generated relational table. On the completeness of the retrieved data, one user only was able to detect the missing cross-table marks from the target relational table. The users, after completing their verification, gave a high rating for the approach's extraction capabilities, and regarded it as *"the only gathered data for the sold licenses"*. In addition, the approach was described as *"useful"*, especially for *"these kinds of auditing projects"*, and saving a lot of manual work. However, one user in particular highlighted that *"there will always be a risk"* of extracting *"wrong data"*. He reasoned that the spreadsheets are *"manually designed"* and do not follow a strict template. He concluded that adopting *"more standardized"* designs in the spreadsheets will minimize this risk in the future.

3.6 RELATED WORK

Cunha *et al.* [63] and Hermans *et al.* [101] targeted the transformation of data inside a spreadsheet into a class diagram, for the sake of improving spreadsheet comprehension through visualization. Both works focused on tabular data inside spreadsheets, and were effective in the detection and transformation to class diagrams, one spreadsheet at a time. However, the prototypes were evaluated on a small number of spreadsheets, and no aggregation approach was considered. Cunha *et al.* [64] implemented an approach that maps spreadsheet data into a relational database, with the aim of normalizing data in a spreadsheet. The work, however, was evaluated on simple spreadsheets without addressing any semi-structured data forms. All mentioned work aimed at keeping the end-user working within the spreadsheets, but with an improved environment and understanding. Among the work that aimed at migrating from spreadsheets is Senbazuru [50, 52]. They targeted the extraction of hierarchical data from spreadsheets into a relational database. Their approach, however, is more domain specific, and does not consider aggregating similar data in multiple spreadsheets.

3.7 DISCUSSION

In prior sections, we described an approach that extracts data from cross-tables in spreadsheets. In this section, we highlight some issues related to our approach.

Using the Spreadsheet Formulas: Formulas are an essential part of spreadsheet development. One cell of data may be the output of a series of formulas which use multiple data cells and ranges. Our current approach considers data in cross-tables, for a more complete migration, the formulas should be considered. Considering formulas' parameters and calculations as part of the data migration may help in building a better relational structure of the extracted data. However, the automatic parsing and translation of formulas to another language is a complex process, therefore it can be an area to explore in future work.

More Complex Cross-tables: Cross-tables may vary in design to some extent because of the flexibility provided to the end-user, and the nature of the cross-table itself. In our approach, we considered cross-tables with simple attributes, with one row of data. However, users may build more complex, and hierarchical headers. Since previous research extracted hierarchical data structures from spreadsheets [51, 52], it is a viable option in future to incorporate hierarchical data detection to widen the application of our approach.

Data Integration: The process of data integration includes the activities performed, and the approaches followed to combine heterogeneous data from multiple sources [140]. Our approach followed a two-step method: first is to decide the common columns from multiple files, and second is to do an exact match of key names. Even though the method is considered simple, it showed near perfect results in the study evaluation, due to a high level of consistency in the naming of columns. However, variation in the naming may occur in another setup. One way to improve is to consider a general schema resource in order to compare and integrate column headers, or a to build a schema dictionary from within the spreadsheet dataset, prior to to the extraction [51].

3.8 CONCLUSIONS AND FUTURE WORK

In this chapter, we provided the design and implementation of an approach aimed to extract and transform data in cross-tables from multiple spreadsheet files, into one relational table.

Our approach was successfully evaluated in an industrial case study, where the resulted table was used in further analysis. The approach succeeded in minimizing the human efforts in the extraction, compared to a manual process. Results show that the approach was able to transform 92.83% of the data in the targeted cross-tables, based on the intersection mark detection. In future work, we aim at eliminating the user role in the extraction, through adopting a machine learning algorithm.

4 Identifier Naming Practices in Scratch

Research shows the importance of selecting good names to identifiers in software code: more meaningful names improve readability. In particular, several guidelines encourage long and descriptive variable names. A recent study analyzed the use of variable names in five programming languages, focusing on single-letter variable names, because of the apparent contradiction between their frequent use and the fact that these variables violate the aforementioned guidelines.

In this chapter, we analyze variables in Scratch, a popular block-based language aimed at children. We start by replicating the above single-letter study for Scratch. We augment this study by analyzing single-letter procedure names, and by investigating the use of Scratch specific naming patterns: spaces in variable names, numerics as variables and textual labels in procedure names.

The results of our analysis show that Scratch programmers often prefer longer identifier names than developers in other languages, while Scratch procedure names have even longer names than Scratch variables. For the single-letter variables, the most frequent names are x, y, and i. Single-letter procedures are less popular, but show more tendency to be in upper case. When compared to the other programming languages, the usage of single uppercase letters in Scratch variables seems to be similar to the pattern found in Perl, while for the lowercase letters—to the pattern found in Java. Concerning Scratch specific features, 44% of the unique variable names and 34% of the projects in the dataset include at least one space. The usage of textual labels between parameters in procedure names appears as not common, however textual patterns used imply an influence from textual languages, for example by using brackets.[1]

[1]This chapter is based on our paper: How do Scratch Programmers Name Variables and Procedures? (Alaaeddin Swidan, Alexander Serebrenik and Felienne Hermans) which appears in the proceedings of IEEE 17th International Working Conference on Source Code Analysis and Manipulation, SCAM 2017

4.1 INTRODUCTION

The naming of identifiers in the source code has been extensively studied (see, e.g., recent studies of this subject [6, 14, 24, 38, 106, 137, 196, 222]). Still, the impact of the variable name choice on code readability and maintainability is controversial, as witnessed, e.g., by recent studies of Beniamini et al. [24] and Hofmeister et al. [106] reaching contradictory conclusions.

Furthermore, many computer science and programming curricula focus on the programming concepts and the syntax of the languages as opposed to practices in naming variables and identifiers. Indeed, while "meaningful variable names" are advocated by some teachers [97, 117] and practitioners [188] neither the ACM Curriculum lines for Undergraduate Programs in Computer Science[2] nor the Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering[3] discuss this topic. In fact, as discussed by Raymond [182] standard metasyntactic variables used in syntax examples are *"foo"* and *"bar"*. The names of these identifiers are meaningless, and to some extent, they represent a refusal to name, suggesting to the learner that naming is less important to the programming task.

The goal of this chapter is to contribute towards **understanding the patterns in variable and procedure naming in Scratch**. Scratch is a block-based visual language developed by MIT with the aim of helping young people learn the basic concepts of programming and collaboration Scratch has recently become very popular among school-age children and in several countries has been introduced as part of the school curriculum as a means to teach programming [191, 225]. Moreover, the overall popularity of Scratch is witnessed by Scratch being currently rated 19 in the TIOBE index[4], topping such languages as Lua, Scala and Groovy and since early 2014 exhibiting an increasing trend shown in Fig. 4.1.

Scratch programming materials too do not focus on naming. For example, the Creative Computing Learner Workbook created by the ScratchEd group at Harvard does not explain how to select good names for procedures and variables[5]. It is therefore interesting to explore

[2]http://www.acm.org/education/CS2013-final-report.pdf
[3]http://www.acm.org/binaries/content/assets/education/se2014.pdf
[4]https://www.tiobe.com/tiobe-index/
[5]http://scratched.gse.harvard.edu/guide/files/CreativeComputing20140820_LearnerWorkbook.

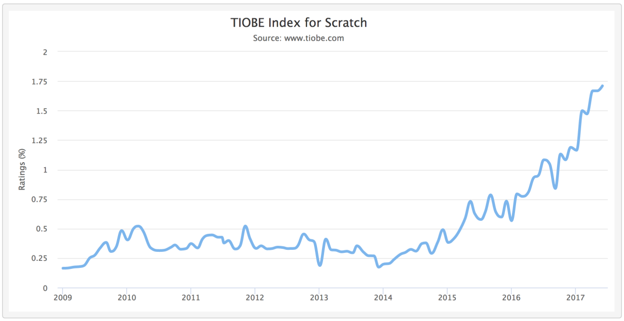


Figure 4.1. TIOBE Programming Community index: evolution of the popularity of Scratch

the naming habits of Scratch programmers. There are other reasons why understanding the naming practices in the Scratch community is important. First, it is important for the Scratch community itself as bad naming practices can easily propagate from one program to another through ‘remixing’ [70, 105], a code sharing practice similar to GitHub forking [181]. Second, it is important for researchers. Software engineering researchers can learn how to support novice programmers, taking their first steps in programming. As shown in Fig. 4.2, 51% of Scratch programmers are aged between 10 and 15 at the time of registration according to the statistics provided by Scratch[6]. Therefore, researchers of software engineering education can obtain insights in how to define naming guidelines for educational materials, and analyzing the differences between Scratch and textual languages can help in supporting the transition from visual languages to textual ones [68, 145].

We start by a **general discussion of naming practices** in Scratch and analyze the previously published collection of 250,000 Scratch projects [4]. We replicate two studies from a recent paper by Beniamini et al. [24]. Similarly to Beniamini et al. we investigate the distribution of the lengths of variable names and study popularity of single-letter variable names such as *i* and *x*. As opposed to Beniamini et al. who focused on variable naming in five mainstream programming languages we focus on Scratch. Furthermore, while Beniamini et al. solely focused on the names of the variables, we repeat their study for procedure names as well.

Variable names in Scratch range mostly between 4 and 10 characters, procedure names tend to be longer. For the single-letter variables, the most commonly used names are *x*, *y* and *i*, procedures—*a*, *y* and *r*. Compared to the other programming languages, single-letter variable names are less common in Scratch and overall Scratch variables have longer names. The usage of single uppercase letters is similar to Perl, for the lowercase—to Java.

Next we focus on **Scratch-specific features in naming identifiers**. Scratch supports a number of less commonly used naming features, for example spaces may be used in names e.g., a variable *max i* and integers and even floating point numbers can be used as variable names e.g., a variable named *6* or *3.14159*. Finally, Scratch allows for textual labels in between parameters. For example a method for printing the first *n* letters from a string *s* could be called “printnof(*n*,*s*)” in a textual language. Scratch allows for this too, however, one can also define a procedure called “say *n* characters from text *s*", as shown in Fig. 4.5. This feature does exist in textual languages too—most notable in SmallTalk—but is not commonly found in most mainstream languages.

Investigating the use of these Scratch-specific naming patterns is interesting to understand their role in novice programming. If they are popular among Scratch programmers, this might be because they ease novice programming, and that means one could even advocate that these features should be integrated in the textual programming languages, if only to ease the transition for the block-based languages programmers.

pdf

[6]https://Scratch.mit.edu/statistics/

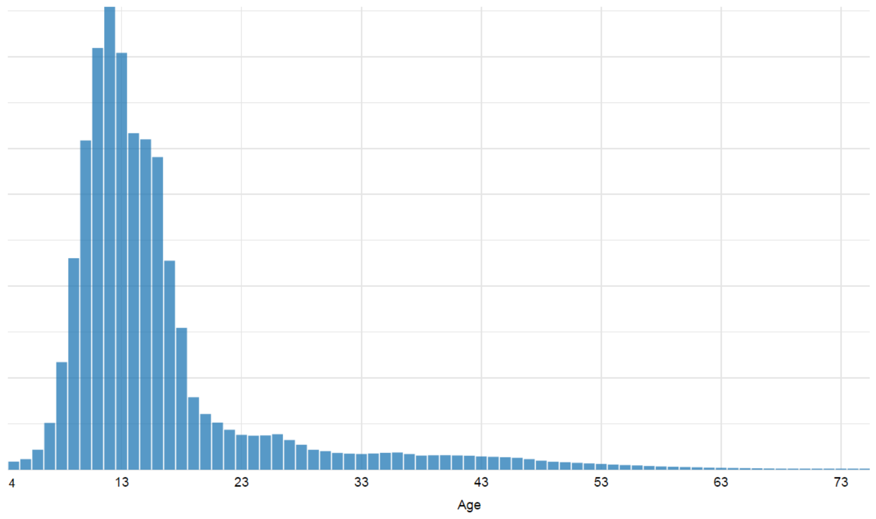


Figure 4.2. Age distribution of Scratch users at the time of registration according to Scratch statistics web-page

Spaces in variable names are common: 34% of projects use this feature. Numeric values as identifiers are rarely used, and mostly represent constants or parts of the data structure. The usage of textual string between parameters is not as common; however, textual patterns used imply an inference from textual languages, e.g., by using brackets.

4.2 RELATED WORK

Naming identifiers in software code has been studied extensively in the past decades [6, 9, 14, 24, 37, 38, 42, 106, 133, 137, 195, 196, 213, 217, 222]. In practice, identifiers constitute a major part of the source code: e.g., in Eclipse 3.0M7 which is tantamount to 2 MLoC, 33% of the tokens and 72% of characters correspond to identifiers [71].

For a human reading code, it is crucial to the understanding of code to also understand what the identifiers mean. As such, several studies have investigated the link between identifier naming and code readability and comprehension [14, 106, 133, 213, 217] or identifier naming and externally observable aspects of the software development process that are expected to be affected by comprehension such as change-proneness [6], quality [38, 137] and presence of faults [196, 222]. Caprile and Tonella [41] have observed that names of C functions consist of on average of 2.04–3.36 words, often verbs expressing action, while Caracciolo et al. observed that most method names in Java and Smalltalk also tend to consist of several words but rarely more than five words [43].

Going beyond the discussion of whether variable names should be shorter or longer, Arnaoudova et al. [11, 12] have studied linguistic anti-patterns, “recurring poor practices in the naming, documentation, and choice of identifiers in the implementation of an entity” such as discrepancies between the behavior implied by the identifier and the corresponding comment, while Høst has studied whether the Java method names relate to the behaviour of those methods [108]. In the educational setting Glassman et al. propose Foobaz, a tool giving semi-automatic feedback on student variable names based on the values the variable can take during the execution and limited input from the teacher [87].

While visual languages such as Scratch have recently become a favorable choice for schools as an introduction to programming [145], the lion's share of the previous work on identifier naming has focused on textual languages. Notable exception is the recent work of Moreno and Robles [150]: they observed that Scratch programmers often do not change the sprite names that were automatically generated by the environment. van Zyl et al. have observed that one of the interviewed school teachers working with Scratch has taught the students to integrate variable types in their names, e.g., 'S' for Strings [225]. Finally, Hermans and Aivaloglou encouraged the students in Scratch MOOC to choose meaningful names and to avoid keeping default ones [97].

Understanding the naming patterns and preferences of students learning how to program with visual languages is essential to act upon the difficulties faced by students when transitioning to textual programming languages. According to Kolling et. al. [126], for these learners, dealing with user-defined identifiers is one of the *"fundamental challenges"*. It involves an extra cognitive load to remember identifiers, with case sensitivity in some cases, instead of selecting a variable from a drop-down list in Scratch. It also relates to the broader challenges of spelling and writing for these students.

4.3 Relevant Scratch Concepts

We introduce several core features of Scratch required for understanding the reminder of the chapter. Readers are referred to *"Creative Computing"* [34] for an extensive overview.

Scratch is a block-based programming language aimed at children, developed by MIT. Scratch can be used to create games and interactive animations, and is available both as a stand-alone application and as a web application. Figure 4.3 shows the Scratch user interface in the Chrome browser.

Sprites

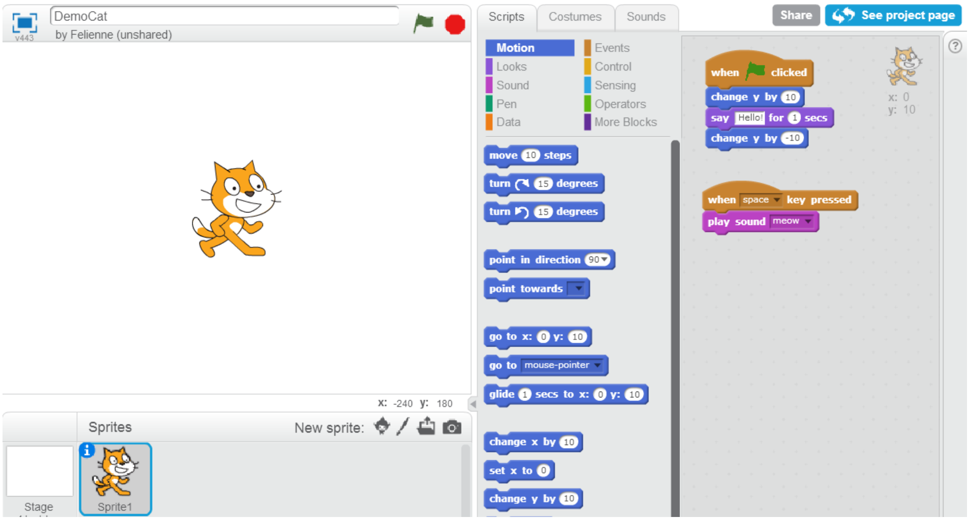
Scratch code is organized into 'sprites': two-dimensional pictures that each have their own source code. Scratch allows users to bring their sprites to life in various ways, for example by moving them in the plane, having them say or think words or sentences via text balloons, but also by having them make sounds, grow, shrink and switch costumes.

The Scratch program in Fig. 4.3 consists of one sprite, the cat, which is Scratch's default sprite and logo[7]. The code in the sprite will cause the cat to jump up, say "hello", and come back down, when the green flag is clicked, and to make the 'meow' sound when the space bar is pressed.

Scripts

Source code within sprites is organized in scripts: a script always starts with an event, followed by a number of blocks. The Scratch code in Fig. 4.3 has two distinct scripts, one started by clicking on the green flag and one by pressing the space bar. It is possible for a

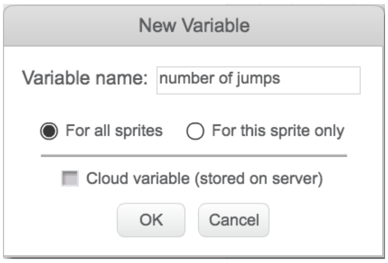
[7]`https://Scratch.mit.edu/projects/97086781/`



```
when [flag] clicked
change y by 10
say Hello! for 1 secs
change y by -10

when space key pressed
play sound meow
```

Figure 4.3. The Scratch UI consisting of the ‘cat’ sprite on the left, the toolbox with available blocks in the category ‘motion’ in the middle and the code associated with the sprite on the right.



New Variable

Variable name: number of jumps

● For all sprites ○ For this sprite only

☐ Cloud variable (stored on server)

OK Cancel

Figure 4.4. The Scratch user interface to create a variable

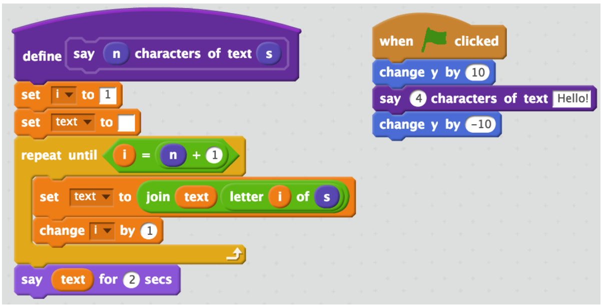
single sprite to have multiple scripts initiated by the same event. In that case, all scripts will be executed simultaneously.

Variables

Like most textual languages, Scratch programmers can use variables. Variables are untyped, but have to be ‘declared’ through the Scratch user interface, shown in Fig. 4.4. This figure also shows that, contrary to most programming languages, variable names in Scratch may contain spaces.

Procedures

Scratch allows programmers to create their own blocks, called procedures. They can have input parameters, and labels in between. Procedures are created with an interface similar to



```
define say (n) characters of text (s)
set i to 1
set text to
repeat until <(i) = ((n) + (1))>
    set text to (join (text) (letter (i) of (s)))
    change i by 1
say (text) for 2 secs
```

```
when ⚑ clicked
change y by 10
say 4 characters of text Hello!
change y by -10
```

Figure 4.5. Scratch code to define and invoke a procedure

the one to create variables. Figure 4.5 shows the definition and the invocation of a procedure in Scratch.

4.4 RESEARCH DESIGN AND DATASET

4.4.1 OVERALL DESIGN

The goal of this chapter is to compare naming practices of the Scratch programmers to those of the developers in mainstream programming languages. To that end, we start by partially replicating the recent work of Beniamini et al. on single-letter variables in Java, C, PHP, Perl, and JavaScript [24]. In terms of the classification of Shull et al. [203], we perform a dependent replication of the studies summarized in Fig. 1 and 2 of the original work. Inherently, the programming language is the only factor we vary when compared to the original study. However, as opposed to the data in the original study, Scratch programs are not available on GitHub. Hence, we use the dataset previously scraped and processed by Aivaloglou and Hermans [4]. We report on the results of these replications in Section 4.5.1. Furthermore, we perform another dependent replication of the same studies by considering Scratch procedures rather than variables.

Next, in Section 4.5.1 we perform a conceptual replication of the study of the single-letter variable types of Beniamini et al. [24]. The original study conducted a survey to understand the type-related user perceptions, with questions such as *“what type would you consider for a variable called ...?”*. We however focus on the types as used in the program. We investigate types *“as-being-used”*, as opposed to *“as-being-perceived”* in the original study due to the limited programming experience of the intended Scratch programmers. Scratch is meant for people taking their first steps in programming, such as school-age students, and we do not expect them to have established perceptions on data types of single-letters variables. As opposed to our work, in the original study, however, 30% of survey respondents claim 10-years experience in programming, while 23% have programming knowledge in six different languages or more [24]. Furthermore, we study types as used as opposed to types as defined, since Scratch does not have a concept of an explicit variable type. However, we can deduce the variable types from assignments involving those variables. For example, variable i in Fig. 4.5 represents an integer since it is assigned 1.

Finally, in Section 4.5.2 we report on the ways Scratch developers employ Scratch-specific naming practices such as spaces in variable names, numeric values as variables, and the use of textual labels in between parameters.

4.4.2 DATASET

We use the dataset created by Aivaloglou and Hermans [4], consisting of 250,000 Scratch projects scraped from the Scratch website in March 2016. We select the projects that use variables or procedures, which amount to 73,473 projects, 29% of the original dataset. Variables are used more often: 69,045 projects (27.6%) use variables, while 17,605 use procedures (7%). We used Python and R to process and analyze the data.

4.4.3 IDENTIFIER EXTRACTION

To follow the steps of the replicated study, we collect the unique variable names used in the projects' scripts. Within the scripts we identify Scratch blocks that are used to assign a value to the variables, : e.g., "Change *variable* by *value*" or "Set *variable* to *value*". Variable names in Scratch are unique: once a variable is declared in a project, its name cannot be used to create another variable in the same project, even in a different scope, e.g., in a different sprite.

We note that we focus solely on artifacts created specifically through "Make a Variable" button in Scratch UI, rather than other named entities, such as sprites. However, we believe sprites should be excluded from consideration, since we perform a dependent replication study to variable names in textual languages. Therefore, what we consider as a Scratch variable must hold a major property similar to variables in the textual languages: its name must be entered by the user. In Scratch, however, sprites are assigned default names automatically, and they often remain unchanged by the users [150].

To determine the type of single-letter variables, we perform type inference on the parameter value assigned to each variable. The inference is performed via standard data type conversion of the value. If the variable is accessed multiple times in a project with different data types, for example first as a string and then as an integer, both data types are counted.

For the procedure names, we identify the blocks used to call a procedure, extract the name of the procedure, and count the number of projects in which it occurs.

We note here that Scratch allows the user to create multiple procedures with the same exact name in the same sprite. It is not clear to us why the language would support such a feature. We argue, however, that counting the procedure names once per project is an indication of the naming patterns used and fits the needs of this study. For the presence of spaces in and numbers as variables we simply analyze the previously extracted variable names, while textual patterns in procedure names are detected from the extracted procedures' names. We provide the analysis code, input and output files for verification and replication purposes on a GitHub repository[8].

4.4.4 DATA ANALYSIS

Understanding differences in variable name lengths occurring between different programming languages requires comparison of multiple distributions. Such a comparison is traditionally performed as a two-step process consisting of (1) testing a global null hypothesis,

[8]`https://github.com/Felienne/ScratchVars`

that can be intuitively formulated as “all distributions are the same”, using ANOVA or its non-parametric counterpart, the Kruskal-Wallis test, and (2) performing multiple pairwise comparisons of different distributions, testing specific subhypotheses such as “distributions 2 and 4 are the same”. However, it has been observed that such a two-step approach can result in inconsistencies when either the global null hypothesis is rejected but none of the pairwise subhypotheses is rejected or vice versa [86]. Moreover, it has been suggested that the Wilcoxon-Mann-Whitney test, commonly used for subhypothesis testing, is not robust to unequal population variances, especially in the unequal sample size case [241]. Therefore, one-step approaches have been sought. We opt for one such approach, the **T**-procedure of Konietschke et al. [127, 128]. This procedure is robust against unequal population variances, respects transitivity, and has been successfully applied in empirical software engineering [66, 115, 224, 226, 227, 240]. We use the Tukey (all-pairs) contrasts to compare all distributions pairwise.

To understand differences between the distributions of single-letter variable names in different languages, we represent each programming language as a 26-dimensional vector with the dimensions corresponding to ‘a’, ..., ‘z’. For each language we consider two distributions: the distribution of the lower case letters (‘a’, ..., ‘z’) and the distribution of the upper case letters (‘A’, ..., ‘Z’). We compute the mean Euclidean distances between pairs of distributions and then perform hierarchical clustering based on the Euclidean distance.

When comparing distributions of variable name lengths with the procedure name lengths, the **T**-procedure is not applicable. Hence, we perform the Mann-Whitney-Wilcoxon test together with the two-sample test for the nonparametric Behrens-Fisher problem, i.e., test for $H_0 : p = 1˙2$, where p denotes the relative effect of the two independent samples [128, 158].

For space usage in variables’ names we do two things: (i) To understand how popular spaces are among all the names, we extract the space count per unique variable name. (ii) To understand the trend of using spaces across the dataset (multiple projects and multiple users), we extract per project the maximum space count found in the project’s variables. For the textual patterns in procedure names, we count the occurrence of each of the extracted token.

4.5 Results

This section presents an overview of our analysis of variable and procedure name used in the previously published Scratch dataset [4]. We start by replicating studies of Beniamini et al. [24] and proceed with investigating Scratch-specific features.

4.5.1 Replication Studies

Our first analyses are the replication studies regarding one letter variables.

Variable Name Length

The original study of Beniamini et al. [24] concluded that the single-letter variable names *“are approximately as common as other short lengths except in PHP”* and that *“in C, Java, and Perl they make up 9–20% of the names”*. Figure 4.6 shows the distribution of lengths in the Scratch dataset. A closer look at the data reveals that the single-letter variables constitute ca. 4% of all the variable names, i.e., less than the 9–20% observed by Beniamini et al. Compared to the five programming languages in the study of Beniamin et. al., single-letter variables seem to be less common in Scratch, while the maximum length of a variable’s

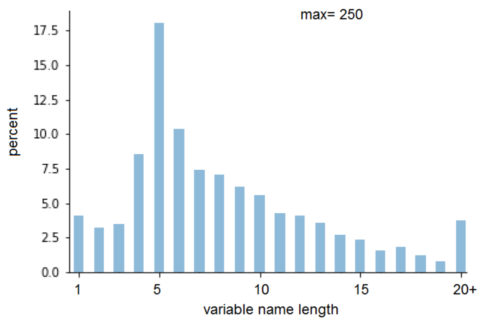


Figure 4.6. The distribution of variable's name length in Scratch.

name –250 characters– is significantly larger. These *Observation*s lead us to wonder whether overall the variable names in Scratch are longer than in other programming languages. To this end, we apply the **T**-procedure (Section 4.4.4) that reveals that variable names in Scratch indeed are longer than in the textual languages studied in Beniamini et. al. Moreover, variable names in Java tend to be longer than those in PHP, those in PHP than those in C, those in C than those in JavaScript, and finally, those in JavaScript than those in Perl. In all cases p-values have been too small to be computed precisely ($p < 2.2\text{x}10^{-16}$).

Single-letter Variable Names

Further we investigate the case of single-letter variable names. For the previously studied programming languages, Beniamini et. al. [24] highlight the following *Observation*s about the single-letter usage:

a) The most commonly occurring single-letter variable name is *i*. The authors attribute this to *i* being commonly used as a loop counter. As opposed to the studied mainstream languages, loops are performed in Scratch using predefined blocks. As illustrated on Fig. 4.7, the two left-most blocks –forever and repeat 10– do not require a variable to control the loop iterations. However, inside the loop the user will have no access to the built-in loop's iterator. When that is needed by the user, then the third block in Fig. 4.7 –repeat until– can be used. To understand its use see Fig. 4.5. In summary, because of the built-in language support to variable-free loops, we expect the usage of the variable name *i* to be less common in Scratch.

b) Apart from the popularity of *i* the distribution is language-dependent. Since Scratch is quite different from the programming languages considered by Beniamini et al., we expect the distribution of the single-letter variable names in Scratch to be different to the distributions in those languages. Hence, we expect the *similarity* between Scratch and the languages considered by Beniamini et al. to be lower than the *similarity* between those five languages in Beniamini et. al. study.

c) Finally, the authors observe that the lower case letters are used more frequently than the upper case letters. Since this is also the case for regular text in most natural languages as well, we expect the Scratch programs to follow the same pattern.

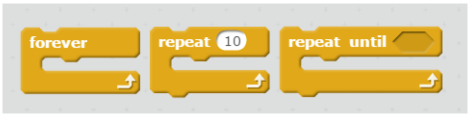


Figure 4.7. Scratch blocks that are used to repeat specific actions

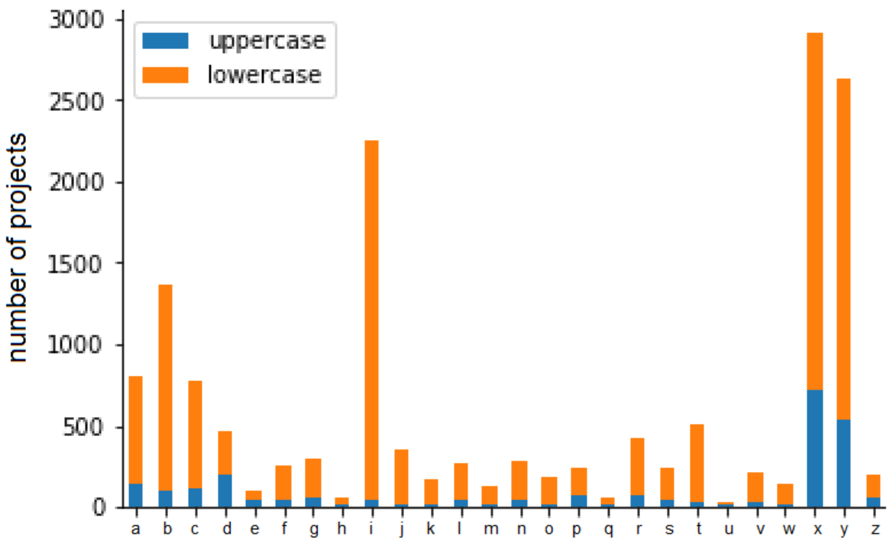


Figure 4.8. A histogram of single-letter variables occurrences in Scratch projects

Figure 4.8 shows the distribution of variables of one letter, in upper and lower case, in the Scratch dataset. Note that one Scratch project may contain both the upper and lowercase version of one variable, and they will refer to a different variable. Inspecting the data, we observe that similarly to the previous study *i* is the third most commonly occurring variable name as it is used in more than 2000 unique projects. Hence, we conclude that *contradicting our expectations Observation a*) above also holds for Scratch. Furthermore, we observe that *x* and *y* are extremely popular in Scratch. This can be explained by the fact that *x* and *y* represent the coordinates of the sprites on the stage, and when reading the position of a sprite, Scratch developers access the built-in *x* and *y* properties. As such, they form the basis of moving sprites in the 2-D stage. There seems to be an agreement on this use among the developers of textual languages studied in Beniamini et. al. When they were surveyed about variable name interpretations, *x* and *y* were commonly interpreted as “coordinates". In addition, built-in Scratch blocks often use *x* and *y* as shown in Figure 4.9. We conclude that Scratch programmers seem to be inspired by the Scratch language in naming variables. Next we study the similarity of Scratch to the five programming languages in terms of frequency distribution of single-letter variable names. Hence, we compare the mean Euclidean distance

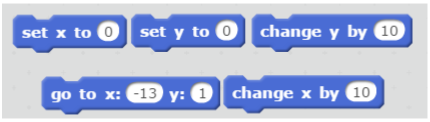


Figure 4.9. Examples of blocks using built-in variables x and y

between the pairs of the twelve distributions (the five programming languages plus Scratch, considered for the uppercase and the lowercase letters). The mean distance shows that the Scratch usage of the uppercase letters in the single-letter variable names is close to the ways the upper case letters are used in Perl, C and Java (mean distances between 36712.85 and 37076.95), while the way lowercase letters are used in Scratch quite similarly to the way letters are used in Java (Scratch—44572.57 and Java—46706.21). Hence, we claim that our expectation based on *Observation b*) has been confirmed for the uppercase letters and rejected for the lowercase letters. Closer look at Fig. 4.10 shows that the usage of single uppercase letters is similar to the pattern found in Perl, for the lowercase—to the pattern found in Java.

Finally, Fig. 4.8 clearly shows that the lowercase letters are much more frequently used as variable names than the uppercase letters, providing support for *Observation c*).

Single-letter variable names are less common in Scratch than in other programming languages (ca. 4% vs. 9–20%), and Scratch variables have longer names than variables in other programming languages.

Procedure Names

Going beyond the study of Beniamini, we additionally consider the naming of procedures in Scratch. For a detailed explanation of procedures in Scratch see Section 4.3 and Fig. 4.5. Figure 4.11 shows the distribution of the procedure names length in the Scratch dataset. By inspecting this figure, we observe that *the procedure names tend to be longer compared to Scratch variable names*. Indeed, the two-sample test for the nonparametric Behrens-Fisher problem estimates the relative effect of the two samples (procedure name lengths vs. variable name lengths) as 0.149 (with the *p*-value being too small to be computed precisely ($p < 2.2\text{x}10^{-16}$), i.e., it indicates that the procedure names' lengths tends to be larger than the lengths of the variable names. This *Observation* is additionally confirmed by the Mann-Whitney-Wilcoxon test (the *p*-value is too small to be computed precisely). Short procedure names are not common, they are even less common than variable names: single-letter procedure names compose less than 1% of the extracted names, less than 4.9% observed for Scratch variables and 9–20% in C, Java and Perl variables [24]. The maximum length for a procedure name is 250 characters, which is the same as the maximum length for the variable names. We suspect this exact match is caused by a language constraint that was imposed in previous versions of Scratch. The current version of Scratch, however, allows for names longer than 250 characters.

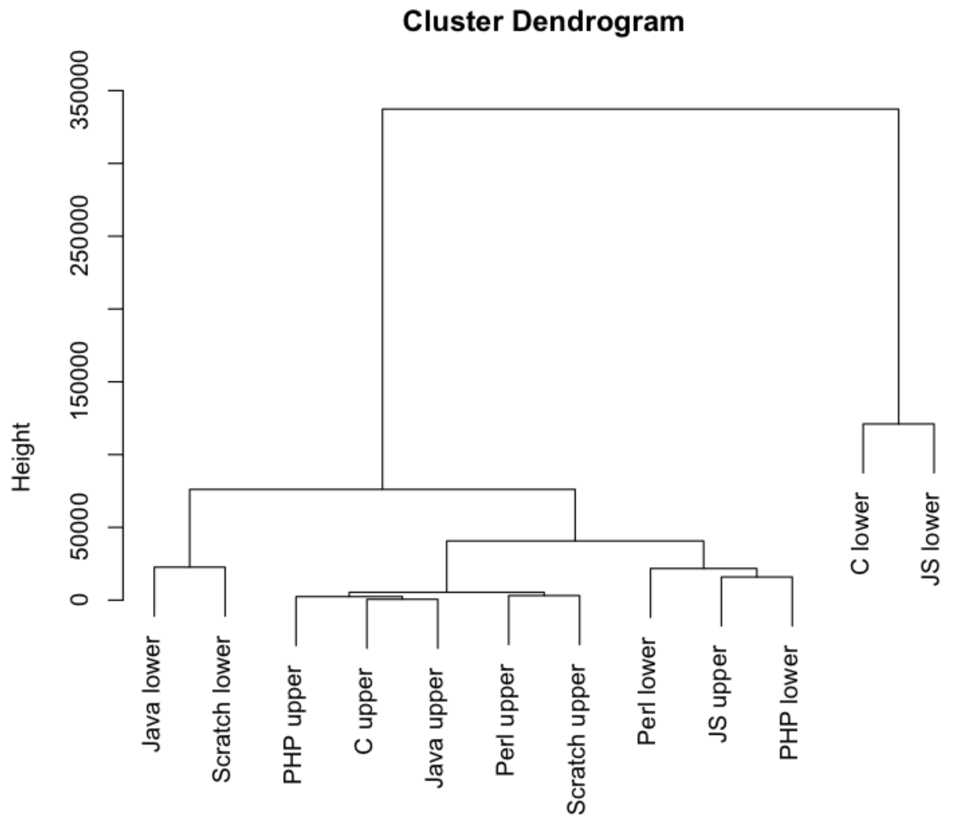


Figure 4.10. A cluster dendrogram of Scratch compared to other programming languages for the single-letter pattern

Next, we consider single-letter names for the procedures and revisit *Observation*s *a*) and *c*). We could not revisit *Observation b*) with respect to procedures since data on single-letter procedure names was not collected for the five programming languages in the study of Beniamini et. al. Figure 4.12 shows the number of occurrences for each alphabetic letter. We see that *i* is no longer among the most commonly used letters rejecting *Observation a*). The top used single-letter name is *a*, the first letter in the alphabet, which might explain its popularity. The uppercase letters are used more often than the lowercase letters (in 267 vs. 218 projects), rejecting *Observation c*).

TYPES

In the paper of Beniamini et al. [24] the authors observe that some letters are highly associated with the data type starting with the same letter: e.g., char for *c* and string for *s*. They also highlight that the integer data type is a common association for many other letters.

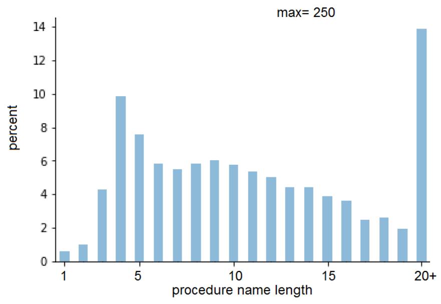


Figure 4.11. The distribution of procedure's name length in Scratch projects

Furthermore, a survey for developers showed that variable names *x*, *y* and *z* are commonly interpreted as coordinates, and for these letter it seems a balance exists between integer and float associations.

As explained in Section 4.4.1 we conduct a conceptual replication by considering the types of variables as they are used, as opposed to types as guesses by programmers. Figure 4.13 shows the distribution of single-letter variables with the types inferred. The majority of single-letter variables are encountered as integers in the Scratch dataset. This partially agrees with their *Observation* of integers being common for many letters, however, the data types are less diverse in Scratch compared to the five programming languages considered. One *Observation* that contradicts *Observation*s in the original study is related to the string data type. While string data type in the five programming languages studied is commonly associated with *s* and less frequently with many other letters, the strings are almost completely absent in the Scratch dataset apart from *i*. This is also observed for the other data types where no noticeable usage could be observed for floats, lists or strings. The only exception to some extent is the variable *c* where we observe some usage linked with the *char* data type. Floats as types for *x*, *y* and to lesser extent *z* seem to support the *Observation* that these single-letters variables are perceived as coordinates as suggested by Beniamini et al. [24].

4.5.2 Scratch-specific Constructs

In this section we analyze the occurrence of naming practices that are allowed in Scratch, but are missing from or are not common in most mainstream textual languages.

Spaces in Variable Names

Most textual programming languages do not allow spaces in variable names. FORTRAN ignores spaces, this means spaces could be used in FORTRAN variable names. However it does mean that 'apples' and 'app les' refer to the same variable. Other languages supporting spaces in variable names are SQL and some Scheme implementations. Even languages targeting data analysts rather than software developers recommend spaces be avoided [30]. To understand the usage of spaces in variable names, we measure their presence across the unique variable names, and across projects. Out of 67,286 unique variable names in the dataset, we find that 44.05% have at least one space: 30.95% have one space, 9.91% have

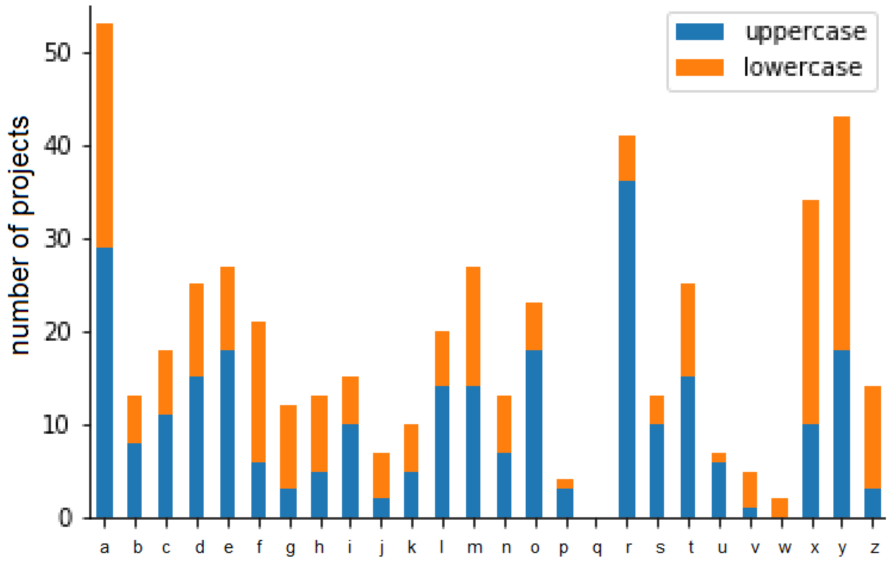


Figure 4.12. A histogram of single-letter procedures occurrences in Scratch projects

two and 2.15% have three space characters. For the usage of space character per project, we count the maximum count of a space character in the project's variable names. Out of 69,045 projects which include variables, we find 34% include variables names with at least one space character. As Fig. 4.14 shows, variable names with one space character are the most prevalent. We conclude that Scratch programmers prefer natural language naming of a variable: the use of a space character in variable names is common to some extent for many users, indicated by the project count, and for many used variable names.

Use of Numeric Variable Names

In addition to spaces in variable names, Scratch supports the use of numbers and even floating point numbers as variables. We found 718 projects using integer variable names and 19 with floating point names. While their use is rare, we manually examined some projects and numbers are used in interesting and clever ways. The most popular numeric values used as variable names include small natural numbers and 360 likely to represent 360 (cf. Fig. 4.16).

There seem to be two main uses of numeric variable names. First, some variables with numeric names represent constants (cf. Fig. 4.15). This seems to indicate the Scratch programmers prefer to drag in a constant rather than repeatedly type it.

A second use is the use of integer variables as simple list structures. For example, one of the projects we analyzed is a tic-tac-toe game. In that project, the programmer defined nine variables named 1 to 9. Each variable represents one of the nine boxes. Scratch supports lists, so the user here could have also used a list of 9 items, however, they did not. Maybe

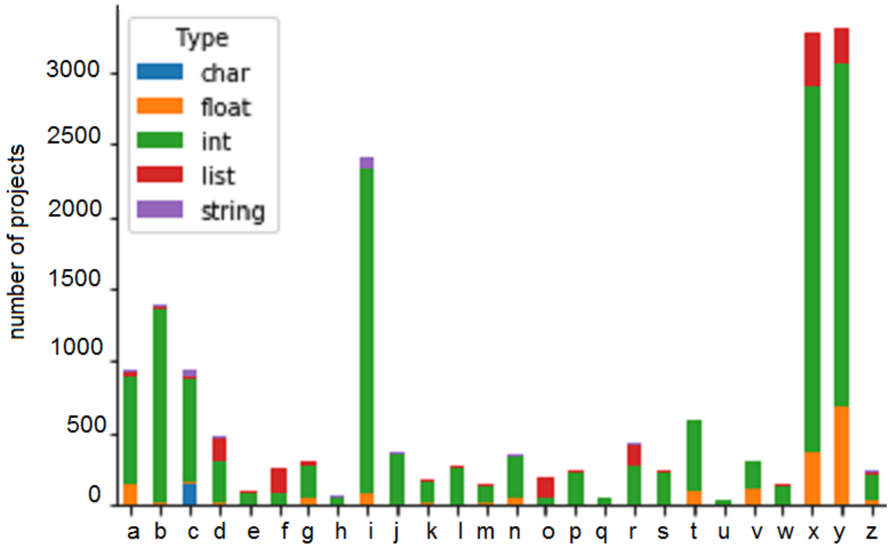


Figure 4.13. Inferred types for variables of one letter

because they were not familiar with the concept of lists, or maybe they thought this would be easier for the user to memorize the game logic.

USE OF TEXTUAL LABELS BETWEEN PARAMETERS

Scratch is influenced by the SmallTalk family of languages which is visible from the fact that Scratch allows users to insert textual labels in between parameters in order to make procedures more readable, as shown in Fig. 4.5. This practice seems particularly idiomatic to Scratch, since built-in Scratch blocks use a similar syntax, e.g., in the “say ... for ... seconds" block. In total 4,414 projects use textual labels accounting for 25% of projects with procedures, and 1.77% of all projects in the dataset. Their use is relatively uncommon, however, we do find some interesting patterns. Fig. 4.17 shows the most commonly used labels. Here we see some patterns common in textual languages, like the use of labels for the names of the parameters ‘x:’ and ‘y:’, and the use of a closing bracket. We suspect the lack of the opening bracket (to be caused by the fact that it would be included in the procedure name. We also see the use of ‘:’ at the end of many patterns, which could come from the users being inspired by Scratch default blocks, which use the colon as shown in Fig. 4.9. Finally, the use of the space (char-space in Fig. 4.17) is interesting, since Scratch already leaves some room between the parameters, also when a space is not used.

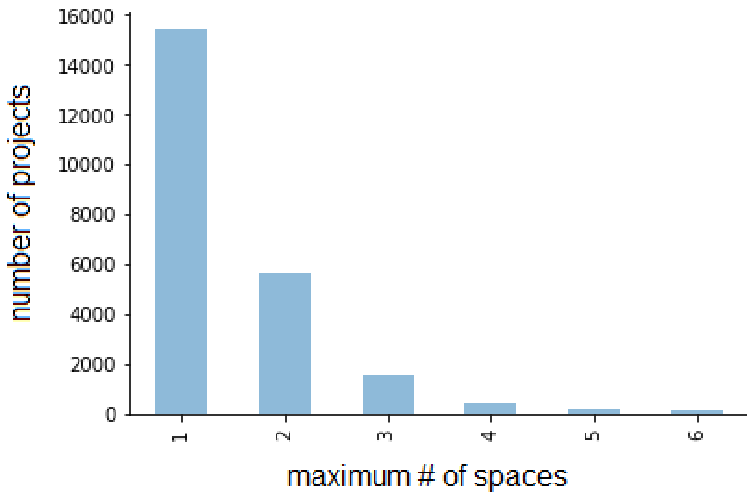
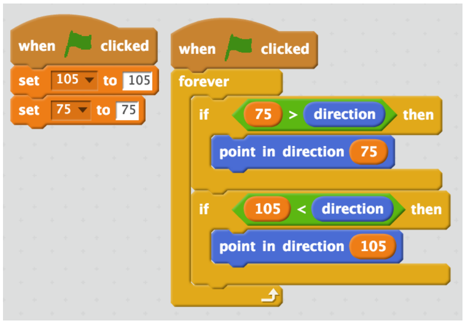


Figure 4.14. Number of spaces in variable names


```
when [flag] clicked
set 105 to 105
set 75 to 75

when [flag] clicked
forever
    if 75 > direction then
        point in direction 75
    if 105 < direction then
        point in direction 105
```

Figure 4.15. Numeric variable used as a constant

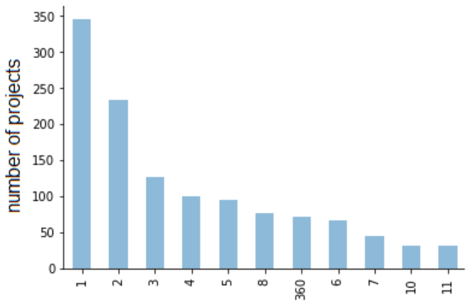


Figure 4.16. The most popular numeric values used as variables

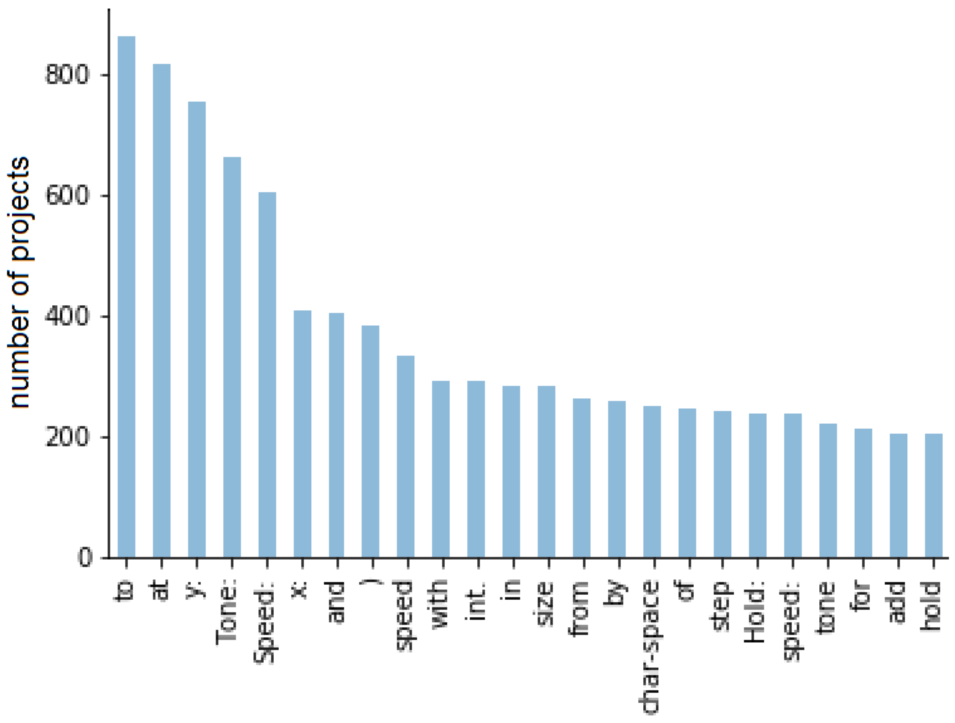


Figure 4.17. The most used textual labels in between parameters of procedures

4.6 Threats to Validity

As any empirical study our work is subject to series of threats to validity. Construct validity of our study might be threatened by the operationalization of the notion of a type. Due to the lack of explicit type declaration in Scratch our approach inferred the type by inspecting the value assigned to the variable. To be representative for a project, our method counted the different types encountered for the same variable within a project. For example if the variable “score" was set to “empty" at first, and then set to 0 in a different script, we count two types for the variable: one string and one integer. The same can be applied in textual languages with less strict type systems such as PHP. While the method is simple, it assures that data types used by Scratch programmers are represented in the data, and forms a good proxy to compare the results to the perceptions of professional developers. It is possible that these data types do not reflect the perception of users precisely. This is why this threat will be addressed in future work by surveying students in our running Scratch MOOC.

Another threat to the construct validity is our decision of what is a variable name in Scratch. As explained above for the sake of replication we choose to include the names of variables created specifically through “Make a Variable" button in Scratch UI.

One threat to the internal validity of our study comes from the use of non-Latin characters in identifiers. These characters are encoded as series of question marks, a limitation we inherit from the original dataset. Since one non-Latin character can be encoded as several question marks, the variable/procedure names length reported would overestimate the actual length. However, the impact of these identifiers is limited as the majority of Scratch programmers are from countries that use Latin alphabet[9]. To validate this, we manually analyzed the variable names extracted and found that merely 450 variables have non-Latin alphabetic characters, less than 0.67% of the total 67,287 unique variable names. Therefore, we did not exclude names with non-Latin characters from the analysis.

Finally, a threat to the external validity concerns the generalization of the study results. We argue that we use a large dataset which comprises around 1% of 23 million currently shared Scratch projects. It could be that the dataset does not reflect the users' trend for identifier naming. However, the Scratch community represents novice programmers with younger age. For these users, it is difficult to imagine they have an established trend in naming without prior education.

4.7 Conclusion

In this chapter, we study naming patterns for variables and procedures in the Scratch programming language, a block-based programming language aimed at novice programmers. We use a previously released dataset consisting of 250,000 Scratch programs.

Our analysis shows that Scratch programmers most often use variable names between 4 and 10 characters in length, while procedure names in Scratch tend to have longer names than the variables. For the single-letter variables, the most commonly used names are *x*, *y* and *i*. When compared to the other programming languages, Scratch variable length distribution, and the usage of single uppercase letters seems to be similar to the pattern found in Perl, while for the lowercase letters—to the pattern found in Java. Spaces in variable names, a feature relatively unique to Scratch, are used in 34% of projects in which variables

[9]`https://scratch.mit.edu/statistics/`

can be found. The usage of textual string between parameters appears as not so common, however textual patterns used imply an inference from textual languages by using brackets for example.

The chapter makes the following contributions:

- A detailed analysis of variables name length and single-letter variables in particular, replicating [24] on the Scratch programming language
- An analysis of procedure names in Scratch
- An analysis of naming patterns unique to Scratch, including spaces in variable names, numeric variable names and textual labels in procedures.

Studies concerning the difficulties novice programmers have when transitioning to textual programming languages highlight that handling identifier naming is one of the fundamental challenges faced by these learners, as part of a broader spelling challenge. We believe the naming patterns found in our study support the claim that challenges will be faced by Scratch programmers when transitioning to mainstream textual programming languages. Those languages restrict the use of spaces in identifiers and more often divert into short and single-letter names—tendencies opposite to the naming preferences in Scratch.

This study gives rise to a number of directions for future work. Firstly, Beniamini et al. [24] included a survey in which they ask developers to predict the type of a (one letter) variable. It could be interesting to ask a similar question to Scratch programmers for common variable names. Furthermore, a detailed study into the readability of variable names with and without spaces, and procedures with and without labels would help us to create naming guidelines for Scratch. Finally, one additional area to explore is the relation between name length of variables with the level of computational thinking [153, 187] of the Scratch programmer in addition to other demographic factors like gender, age and language.

5 Programming Education to Preschoolers

In recent years, there has been a rise in methods and tools dedicated to programming education for children of primary school age. In this chapter, we present our experience of providing five programming sessions to a group of eleven children between four and six years old. Our sessions follow problem-solving and game-playing themes and feature two newly-developed tools: the unplugged Robot Turtles, and the robotic Ozobot. The activities embed programming concepts such as the order of operations, symbolic representations, and functional abstraction. The observations show that children understand and apply concepts such as sorting, sequential operations, and functional abstraction. However, children struggle with giving directions to the object which highlights a spatial awareness limitation. Finally, we link the observations to Piaget's theory and his limitations to thoughts for children in this age. We find that some of Piaget's limitations such as egocentrism can explain few of the observed behaviors. However, other limitations contradict our observations such as irreversibility and transductive reasoning.[1]

[1]This chapter is based on our paper: Programming Education to Preschoolers: Reflections and Observations from a Field Study (Alaaeddin Swidan and Felienne Hermans) which appears in the 28th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2017

5.1 INTRODUCTION

Computational thinking education for children has been developing for decades [154, 165]. The aim is to make computer programming, a core aspect of computational thinking [7], widespread and more accessible. Many programming systems and languages are developed primarily for children as young as four, in schools and kindergartens [155]. The majority of these are computer aided [118]. For older children, Scratch has recently become a favored platform [97, 148]. Unplugged techniques, i.e., away from computers has also evolved [2]. With these tools in place, the focus is still to a great extent on developing more systems and tools for children, and to incorporate them into school curriculum [81]. However, few studies investigate children's cognitive and social reactions to these educational tools [155].

To that end, we provide programming education sessions to eleven children between four and six years of age at a primary school in the Netherlands. We use both unplugged and robotics materials to teach concepts such as the order of operations, symbolic representation, and functional abstraction. By guiding the children throughout these sessions, we aim at observing their reactions while using these educational tools and what difficulties they face in completing the tasks. We then compare our observations to Piaget's theory [173], one of the dominant theories in the psychology of educational development [149]. Results show the children's ability to perform sorting and classification tasks, to order the operations sequentially and to use a basic level of abstraction. On the other hand, children have difficulties with the spatial awareness needed to give directions to objects within the games. For the teacher, it is recommended to establish an active learning environment with a high level of engagement with the children. Finally, we find that some of Piaget's limitations to thought, such as egocentrism, help in explaining some of the observations. In other cases there are opposing observations to other limitations such as irreversibility and transductive reasoning.

5.2 BACKGROUND: PRESCHOOLERS EDUCATION

In this section, we provide an overview of constructivism, one of the most established theories in educational and development psychology. We follow by highlighting a few aspects of Piaget's theory as a major influencer in constructivism.

5.2.1 CONSTRUCTIVISM

In the psychology of learning, constructivism is one of the dominant theories [149, 155]. Derived from the word construct, constructivists assert that students and learners acquire new knowledge by actively processing sensory data against their previous knowledge. Ben-Ari [22] provides a comprehensive revision to constructivism and its application in education, with a focus on computer science. In summary, constructivism promotes an active learning environment combined with exploration and discovery activities. The role of the teacher is important to guide this process by designing the activities, and assisting the students especially when conflicts occur between new data and previous knowledge. In his revision, Ben-Ari provides examples showing that constructivism is applied in mathematics and physics education, but fewer efforts in CS education. Nevertheless, constructivist approaches are well-suited for programming education, especially for the introductory levels. For

[2] http://csunplugged.org/

these novice learners, researchers additionally recommend using games, diagrams and visualization to facilitate developing CS concepts [110, 223].

5.2.2 Piaget's theory

There are a variety of theories under the umbrella of constructivism [155]. We highlight Piaget's theory [173] as it has one of the major influences in constructivism. His theory focuses on the cognitive development of children. Piaget theory proposes four stages of cognitive development: sensori-motor, pre-operational, concrete operational and formal operational [173]. Moreover, new information directed towards the child is either assimilated according to the existing cognitive models or accommodated by building a new cognitive construct [147]. A state of equilibrium occurs in between these two processes where no new knowledge is acquired. Even though the child, through personal experience and social relations, is responsible for these mental processes, the environment is still recognized as a playing factor in Piaget's theory [155]. The teacher as part of the environment is the motivator to the learning process by creating situations which cause the child to be in a disequilibrium state, and allow new cognitive constructs to be developed. The teacher's role expands to provide helping information, asking questions and making comparisons for example. In this chapter, we investigate children in the preoperational stage of development, which Piaget defines between two and seven years. Children in this stage are expected to develop language, complete operations, solve one-step logic problems [147]. More complex logical thinking starts to develop only after the age of seven. During the preoperational stage, children suffer from many limitations to acquiring knowledge, which Piaget calls *"limitations to thoughts"* [155]. Table 5.1 summarizes these limitations in the preoperational stage.

Table 5.1. Piaget's limitations to thought in the preoperational stage of development. The first three columns are taken from [155]

Limitation	Description	Example	Category
Centration	The child focuses on a single aspect of the situation, disregarding any other.	John cries when his father gives him a biscuit broken in half. Since each half is smaller than the full biscuit, John thinks he's getting less.	**Agree**
Irreversibility	The child is unable to realize that an operation or action can be reversed.	John doesn't realize that both halves of the biscuit can be joined to make a full biscuit.	**Disagree**
Static thinking	The child is unable to realize the meaning of state transformations	In a conservation task, John fails to realize that the shape transformation of a liquid (from a glass to another) doesn't change the quantity.	**Not observed**
Transductive reasoning	The child doesn't employ either deduction or induction; going instead from one particular aspect to another, seeing a cause where there is none.	"I had ill thoughts about my brother.My brother got ill. So, I made him get ill." Or: "I misbehaved, so mum and dad divorced."	**Disagree**
Egocentrism	The child assumes that everyone thinks like he/she does.	Mary picks up a game and tells her mum, "This is your favorite." She's assuming that her mother likes the game as much as she does.	**Agree**
Animistic thinking	The child sees life in inanimate objects.	Mary thinks the clouds are alive because they're moving.	**Agree**
Inability to distinguish appearances from reality	The child mistakes appearances for reality.	John thinks that a sponge made to look like a rock is indeed a rock.	**Not observed**

5.3 STUDY GOAL

In this chapter, we describe our efforts to teach programming concepts to children between four and six years old. Our study aims first at exploring how children in this early age react to problem-solving activities in a programming context. Secondly, we target to observe difficulties that limit the children's ability to complete an activity or grasp a particular concept. These difficulties might be related to the internal mental structures already developed in children or to the complexity of the concepts being introduced. Finally, we want to compare the difficulties we observe to Piaget's limitations of thought. In summary, we want to answer questions such as:

- How do children react during these activities? For example: to what extent are they motivated and engaged? Do they enjoy the materials used?
- What limitations to thought from Piaget's theory on preoperational children still apply in activities related to programming education?
- What variations in performance are observed in concerning age difference?

5.4 SETUP

To achieve the goals mentioned in the previous section, we provide five sessions to eleven children aged between four and six (five girls and six boys). The sessions include problem-solving activities in a game-playing theme. Our approach is influenced by the core principles of the constructivism: an active learning environment, with a guided exploration and immediate feedback. We follow an observational methodology where two to three supervisors guide and monitor the children's activities in each session. After the sessions, we discuss the observations and write the ones that are agreed upon by two supervisors. The sessions are held in a Dutch primary school. The children are part of an after-school club, and activities are performed in their daily classroom, a familiar and friendly environment. The children work in groups of two or three. When arranging the groups, we make different combinations in each session; sometimes based on gender or age differences and in other times purely random.

5.5 MATERIALS

We use both unplugged and robotic tools, in a game playing setup. Previous research show the positive effect of game playing activities in the education of computational thinking and programming [223]. In addition to the constant problem-solving theme, the exercises embed a variety of programming concepts, such as the order of operations, the symbolic representation, and functional abstraction. See Table 5.2 for the full list of the exercises, materials and their associated programming concepts. Following is a description of the tools we use.

5.5.1 SORTING AND CLASSIFICATION

This is the first session with the children. It consist of light activities because we aim at introducing ourselves to the children and getting to know each other. By doing this, we create a friendly environment where children feel free to act upon and express their thoughts. The activities employ papers with printed animal pictures, we then ask the children to perform

Table 5.2. Summary of exercises performed during the sessions showing the programming concepts which the game and the exercises collectively serve

Session Order	Programming Environment	Exercise	Overview	Concepts
1	Printed animal pictures	Arranging cards based on some criteria	Sort the animals based on size and classify them based on their habitation.	Data sorting and classification
2 and 3	Robot Turtle	1. An Empty maze with diamonds in the center	Give directions to the turtle in relation to the diamond	Problem analysis Symbolic representations Order/sequence of operations Abstraction (functional) Fault isolation and debugging Team working (pair programming)
		2. A maze with obstacles	Implement a workaround based on the obstacle's type	
		3. A maze with obstacles #2: Full path constructed	Without moving the turtle, the child need to complete the whole path.	
		4. A maze without obstacles, only with diamonds, with the need to use the frog card	Use the frog card to help the frog to reach the diamond faster	
4 and 5	Ozobot	Explore with the Ozobot	Explore the basic concepts of the Ozobot: sensor-based autonomy following a line, reflections of colors, and the behavior at line branches	Problem analysis Symbol interpretation What-if analysis Alternative branch concept (visual) Team working (pair programming)
		Use the color codes in two tracks, so that the Ozobot can get from home to school and from home to shop	Choose the appropriate color codes	

various sorting and classification tasks. For example, to sort animals based on size or color. Another example is to classify animals based on the habitation such as farm, wild and sea animals.

5.5.2 ROBOT TURTLES

Robot Turtles is a board game that *"teaches programming to kids"*[3]. The board game, shown in Figure 5.1, is introduced by Dan Shapiro, a software engineer. According to the game's website, it aims at allowing preschoolers to learn *"the fundamentals of programming while they are playing"*. The game resembles a simple visual programming language, with the child ordering the cards to direct the turtle through obstacles until it reaches the diamond. The adult demonstrates the computer processor and moves the turtle depending on the order of the cards. The game, as a result, features principles of simplicity, visualization, simulation and autonomy, which are recommended when considering children and novice programmers [75, 110]. In addition to the basic direction cards, the game has some special cards. The **laser card** melts down the ice obstacle, the **frog card** abstracts a function that helps the turtle perform repeatable actions, and the **bug card** by which a player breaks the

[3]http://www.robotturtles.com/

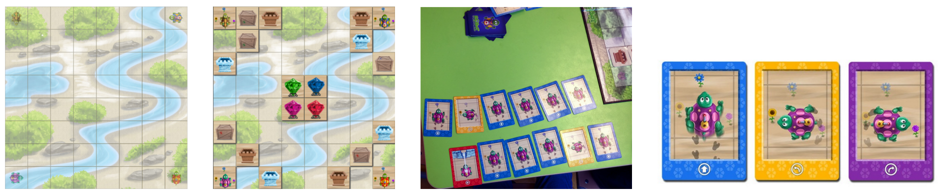


Figure 5.1. The Robot Turtle game: (a) an empty board (b) a board with a maze (c) one solution to a Robot Turtle exercise in the classroom (d) the basic movement cards

execution and notifies the adult that a problem exists within the sequence of cards. We notice that the Robot Turtles game focuses on programming concepts. In particular, the order of operations (through the sequence of cards), functional abstraction (through the frog card), debugging (through the bug card) and code refactoring/optimization (through using shorter paths i.e., less amount of cards). We use the Robot Turtles in two sessions, during which we gradually introduce more game features and programming concepts. We additionally increase the difficulty of the maze the turtle has to solve. In total, we have four exercises where the children perform the tasks on two copies of the game. See Table 5.2 row 2 for an overview of the exercises.

We introduce the game to the whole group by describing the basic cards with their associated actions. We then work closely with the children to repeat some information. We perform the same collective description for each new exercise; describing the goal and introducing new cards. Within the same exercise, the difficulty level was unified: children have the same mazes. We do not know in advance what limitations exist per child; in fact, this is an issue we want to observe.

5.5.3 OZOBOT

The Ozobot [4] is *"a miniature smart robot that can follow lines or roam around freely, detect colors, and can also be programmed"* [5]. The Ozobot can be programmed through a set of color codes, to perform a variety of actions: speed, direction, and funny moves. The Ozobot is designed to randomly choose directions when facing a line branch or a junction. According to the Ozobot website, the Ozobot empowers the STEM (Science, Technology, Engineering, and Mathematics) education, and can be used to teach subjects such as programming. Regarding age, the Ozobot is designed for all levels of the elementary school. For younger children, it is not advised without an accompanying adult, primarily because of its small size. Similar to other studies which used robotics in programming education [116, 141], the Ozobot can be used to develop certain programming concepts. These concepts include the analytical problem solving, sequential programming, and understanding of computer autonomy. See Table 5.2 row 3 for the concepts associated with the Ozobot. We employ five Ozobots in two sessions. Each Ozobot is controlled by two to three children. We use the Ozobot Basic Training lessons and resources provided by Ozobot [6]. The first session

[4]http://ozobot.com/
[5]https://education.microsoft.com/Story/Lesson?token=qgSYB
[6]http://portal.ozobot.com/lessons

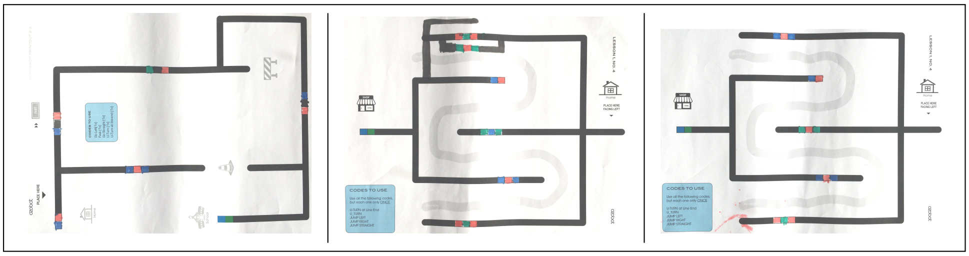


Figure 5.2. Ozobot exercises performed by children throughout the sessions

was more exploratory to the features of the Ozobot. Children are expected to learn more about its tracking the lines and colors, but also about the randomness and indeterministic decision at a junction. They also learn about the color codes, and explore possibilities in controlling movement, speed, and directions. In the second session, the children are asked to solve two exercises which require choosing specific color codes in order to get the Ozobot from a source to a destination. Figure 5.2 shows three solved exercises stemmed by children in our sessions.

5.6 OBSERVATIONS

In this section, we list six common behaviors of the children during the sessions categorized as observations. For a behavior to qualify into an observation, it needs to be demonstrated by at least two children and observed by two of the supervisors.

Observation 1: Children suffer when giving directions especially with different viewpoints

One of the most recurrent mistakes in both games is the choice of a wrong direction to the turtle or the Ozobot. Typically, the direction decision involves two objects: where the child is looking initially, and where to turn/look finally. In these games, however, the moving objects are different than the child; they are the turtle and the Ozobot. Figure 5.3 shows two different boards where the turtle has a similar and a different viewpoint. The children in many cases consider their position in relation to the diamond, rather than the turtle. This behavior is associated with a computational cognitive skill known as spatial reasoning, and it defines *"the ability to recognize and view figures, and the ability to rotate or follow the movements of figures"* [7]. In their study, Ambrosio *et al.* considered it as one of *"the crucial dimensions for introductory programming"*.

Observation 2: Children get derailed from a task by fantasies they connect to

Children find themselves immediately drawn to joys and fantasies familiar to them found in these games. For example, quite a few children, mostly boys, felt very enthusiastic when we introduced the laser card in the robot turtles exercises. They started to point it to each other making a sound imitating the laser hitting other objects. One boy later in the exercise asked if he can put the laser card before reaching the ice obstacle because the laser can transfer and melt from a distance. In the Ozobot exercises, it was more tempting for children to get away from the original task because of the many temptations involved: robots, funny and fast moves, and coloring. Almost all of the children were amazed when we introduced the Ozobot. Some kept the amusement for the overall two sessions. They enjoyed the idea of

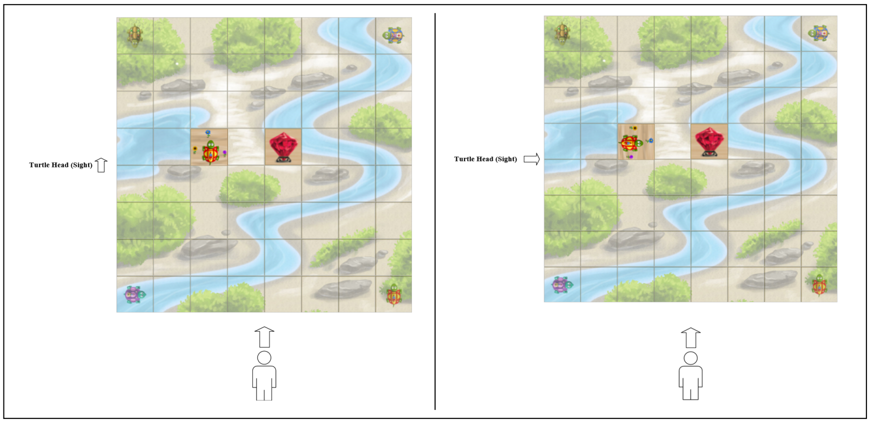


Figure 5.3. Spatial reasoning arises when a child has a different view than the controlled object

having a moving robot, and tried to control it by voice commands like *“move”* or *“go left”* and ordering it to be colored. Some children got their hands on the coloring markers quickly and immediately started coloring on their own. When it comes to color codes, many children (mostly boys) wanted the Ozobot to go fast and make funny moves all the time regardless of the problem in question.

Observation 3: Immediate execution helps children better grasp a game's rules and identify faults

The immediate feedback of the computer (the adult in Robot Turtles and the Ozobot) was essential for children to understand the symbolic meaning behind the games’ material. In the case of Robot Turtles, we simulated sounds, when the turtle moves, to send confirming messages to the children about the meaning of some actions and obstacles. For example, the turtle can push a box, but not a wall. Even though we told the children about the meaning verbally, some children made mistakes and tried to move forward through a wall. By making a special sound and visual appearance of difficulty to move a wall, children made fewer faults in the next tasks. The exercise of the full path creation at once showed that the performance, when the turtle starts to move, motivates the child to anticipate the fault in the card ahead. As soon as the turtle moves, we can see that some of the children detect the error just before the adult executes the subsequent action. They then try to intervene and fix it with the correct card. Adding the right card, however, did not cause the children to adjust the sequence of cards which follow after the fault. They again waited for the simulation of the turtle and then react, and so on.

Observation 4: Children comprehend functional abstraction in Robot Turtles faster than expected

We introduced the concept of the frog card as a helper to the turtle, to move faster through jumping some ordered actions. We tried to be very clear in the explanation, giving them examples. We found the children to be reasonably fast in understanding the application of a frog (function) card in the main sequence. Having said so, we highlight some concerning observations in this area. All of the children chose to apply a single-type operation for the frog function, which is the moving forward card. They wanted the frog to move the turtle two or three steps ahead. In response, we provided thought-provoking questions after they completed the task like *"what other cards you could have used"*. Another concern is that some children placed the cards for the frog i.e., defined the function, but they did not use it within the main sequence of cards. When notified by the supervisor *"Oh I don't see a frog!"* the children eventually placed the card, only in one location in the main sequence. We provided them with additional suggestions, and only then few children got the idea that another place in the main sequence is possible for a second frog card call. Finally, we observed some variances in the speed of getting the idea and the quality of its application among the children. These differences relate primarily to the age group of four years old and will be discussed later in Section 5.7.

Observation 5: Adult-supported and active learning environment is a necessity
Limiting the role of the adult in such experiments to the activity designer, and imitator of knowledge does not suit the young children. These children have different interpretations to a single information outspoken by adults. They might not understand it or might accommodate it with the incomplete knowledge they already have. Thus it is important for adults to discuss individually or in small groups of the children what they think about a particular aspect of a task or a concept. We often did this by asking questions in the form of *"What do you think..."*, *"Why did the object behave in a specific way..."* and *"What if you did this action? what will happen?"* The environment of the classroom was an additional support to children: it is their daily place to learn and play, they feel very confident and relaxed moving around and using available resources. We saw little influence to learning from peers. Time was not sufficient for one group to be more knowledgeable than the other group. In one occasion a child who finished his robot turtle exercise tried to give his peer group the answer to their next card. Unfortunately this was the wrong direction card, and fortunately, the recipient boy thought for a second and chose to stick to his original card. The influence, in this case, would have been adverse to the solution of the problem. We suspect this is because the intervening child lacked the proper spatial awareness, and so he rushed with the wrong card.

Observation 6: Children often do not express their sense of difficulty or lack of understanding
Children will go directly into action after explaining the task. In one case only we saw a six years old girl approaching us saying *"what do you mean by this"* in referring to the final Ozobot exercise. This question came after explaining the task to all children. In another area, children seem to prematurely assess the difficulty of a game. In one case, a boy of six years old, when introduced with the Robot Turtle exercise in the second session shouted *"oh no, not this game again. It is so easy"*. However, this boy took the longest time to complete the exercise which followed. In fact, other children said the word easy so often when playing with the Robot Turtle, combined with their fingers naturally pointing the route the turtle should follow to reach the diamond. However, their route building procedure using cards usually did not come as smooth as the path pointing. We cannot tell the exact reason. Research shows that young children tend to prematurely evaluate the explanation of an issue based on incomplete aspects [223]. In our case, this means that their mind considers the pointing as the challenge, and then judge the easiness of the game based on it. It can be other things such as a mechanism to avoid embarrassment or showing off among peers. It can be that these children did not fully understand how to build the routes by cards, despite being

able mentally to point out the route.

5.7 Observation Analysis

In this section, we provide an analysis of the observations by looking back into Piaget's theory in general, and Piaget's limitations of thought for preoperational children in particular. We classify the limitations depending on whether it is spotted in one of the observations or not. Additionally, this classification includes other core aspects of Piaget's theory which we believe worth mentioning.

5.7.1 Piaget's theory in agreement with the observations

Egocentrism

Piaget's definition of egocentrism is not about improper social behavior. It primarily focuses on the child's ability to consider the perspectives of others [155]. He showed, using the three mountains experiment [149], that children fail to recognize the viewpoint of an object different than their own. The egocentrism of the children at this age is still an aspect that the educators need to consider. In our sessions, wrong direction choices were the most frequent mistakes. In the Robot Turtle, the child has extra support by colors and flower painting on the cards. Each move is associated with a color, as shown in Figure 5.1d: blue card is forward, the yellow card is left, and the purple card is right. Also, each card has three different flowers matching the color scheme mentioned above. Despite this kind of visual support, children still suffered in spatial reasoning. The problem is observed more when the child perspective is different than the perspective of the turtle or the Ozobot (see Figure 5.3). The child will choose the card that agrees with his position in relation to the goal (the diamond or the branch respectively).

Animistic Thinking

According to Piaget, the child in this age gives life to non-living objects [155]. Observation 2 includes behaviors in common with this limitation, as some fantasies the children brought during the sessions fit under this definition. However, this is not necessarily a limitation towards the cognitive development. In some cases, it is a part of the game theme of the activity, such as making the laser sound and playing with each other using the laser card. While in the case of giving voice orders to the Ozobot, it can be considered as a challenge to the child's ability to understand the nature of forces controlling the Ozobot movement, and then his ability to apply color codes in the process. It is worth noting that the Ozobot has some similarities with living objects, especially the moving. However, children are also affected by previous knowledge about robots. To handle distractions caused by this limitation, we suggest to allow free time playing with the materials, making it clear that afterward attention shall be given to the exercises.

Centration

Centration is defined as the preoperational child tendency to focus on a single aspect of the problem disregarding the other aspects. Observation 3 includes behaviors that can be explained by this definition. For example, the children were more engaged and reacted faster when the turtle was moving, as they introduced the next move's card quickly. On the other hand, building the full path required an extra step of imagination from the child. Most of the children committed more mistakes in the full path exercise than when the turtle moved step by step.

5.7.2 PIAGET'S LIMITATIONS IN DISAGREEMENT

IRREVERSIBILITY

As stated in Table 5.1, the irreversibility limitation in preoperational children describes the inability to understand that a particular operation or action is reversible. This kind of behavior was not observed during our sessions. We found the children were motivated to amend any fault immediately, sometimes during runtime. As mentioned in Observation 3, children started to put correct cards instead of the wrong ones when the turtle starts moving in an unintended direction. In Ozobot, less reversible actions were possible because of the permanent coloring. However, some children managed to create extra branches for the Ozobot in order to overcome one wrong color code, see Figure 5.2. Nevertheless, this behavior was limited and occurred following a hint from the supervisor.

TRANSDUCTIVE REASONING

Piaget defined it as the child's inability to deduce or induce causes and effects correctly (see Table 5.1). We found opposing observations to this limitation in our sessions. We note, however, the careful and precise explanations we provided to the children prior to making the reasoning. One example from the Ozobot is related to the branch decision. The ozobot would choose a direction randomly when no color code is provided ahead of the branch. First, we asked the children: *"Where do you think the ozobot goes on a branch?"* We received rushing contradicting answers *"always red"*, *"always left"* or *"always straight"*. We then let them explore with two branches where we ask them *"where do you think the Ozobot will go?"* before it moves. Children observed how the Ozobot is not predictable: sometimes they get it right by chance, and many other times they do not. After a few runs with these exercises, we asked the first question again. We got a strange moment of silence, followed by a four-year-old saying exactly what we wanted to hear *"We do not know!"*. As long as we do not tell the Ozobot, by color codes, where to go, we do not know its decision for sure. This is aligned with findings of [75, 79] who described this as the indeterminate problem. Their results concluded that children in this age should be able to distinguish between deterministic and indeterministic situations following some exercises.

ABSTRACTION, SORTING, AND CLASSIFICATION

Piaget theory believes that preoperational children are unable to develop sorting, classification and abstraction thinking until reaching the concrete operational and formal operational stages. Children in our case showed a high level of skill in sorting animals based on attributes as size, color, and habitation. While in the Robot Turtle's frog card, children grasped the concept of the helping card rather quickly. Before the exercise, we presented the frog card usage as a helper to the turtle and gave examples on how to use it. The application of the frog card into the main sequence, however, was limited. Only one type of movement card was used in building the functions, the forward card, and it was called in one place in the main sequence.

5.7.3 Age factor
Despite having a group who are very close in age, we noticed a variation between four and six-year-old children. From what we observed, this difference is primarily related to their focus span increasing with age. As a result, four-year-old children had more tendency to lose track of the task and get easily distracted by other joyful activities like drawing and coloring. Besides affecting their level of understanding the concepts presented, this sometimes negatively affected their teammates in the group.

5.7.4 Motivation
Overall the children were motivated and engaged in the activities. This motivation was shown during the free time given to them just before leaving. In the robot turtles session, they engaged in building a maze for each team trying to challenge each other. For the Ozobot they created by drawing and coloring their routes and problems for the Ozobot to overcome. Also, they expressed to their parents the wish to continue playing these games at home.

5.8 Conclusions
In this chapter, we report our experience to provide five programming lessons to children between four and six years old. We present six observations related to the children reaction and understanding of the programming concepts embedded in the activities. Among these observations, we highlight the children's ability to perform sorting and classification tasks, to order the operations sequentially and to use a basic level of functional abstraction. On the other hand, children had difficulties with spatial awareness needed to give directions to the objects within the games. We compared our observations to Piaget’s limitations to thought and showed the egocentrism helps in explaining the spatial awareness observations, while transductive reasoning and irreversibility were opposed by our observations.

6 Programming Misconceptions for School Age Students

Programming misconceptions have been a topic of interest in introductory programming education, with a focus on university level students. Nowadays, programming is increasingly taught to younger children in schools, sometimes as part of the curriculum. In this study we aim at exploring what misconceptions are held by younger, school-age children. To this end we design a multiple-choice questionnaire with Scratch programming exercises. The questions represent a selected set of 11 known misconceptions and relate to basic programming concepts. 145 participants aged 7 to 17 years, with an experience in programming, took part in the study. Our results show the top three common misconceptions are the difficulty of understanding the sequentiality of statements, that a variable holds one value at a time, and the interactivity of a program when user input is required. Holding a misconception is influenced by the mathematical effect of numbers, semantic meaning of identifiers and high expectations of what a computer can do. Other insights from the results show that older children answer more questions correctly, especially for the variable and control concepts. Children who program in Scratch only seem to have difficulties in answering the questions correctly compared to children who program in Scratch and another language. Our findings suggest that work should focus on identifying Scratch-induced misconceptions, and develop intervention methods to counter those misconceptions as early as possible. Finally, for children who start learning programming with Scratch, materials should be more concept-rich and include diverse exercises for each concept.[1]

[1]This chapter is based on our paper: Programming Misconceptions for School Students (Alaaeddin Swidan, Felienne Hermans and Marileen Smit) which appears in the proceedings of the 2018 ACM Conference on International Computing Education Research, ICER 2018.

6.1 Introduction

It is known from existing research that learning programming can be difficult [25, 32, 131]. One source of difficulties is holding programming misconceptions [32, 215], which affects performance in writing or understanding code. A programming misconception is having an incorrect understanding of a programming concept or a set of related concepts, typically affected by prior knowledge from domains other than programming such as mathematics and natural languages [208].

Studying programming misconceptions involves identifying their possible origins in order for both learners and educators to rectify relating concepts. Misconceptions have a harmful effect on the performance of students. The effect starts early [215] and may remain for a long time [204]. They have been found to cause failure in introductory programming courses and, in the long run, even cause students to drop out of programming education [138]. Previous studies focused nevertheless on introductory courses in universities. Nowadays, CS education and programming is increasingly introduced to younger students in primary and secondary schools [15, 107]. Many countries have already integrated programming activities into their school curriculum [97]. Moreover, new programming languages and programming environments are implemented especially for younger children. An example is Scratch[2], a block-based language which is developed by MIT with the aim of teaching children how to program. While CS education is moving down to schools, little is known on whether children develop certain misconceptions at this stage. In this study we aim at exploring the programming misconceptions held by school-age children. We developed a multiple-choice questionnaire containing 11 questions representing a selected set of programming misconceptions known from previous research [209]. In total, 145 children aged between 7 and 17 participated in our study. The participants, who were required to have an experience in programming, additionally provided reasoning for their answers in open-ended texts. From the data collected in this survey, we aim to answer the following research questions:

RQ1 Which programming misconceptions are the most common among Scratch novice programmers?

RQ2 How do children holding those misconceptions explain their answers? How do their explanations differ from the ones of children understanding the concept correctly?

RQ3 How do age and previous programming knowledge affect the holding of a misconception or the correct understanding?

6.2 Background

Research defines a programming misconception as an incorrect understanding of a concept or a set of concepts, which leads to making mistakes in writing or reading programs [208]. Misconceptions can be related to basic, yet fundamental, programming concepts, not only to advanced concepts. Apart from syntactic mistakes, there seems an agreement among researchers on particular concepts being difficult for learners. Those language-independent concepts include variables, loops, and conditional statements [32, 85, 131, 139, 204]. An example of a programming misconception: the belief that a variable can hold multiple

[2]https://scratch.mit.edu/

values, or the belief that a variable's assignment goes the opposite direction [32, 209]. In object-oriented languages, common difficulties are related to the scope and visibility of variables, modularization and decomposition and inheritance [88, 138].

Some research has focused on understanding where a misconception originates from. A programming misconception does not mean that the learner has a complete lack of knowledge, rather it indicates partial but self-interpreted knowledge which comes from domains other than programming [32]. Some of the known origins include the use of particular analogies in explaining a concept, the ambiguous and double meanings of some of the programming keywords in English as a natural language, and mathematics [32, 179, 206]. Du Boulay [32] introduced what he called the *"notional machine"* as one origin of programming misconceptions. The notional machine refers to the general properties that a student assumes of the machine executing their code. Having an incorrect understanding of the notional machine of a programming language is believed to be the cause of many misconceptions [139, 208]. For example errors were found as a result of what Pea [168] called the *"superbug"* which is the assumption that *"there is a hidden mind somewhere in the programming language that has intelligent, interpretive powers"* [168, p. 32], or *"forgetting about alternative branches because they are too obvious to merit consideration"* [210, p. 6].

Finally, the variety of misconceptions make it difficult for educators to take them fully into account. In this regard, Sorva [209] provides a comprehensive list of programming misconceptions collected from various studies [32, 74, 138, 206, 208]. In this study, we use Sorva's list as the starting point to investigate Scratch misconceptions in school students.

6.3 SETUP

The goal of our study is to explore the common misconceptions among school-age children in Scratch. To this end, we perform a questionnaire-based study. Participants are given a set of multiple choice questions; each question is a programming exercise in Scratch which tests the holding of a known misconception. Questionnaires have been used to assess the holding of programming misconceptions in previous research [138, 205]. Participants in most cases need to predict the outcome of the script to choose an answer. Figure 6.2 shows an example question from our study. We provide the full questionnaire (English version) here[3]. In the following sections we describe the study setup in detail.

6.3.1 PARTICIPANTS

In total 145 children took part in the study. Figure 6.1 shows the distribution of the participants over the reported age, ranging between 7 and 17 years. Among the participants, 102 (70.3%) are boys, and 42 (29%) are girls; one participant did not specify gender.

6.3.2 STUDY ENVIRONMENT

We ran this experiment at NEMO science museum[4] in Amsterdam. Children visiting the museum were asked to join an experiment on programming but received no further information on what the experiment would measure. Participants did not get financial compensation for participation, but did receive a certificate for their efforts. The experiment

[3]http://cli.re/g1zyQM

[4]https://www.nemosciencemuseum.nl/en/

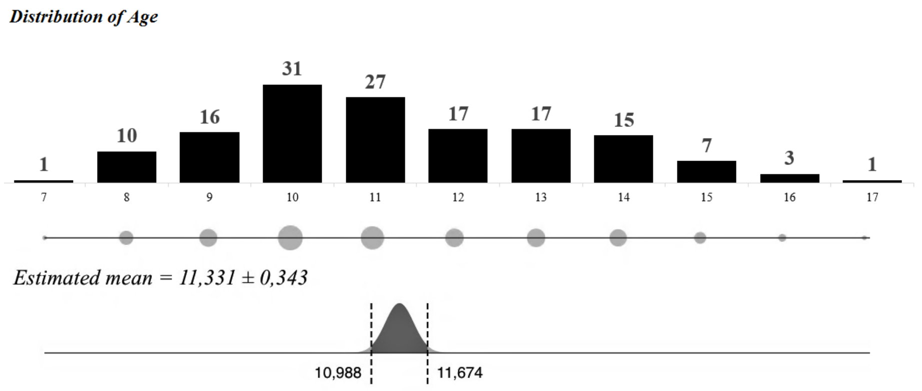


Figure 6.1. Age distribution of the participants. 70.3% of them are primary school students (12 years old and younger)

was assigned a 25-minutes window per child and was run in a separate room in the museum that seated 8 children at a time. In total we spent 14 days at the museum, running the experiment for about 5 hours each day. During the experiment each participant had to fill-in answers to our web-based questionnaire in which the questions were put in static images. The children had access to the machine and the Internet, but we did not observe any participant open other applications or pages than the questionnaire. The number of participants reported in this study is a subset of the total children who visited our booth, we filtered out participants who did not have programming experiences or who indicated guessing the answers. Asking children in such a setup allowed us to study the holding of misconceptions on a sample that is less-dependent on specific teaching methodologies compared to an experiment run in a school.

6.3.3 MISCONCEPTIONS SELECTION

Sorva provides a comprehensive list of 162 programming misconceptions known from the literature [209]. We used the list as our starting point to investigate Scratch misconceptions. We followed a two-step approach to achieve that. **First, we selected the most common misconceptions from the list**. A misconception is common if indicated by at least two separate research works. This step reduced the list to 17 programming misconceptions. **Second, since we study Scratch misconceptions, we filtered out the misconceptions that do not fit Scratch as a language**. For example, we eliminated two misconceptions that concern the use of a loop control variable inside the loop's body. In Scratch, loops use a static value and no variable is necessary to iterate through the body. This resulted in 11 misconceptions, shown in Table 6.1.

Table 6.1. Programming misconceptions included in our study from [209].

Code[5]	Description	Prerequisite Concept(s)[6]	Questions Description (Pseudo-code)	Misconception Choices

M9	A variable can hold multiple values at a time	Variables	Set [X] to 10;
			Set [X] to 20;
			Say [X];
				Gobo says 10, 20

M11	Primitive assignments work in opposite direction	Variables	Set [a] to 10;
			Set [b] to 20;
			Set [a] to [b];
				a=20, b=0
				a=20, b=10

M14	A variable is a pairing of a name to a changeable value. It is not stored inside the computer	Variables	We create a variable called message. Where is the variable stored?	Variable is not stored anywhere Variable is only visible on the screen

M15	Primitive assignments store equations or unresolved expressions	Variables	Set [X] to 1,
			Set [counter] to [X+1];
			Say [counter];
				Gobo says X+1

M17	Natural-language semantics of variable names affects which value gets assigned to which variable	Variables	Set [big] to 1;
			Set [small] to 100;
			Set [big] to [small];
				big=100 small=1
				big=100 small=0

M23	Difficulties in understanding the sequentiality of statements	Variables	Set [number1] to 0;
			Set [number2] to 0;
			Set [total] to [number1] + [number2];
			Set [number1] to 4;
			Set [number2] to 2;
			Say [total];
				Gobo says 6

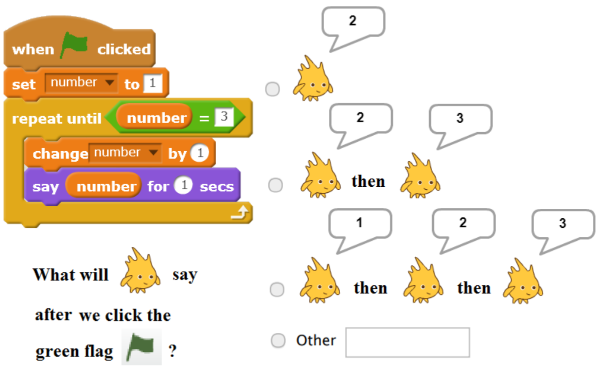
M26	A false condition ends program if no else branch exists	IF ELSE	IF touching the color [black] then {
			Say Auw!!;};
			Say I am moving;
				Gobo does not say anything

M30	Adjacent code executes within loop	Variables
		Loops
			Set [counter] to 0;
			Repeat 5 { Change [counter] by 1;};
			Say [counter];
				Gobo says 1 till 5

M31	Control goes back to start when condition is false	IF ELSE	Say I am moving;
			IF touching the color [black] then {
			Say Ouch!!; };
			Say Done!!;
				Gobo says I am moving

M33	Loops terminate as soon as condition changes to false	Variables
		Loops
			Set [number] to 1;
			Repeat until [number]=3 {
			Change [number] by 1;
			Say [number]; };
				Gobo says 2

M150	Difficulties understanding the effect of input calls on execution	None	Ask [How old are you?] and Wait;
			Say Nice! I will move now;
			Move [10] Steps;
				Gobo says How old are you? and immediately Says Nice! I will move now and Moves 10 steps



```
when [flag] clicked
set number to 1
repeat until number = 3
    change number by 1
    say number for 1 secs
```

What will [Gobo] say after we click the green flag [flag] ?

Figure 6.2. Question and possible answers for M33

6.3.4 QUESTIONS

Before presenting the misconception questions to the participants, we require the participant to answer five close-ended questions in the questionnaire. The answers to those questions indicate the participant's familiarity with these programming concepts: variables, IF statements and loops. We use the answers in an automatic branching logic so that we present a set of three to eleven multiple-choice misconception questions (See Table 6.1) that fit the knowledge of the participant. Each question is designed to elicit one of the well-known programming misconceptions. For this purpose we use programming problems similar to the ones suggested by Ma [138] for Java students. In our case, we design the question in Scratch both in English and Dutch. Scratch enables programming in one's native language, which eliminates the cognitive load for reading a foreign language for the local children and enables them to focus on the programming challenge [69, 215].

Figure 6.2 shows an example of one of the questions related to the misconception of a loop terminating as soon as the condition becomes false (M33). We ask the participants to predict the outcome of the program by asking *"What will Gobo say when we click the green flag?"*, where Gobo is one of the characters known in Scratch. The answers are categorized into *Holds_Correct* (Gobo says 2 then 3), *"Holds_Misconception"* (Gobo says 2), and *Other_Wrong* (Gobo says 1 then 2 then 3). An open-ended question follows so that the participant can explain the reasons behind the chosen answer.

We used the open-ended text to filter the results. The aim of this filtering is to eliminate answers for which children admitted that they either guessed the answer or lacked understanding of the question. We note that we initially received 1,306 answers from 178 participants. Due to the filtering process, 545 answers were eliminated. The number of participants included in the study went down to 145 since for some participants their whole answer set was eliminated.

[5]We add a prefix, M, to the original numbers assigned to the misconception in Sorva's list. This is for the sake of easy referencing in the chapter.
[6]This column indicates how we assign questions to participants based on the familiar concepts they report.

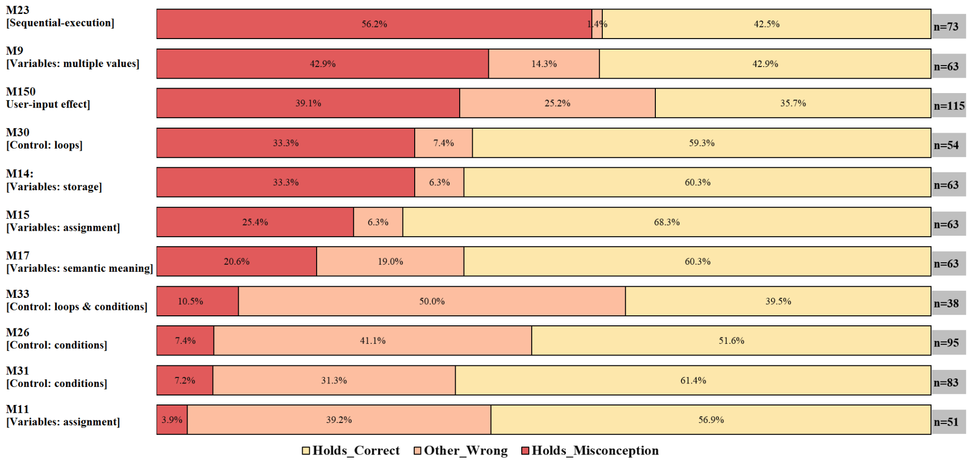
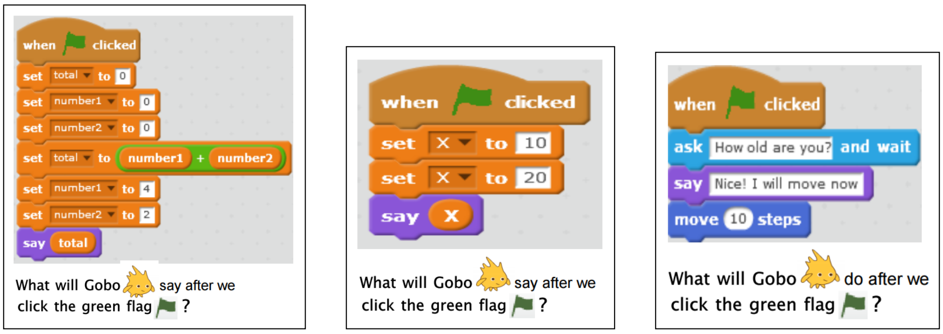


Figure 6.3. Misconceptions and their answer distribution, ordered from most to less common. M14, M26 in addition to the top 3 misconceptions are further discussed by exploring participants open-ended text.


```
when clicked
set total to 0
set number1 to 0
set number2 to 0
set total to number1 + number2
set number1 to 4
set number2 to 2
say total
```
What will Gobo say after we click the green flag ?

```
when clicked
set X to 10
set X to 20
say X
```
What will Gobo say after we click the green flag ?

```
when clicked
ask How old are you? and wait
say Nice! I will move now
move 10 steps
```
What will Gobo do after we click the green flag ?

Figure 6.4. The questions representing M23 (Left), M9 (Middle) and M150 (Right)

6.4 RESULTS

This section provides an overview of the answers to the study's research questions based on the questionnaire's data.

6.4.1 MOST COMMON MISCONCEPTIONS

[RQ1] Which programming misconceptions are the most common among Scratch novice programmers?

To answer RQ1, we analyze the answers for each misconception question. Figure 6.3 presents the percentage of participants who selected the misconception choice per question. The three most common misconceptions are related to different concepts: (i) the sequentiality of executing code (M23), (ii) the variable holding multiple values at a time (M9), and (iii) the human-computer interactivity and its effect on execution (M150). Moreover, we notice that among the least common misconceptions are the misconceptions related control statements:

Table 6.2. Most common words in the open-end text provided by children for the top three misconceptions

	M23: sequential-execution			**M9: variable's multiple values**			**M150: user-input effect**		
	Rank1	Rank2	Rank3	Rank1	Rank2	Rank3	Rank1	Rank2	Rank3
Holds Correct	before	numbers	values	then	set	variable	wait	answer	until
Holds Misconception	value	then	add	then	first	numbers, makes	answer	order	program
Other Wrong	Not applicable			then	say	picture	what	know	does

loops and conditions (M31, M33 and M26). In these cases, however, otherwise wrong choices are popular among participants, which indicates the general difficulty to understand those concepts.

6.4.2 INSIGHTS FROM CHILDREN'S EXPLANATIONS

[RQ2] How do children holding those misconceptions explain their answers? How do their explanations differ from the ones of children understanding the concept correctly?

First we quantitatively analyze the open-text provided by participants per their answer category for the top three common misconceptions (see Table 6.2). To highlight the thinking process of the children both holders of a misconception or the correct concept, we further explore the open-ended answers provided by participants for M14 and M26 in addition to the ones in Table 6.2. M14 is chosen because two aspects of the misconception are provided in the multiple choices. M26 is chosen because although a choice representing the misconception is not provided separately (See Section 6.5.4), participants still indicated holding the misconception through their provided open-ended text.

M23: Difficulties in understanding the sequentiality of statements
Participants with the misconception=56.2%, Correct=42.5%, n=73

Misconception: Participants show a focus on the mathematical operation itself, not on the sequence. This is shown in the the top three words as the the words include *"values"* and *"add"*. Additionally, we find explanations such as *"because if you add number 1(4) and number 2 (2) [then] it will say 6"* or simply *"Basic math man"*. Some participants assumed an automatic aspect of the operation: *"when you change values of these variables total value changes also"*, and *"4 + 2 = 6 the computer should calculate that for you"*.
Correct: Participants are able to identify the sequential nature of the code. In the most frequent words we find the word *"before"* which indicates an order. One participant, for example, explains: *"Because total is set to [no.1] + [no.2] so it equals 0. The variables changed after that are irrelevant"*. Another participant suggested a *"fix"* to the code: *"If the block set [total] to [number1]+[number2] was put lower then it would have worked"*.

M9: A variable can hold multiple values at a time/*"remembers"* old values
Participants with the misconception=42.9%, Correct=42.9%, n=63

Misconception: Most participants referred to the code in the question as their reason without extra highlights. The frequent words used include the words *"first"* and *"numbers"* which shows the attention these participants give to the old value of the variable and both

numbers used in the exercise respectively. However, one participant, despite choosing the misconception answer explains: *"I'm not sure, but if two [instances] of the same variable are used with different numbers, if possible, [the result] will give both numbers. Otherwise it would be 20 because that was the last change"*.
Correct: One of the most frequently used words is the word *"variable"*. This might indicate that participants have a basic understanding of what a variable is and thus they use the term more frequently. Moreover, we notice from the explanations of some of the participants referring to the last change made to the variable's value. Examples on this include: *"X= 20 is after X= 10 and the later one will overwrite the earlier one"*. However, one participant shows a full awareness of this aspect of variables: *"variable X is changed to 10, then 20, and a variable can only have one value"*.

M150: Difficulties understanding the effect of input function calls on execution
Participants with the misconception=39.1%, Correct=35.7%, n=115

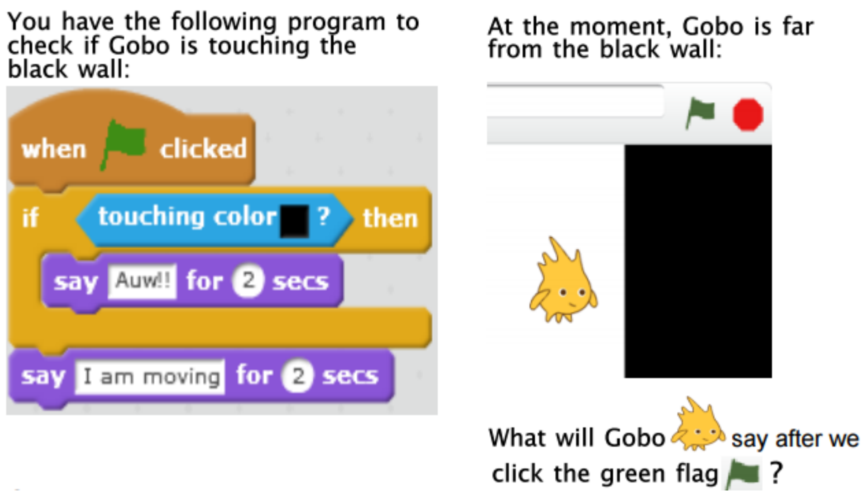
Misconception: The word *"order"* is the second most frequent mentioned by these participants, highlighting the importance they give to the sequential execution that respects the blocks' order. One participant explains for example: *"This is the order from up to bottom"* and another says: *"because [the answer] is in the correct order"*. Moreover we notice that some participants use the word *"answer"* identifying the question-answer nature of the program. However they still provided the wrong answer because of a variety of wrong assumptions such as that since it is not possible for the participant to fill an answer then the computer will continue. Another assumption is that the question is directed towards the computer, therefore the computer will answer and continue the execution: *"Gobo says i am .. years old and directly says ...[continues the next blocks]"*, and another participant: *"I think that in the game Gobo is asked how old he is, then he ...[continues the next blocks]"*.
Correct: We notice again two forms of reasoning when participants give the correct answer. One has a direct approach and relies on the word *"wait"* being present in the question and in the answer text, stating that the choice comes because *"there is wait"* in both question and answer. The second shows that participants have more precise recognition of the question-answer nature of the program, and hence the need for an answer from the user so that the program can continue. One participant explains: *"if he asks you then you have to type an answer, then he will respond"*, and another participant says: *"because you have not typed anything"*.

M14: A variable is (merely) a pairing of a name to a changeable value (with a type). It is not stored inside the computer
Participants with the misconception=33.3%, Correct=60.3%, n=63

Misconception: In total, 33.3% of the participants (n=63) chose a misconception choice (see Figure 6.3). In a detailed manner, 7.9% of the participants chose the answer indicating that *"the variable is not being stored anywhere"*. Participants who provided their reasons here seem confused by the built-in option provided by Scratch: Cloud variable (stored on server), which was shown as part of the question. Two participants highlighted that their choice came as a result of this option not being ticked, which is the default in Scratch. 25.4% of participants chose the answer that indicates *"the variable is only visible on the screen"*. For these participants, the location at which the storage occurs is important and needs to be

sensed. One participant says: *"because you can't see it anywhere else"*, and another one: *"there is nowhere for it to be so it just sits there"*. Another participant, despite choosing the wrong answer, indicates an analytical approach that is one step-away from being correct. The participant is uncertain where a variable should be stored because the program is not run, saying: *"the code will only set to value when run, and as the code is not yet running, the variable is moot"*.
Correct: Participants who answered correctly vary in their reasoning. Some give a concrete reason, for example *"It sits in the RAM memory"*, and *"all computers store data in the hardware, to know what they need to do"*. Others focus on the need to save the whole program for Scratch to *"remember"* it later.

M26: A false condition ends program when no `else` branch
Participants with the misconception=7.4%, Correct=51.6%, n=95


You have the following program to check if Gobo is touching the black wall:

```
when 🏴 clicked
if touching color ■ ? then
    say Auw!! for 2 secs
say I am moving for 2 secs
```

At the moment, Gobo is far from the black wall:

What will Gobo say after we click the green flag ?

Figure 6.5. The question representing M26

Misconception: Although we have not provided an answer option to represent the misconception (See Sect. 6.5.4), some participants (7.4%) show they hold the misconception after analyzing the text they wrote. For those, the code does not execute at all because the condition is `false`. For example, one participant says: *"he [Gobo] does not touch the black, so nothing happens there"*. Another participant finds it illogical to execute the code: *"he [Gobo] is not touching the black so why would he do the commands that only apply to him if he is touching the black"*.
Correct: Participants highlight the false condition as a motive to their answer, from the opposite perspective to the participants with the misconception. The condition being `false` means the program runs, but parts of it are skipped. For example, one participant states: *"he skips the Auw part because he is not standing on the black"*, and another participant agrees: *"Gobo says only I am moving because he does not touch the black wall"*.

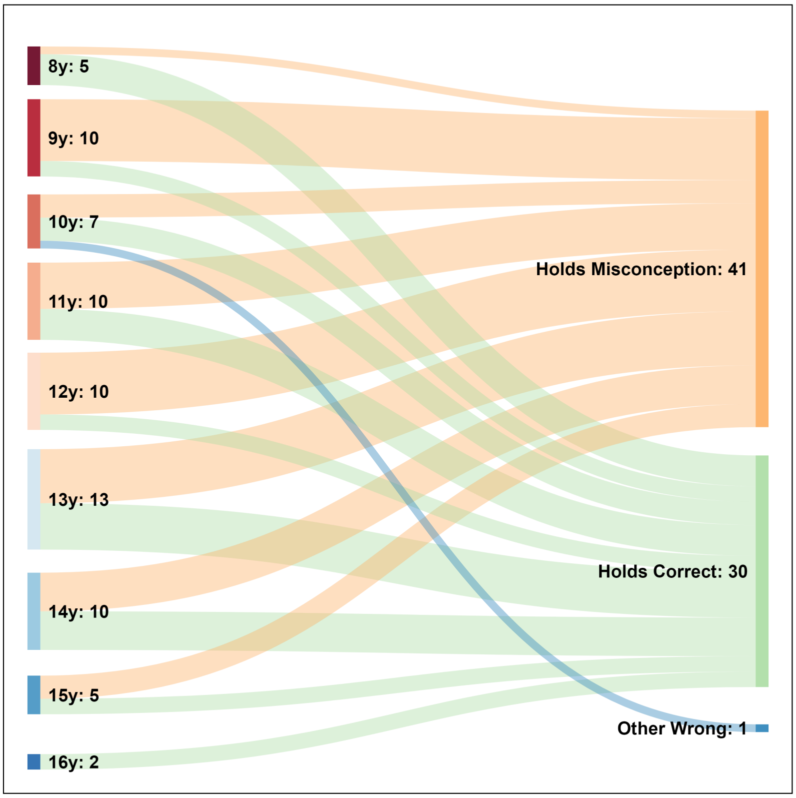


Figure 6.6. An alluvial diagram for M23 (Difficulties in understanding the sequentiality of statement), the top occuring misconception. It shows the flow of answers from age groups 8y-16y towards the answer categories

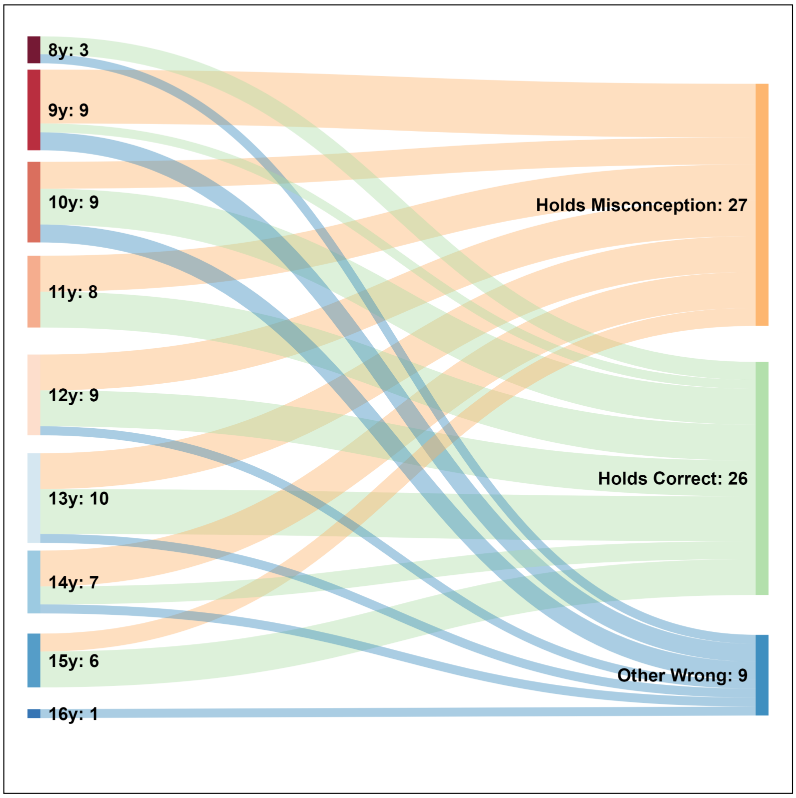


Figure 6.7. An alluvial diagram for M9 (A variable can hold multiple values at a time), the second top occuring misconception. It shows the flow of answers from age groups 8y-16y towards the answer categories

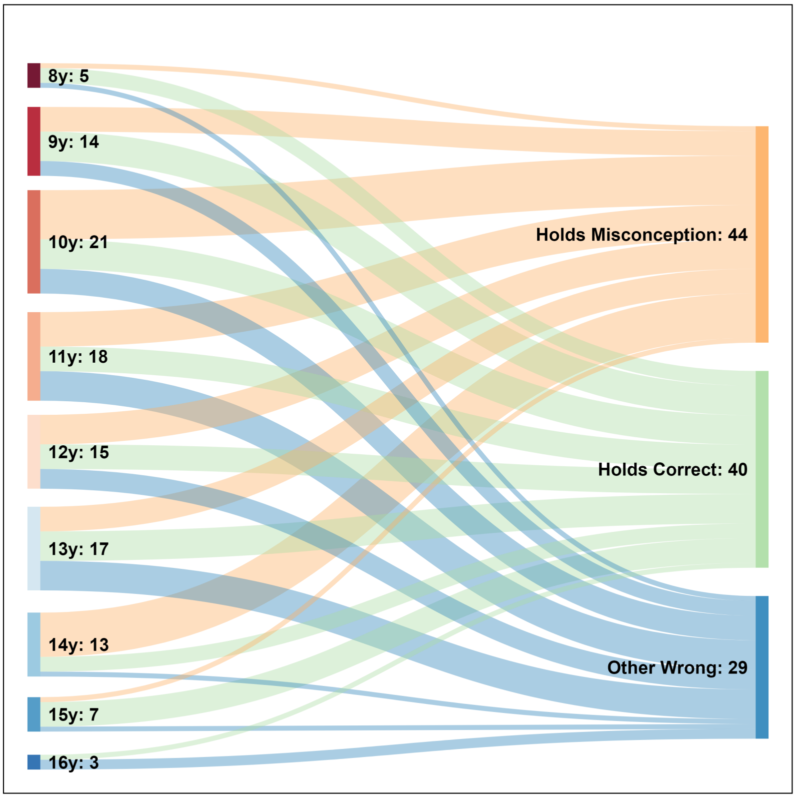


Figure 6.8. An alluvial diagram for M150 (Difficulties understanding the effect of input calls on execution), the third top occuring misconception. It shows the flow of answers from age groups 8y-16y towards the answer categories

6.4.3 EFFECT OF AGE AND PREVIOUS PROGRAMMING KNOWLEDGE

[RQ3] How do age and previous programming knowledge affect the holding of a misconception or the correct understanding?

AGE FACTOR

For the effect of age analysis, we exclude the age points 7 and 17 because only one participant in each of these age categories answered the questionnaire. Results show that a positive correlation exists only between age and holding the correct concept (Spearman's Rank Correlation *p* =0.005). In words, the older the child the more they answer correctly. Additionally, when considering the category of the misconception according to Sorva's original classification (see Table 6.1), positive correlation is found between age and correctly answer the misconception questions under the *"Variable"* category (Spearman's Rank Correlation *p*=0.015) and *"Control"* category (Spearman's Rank Correlation *p*=0.048).

In the alluvial diagrams (See Figures 6.6 , 6.7 and 6.8) we observe how age groups contribute to the answers for the top three misconceptions. The contribution is represented by the thickness of the flow from source to destination. The diagrams show that children younger than 12 are more likely to hold a misconception than older children. However, the relation is only significant in holding the correct answer as stated above, and not in holding a misconception.

PREVIOUS PROGRAMMING KNOWLEDGE

The reported knowledge of programming languages (Figure 6.9) shows that almost two thirds of the participants programmed before with Scratch, while the remaining third used programming with a variety of other languages such as Lego, Alice, Python or Javascript. Moreover, the participants reported where they learned programming: at school (62%), at home (28%) or other places such as friends, communities or courses (10%). Since we investigate misconceptions in Scratch, we explore how knowing Scratch in particular compares to other programming languages when it comes to holding a programming misconception. The results show the following:

Knowing Scratch and other languages: is found to decrease the tendency to holding a misconception (Pearson Product-Moment correlation, r=-0.077, *p*=0.035) and increase the tendency to holding correct concepts (Pearson Product-Moment correlation, r=0.151, *p*<0.001).

Knowing other languages but not Scratch: is found to increase the tendency to holding a misconception (Pearson Product-Moment correlation, r=0.071, *p*=0.049).

Knowing Scratch only: knowing only Scratch does not correlate with the holding of a misconception. However, results show that it correlates with answering incorrectly under the *"Other_Wrong"*category (Pearson Product-Moment correlation, r=0.081, *p*=0.025).

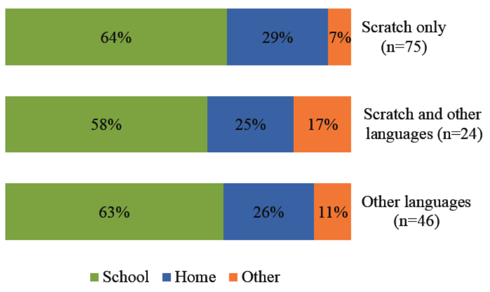
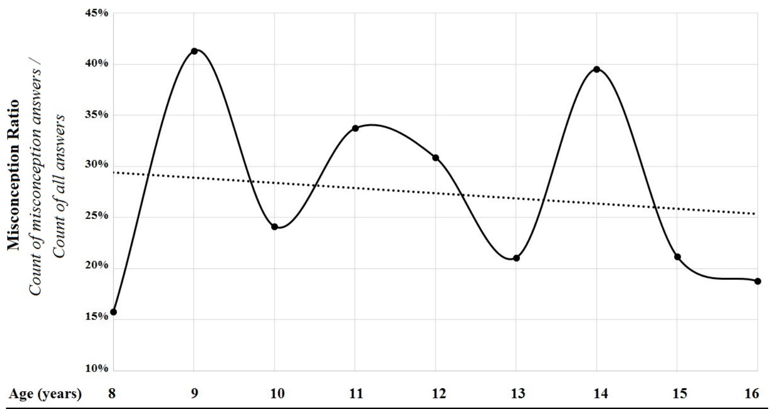


Figure 6.9. Reported programming languages versus where the child learned them: schools are the primary source of learning programming, while almost one-third of the children indicate home as the learning place for programming.

6.5 Discussion



Age (years)	8	9	10	11	12	13	14	15	16
Participants	10	16	31	27	17	17	15	7	3
Answer Count	45	103	100	102	104	128	92	58	18

Figure 6.10. The ratio of misconception answers to the total answers per age (year)

6.5.1 General observations

While some could argue that it is expected that younger children hold such misconceptions, their origins could be different from misconceptions in university level students. First, the relation with mathematics seems more pervasive: whenever a question contains numbers, some participants chose to sum the values, even when there is no sum operation in the question (such as M9). The semantic meaning of variables is another recurring aspect that is present in questions other than M17. For example, for misconceptions M33 and M31, since we use the variable name *"counter"*, some participants believe that the program must *"count"* the sequence from one till the end regardless of the loop's condition and the operation inside.

Looking further into the effect of age on misconceptions, we present Figure 6.10 which shows the misconception ratio for the children per age. The results show that there exists a sudden increase in the holding of a misconception at age=14. The increase is significant compared to the neighboring age groups (13y,15y). Unfortunately, the increase cannot be explained by the currently collected data and needs further research.

6.5.2 Scratch-specific issues

The programming language has its role in eliciting some misconceptions. In previous research this contributed not only to syntactical errors, but also to errors related to the difficulty of understanding the notional machine of this programming language. In our study, we notice that the use of particular blocks in our programming problems caused common and worthy remarks from the participants. First, the *"Say"* block in Scratch caused confusion whenever it was used in the questions. Despite it being a basic and common block, it led to errors when integrated with a variable instead of plain text. Many participants in this case indicated that the program will say the variable's identifier, for example *"X"* or *"total"*, instead of its value. This signifies both a lack of understanding of the variable concept, and the effect of the lingual imperative sense of the verb *"Say"*. A second difficulty we note being related to Scratch is the block used to set a variable to a value: *"Set [variable] to [value]"*. The use of the preposition *"to"* adds ambiguity over the direction of the assignment. Finally, the order of the blocks should be respected because, for some children, the *"visually-attached"* blocks meant that the program will execute in the order from top to bottom even when condition or repeat block exist.

6.5.3 Reflections on and implications of the results

Misconceptions are considered one area of difficulty in learning how to program. Our results suggest that as we move towards younger children we see more difficulties in answering the questions correctly. This is especially the case for the concepts of variables and control. This result confirms previous research that have shown that children younger than 11 have difficulties understanding those concepts in addition to the concepts of parallelization and procedures [97, 198]. Our results shows, however, some different observations than those in university level students: i) children tend to perform mathematical operations (mostly summations) whenever numbers are present in the exercises, and ii) tend to make assumptions based on their understanding to the semantic names of variables. Consequently, we believe that more diverse exercises should be developed to include operations on strings and booleans, in addition to carefully selecting identifier names in those exercises.

Moreover, our findings suggest that children who indicated programming in Scratch only have difficulties in understanding the concepts correctly and tend to choose other wrong answers. This result highlights two issues. First, educators and researchers are encouraged to identify and realize new misconceptions induced by Scratch as a language. We provided a few observations from our dataset (See Section 6.5.2), but more research is still needed. Second, our results suggest that younger children start learning programming in Scratch (age positively correlates with the number of programming languages reported in addition to Scratch), and they do it primarily in schools. Those younger *"Scratchers"* seem to have more difficulties to correctly understand the concepts in our study. This result confirms previous research which found Scratch projects to have low percentage of conceptual constructs such as variables, procedures and conditional statements [4, 198, 212]. As a result, we believe that primary education providers are advised to develop more nontrivial and concept-rich materials in Scratch.

Finally, a few participants showed signs of a struggle between contradicting thoughts while reasoning their answers (See Section 6.4.2). This indicates our belief that holding a misconception is not binary: you either understand the concept or not. On the contrary, it can be a step into grasping the complete concept, and when the confusion is identified it becomes easier to provide the missing piece of information by educators.

6.5.4 THREATS TO VALIDITY

Like all studies, our study has some limitations. An external threat to validity comes from the reported experience in programming which the participants provided. In the questionnaire we ask five questions about the previous experience in programming, including questions to identify knowledge of particular concepts. The participants could have still misjudged their own experience and gave false indications. Moreover, a construction threat to validity comes from using multiple choice questions because some children would have guessed the answer despite lacking the knowledge. We eliminated to the minimum those two threats by adding an open-end text following each multiple-choice question, then strictly filtered out any answer for which the participant indicated in the open-end text that they guessed the answer or lacked the understanding of the question or the knowledge to answer it. An internal threat to validity is the design of the question and possible answers for M26. The answers we provided did not include an answer which reflects the misconception: in this case that the program will do nothing. Despite this design issue, 7.2% of 95 participants who answered this question hold the misconception, which was based on the text they provided using the *"Other"* answer option.

6.6 CONCLUSIONS

This chapter aims at exploring programming misconceptions held by school-age students. The study is based on a multiple-choice questionnaire with programming exercises in Scratch. 145 children participated in the study, aged between 7 and 17 and have some previous experience in programming. The results show that younger learners in school-age indeed hold misconceptions, which caused them to make errors when tracing a small script in Scratch. The top three common misconceptions span over multiple concepts; M23: the difficulty of understanding the sequentiality of statements, M9: the difficulty of understanding that a variable holds one value at a time, and M150: the difficulty to understand the interactive nature of a program when a user input is required. The origins of these misconceptions vary and include the great influence of numbers and mathematical operations in the mindset of the children, the influence semantic meaning of the variable identifier, and the wrongful expectation of what a computer can do, i.e. misunderstanding the notional machine. When analyzing the age effect we found that older children tend to answer the exercises correctly. Moreover, results show that knowing Scratch in addition to at least one other programming language positively influence choosing a correct answer. While children who reported programming in Scratch only had more tendency to choose other wrong answers. Finally, examples we observed in the experiment signify that holding a misconception is indeed a step towards holding the correct and complete concept. Our findings suggest that educators should count for the misconception effect as early as possible. Additionally, we should realize new misconceptions induced by Scratch as a language, due to, among other reasons, the use of visually-attached blocks and the use of special keywords in its block set. Finally, Scratch material and lessons should integrate more concept-rich exercises that highlights areas such as variables and control of execution. This is especially needed for children who start learning programming in Scratch. In future work we have two main directions. First we intend to explore and identify Scratch-specific misconceptions. Second, we aim to study in more depth the effect of learning another programming language on Scratch learners. Finally, we aim at developing and testing teaching materials and methods that have less possibility of inducing misconceptions in children.

7 Effects of Vocalizing Code on Comprehension

We are increasingly teaching programming to younger students in schools. While learning programming can be difficult, we can lighten the learning experience of this age group by adopting pedagogies that are common to them, but not as common in CS education. One of these pedagogies is Reading Aloud (RA), a familiar strategy when young children and beginners start learning how to read in their natural language. RA is linked with a better comprehension of text for beginner readers. We hypothesize that reading code aloud during introductory lessons will lead to better code comprehension. To this end, we design and execute a controlled experiment with the experimental group participants reading the code aloud during the lessons. The participants are 49 primary school students between 9 and 13 years old, who follow three lessons in programming in Python. The lessons are followed by a comprehension assessment based on Bloom's taxonomy. The results show that the students of the experimental group scored significantly higher in the Remembering-level questions compared to the ones in the control group. There is no significant difference between the two groups in their answers to the Understanding-level questions. Furthermore, the participants in both groups followed some of the instructed vocalizations more frequently such as the variable's assignment (is). Vocalizing the indentation spaces in a for -loop was among the least followed. Our results suggest that using RA for teaching programming in schools will contribute to improving code comprehension with its effect on syntax remembering.[1]

[1]This chapter is based on our paper: The Effects of Reading Code Aloud on Comprehension: An Empirical Study (Alaaeddin Swidan and Felienne Hermans) which appears in the proceedings of the 1st Global conference on Computer Education, CompEd 2019

7.1 Introduction

Programming is increasingly taught to younger students, in some countries as part of the curriculum of primary and secondary schools [97]. We know, however, that learning programming can be difficult [32, 136, 232]. The question arises on how do we make learning programming less difficult for younger students? One way could be applying pedagogies we know work for this age group but are uncommon in programming education.

Young children start to learn how to read by learning the connection between symbols, one or more letters in this case, and sounds and then combining them into words and sentences. Reading text aloud is encouraged for beginners since it focuses thoughts, help memorization and improves comprehension of text [35, 72, 214]. Also in mathematics, the same approach to reading aloud can be noticed in vocalizing simple operations and equations, or when introducing a new symbol [78, 197].

Although in later development stages and adulthood silent reading becomes the norm, our brains seem to be always ready for reading aloud. Studies have shown that the brain sends signals to the primary motor cortex, controlling the lips and the mouth, during silent reading [167, 183]. This brain activity is called *subvocalization*, which is used in particular when learners face long and new words. In programming education, educators seem to spend little effort on reading code aloud to, or with the students. The lack of this *phonology* knowledge leaves students with an extra cognitive load when reading code to understand functionality. In this regard, one study measured the subvocalization of experienced developers during programming tasks and showed that the subvocalization signals could differentiate the difficulty of the programming task [167]. Therefore, we hypothesize that training students in reading code aloud will lead them to spend less cognitive effort on the reading mechanics and thus improve their comprehension of code.

Therefore, the purpose of this study is a first quantification of the effect of reading code aloud during lessons on school students' comprehension of basic programming concepts. Furthermore, we investigate how students benefit from the practice of Reading Aloud (RA) by following it as a sort of a guideline later.

To this end, we design and execute a controlled experiment in which 49 primary school students receive three lessons of programming in Python. The students are divided into two groups which get the same teaching materials and times. The students in the experimental group, however, are asked to repeat reading the code aloud following the instructor. We assess students' learning based on Bloom's taxonomy. Since the participants are absolute beginners in programming, we focus our assessment on the first two levels of the taxonomy: the Remembering-level and the Understanding-level. In this chapter, we answer the following research questions:

RQ1 What is the effect of reading code aloud on the performance of students in the Remembering-level questions?

RQ2 What is the effect of reading code aloud on the performance of students in the understanding-level questions?

RQ3 How do students follow the vocalization guideline when they read code later?

Results show that the students in the experimental group scored significantly higher in the Remembering-level questions compared to the students in the control group. There is no significant difference between the two groups in their answers to the Understanding-level questions. The analysis shows that particular code vocalizations, such as the variable's assignment, are common among the two groups. On the other hand, participants in both groups least vocalize the spaces needed for indentation in a for loop and list brackets. The following sections contain the details of the experiment's design and results.

7.2 Background and Related Work

We provide an overview of research related to Reading Aloud (RA), particularly, the RA role in reading education for young students (Section 7.2.1) and previous literature involving the use of voice in programming environments (Section 7.2.2). We also overview selected prior research on the use of Bloom's taxonomy in assessing programming comprehension (Section 7.2.3).

7.2.1 Reading Aloud and Comprehension: Natural Language Perspective

Most psychologists nowadays believe that reading is a process of sounding out words mentally even for skilled readers [183]. Brain studies [167, 170, 183] show that the primary motor cortex is active during reading, "presumably because it is involved with mouth movements used in reading aloud" [183, p. 90]. Therefore, it becomes highly important for beginner readers to learn the connection between sounds and symbols, or phonics. Previous research found that systematic phonics instruction produces higher achievement for beginning readers, where they can read many more new words compared to students following other approaches. For these reasons, in the United States, phonics has been included in reading programs in schools nationwide [183]. As a verbal approach, reading aloud (RA) helps in focuses thoughts and transforming it in specific ways, causing changes in cognition [72]. Takeuchi et al. [214] highlight that RA is effective for children language development in *"phonological awareness, print concepts, comprehension, and vocabulary"* [214]. Bus et al. [35] reports that reading books aloud brings young children *"into touch with story structures and schemes and literacy conventions which are prerequisites for understanding texts"*. Several experiments related to comprehension report that students identified the sounding out of words, or loudly repeating text as a means to regulate their understanding while reading [129, 143]. When comparing RA to silent reading, research has found that students comprehend significantly more information when they read aloud versus reading silently [146, 177]. Although other studies showed opposite results [93], there seems a consensus exists among researchers that the effects of reading aloud may differ based on the reading proficiency of the students: beginning readers, regardless of age, benefited from reading aloud rather than silently [93, 177]. Finally, Santoro et al. [193] stress the importance of careful planning when reading aloud is aimed at improving the comprehension of students. RA activities, in this case, should be combined with *"explicit comprehension instruction"* and *"active, engaging in discussions about the text"*.

7.2.2 The Role of Voice in Programming and CS Education

One main use of code vocalization is as an assistive technology that helps programmers who suffer from specific disabilities or stress injuries (RSI) to program in an efficient matter [19, 73, 84, 207]. Another area where code vocalization is essentially practiced is the remote peer-programming [40]. Vocalizing code can also be an element in some teaching strategies especially in direct instruction, modeling and think-aloud [10, 216]. However, in all of these cases, the way in which people vocalize code is not systematic, standardized, or agreed upon. In addition, there is some ambiguity over what to vocalize and on what granularity level: tokens, blocks or compilation units [84]. These factors lead to challenges for professional programmers and learners alike [20]. For example, [19, 207] mention the problematic issue of how to vocalize symbols, and when to speak out or leave specific symbols. Price et al. [175] mention the effect of natural language's flexibility on the difficulty of vocalizing programming commands, as multiple words could be used to do the same thing (for example begin class or create a class). Another effect of natural language is the ambiguity of the meaning of some words in different contexts, for example, *add* value to a variable and *add* a method to a class. These challenges show that the use of natural language in programming needs more attention from programming designers and educators. Recent work of Hermans et al. [104] calls for the programming languages to have phonology guidelines that specify how a construct should be vocalized. Finally, related is the work of Parnin [167] who investigated the role of subvocalization on code comprehension. Subvocalization is the process of the brain sending electrical signals to the tongue, lips, or vocal cords when reading silently. Silent reading is a relatively new technique for humanity. Therefore, when reading, especially the complicated segments or even words, the brain instructs the lips and the tongue to perform the read-aloud but without a voice. Their experiment on code reading showed that measuring the subvocalization signals can be an indication of the difficulty of a programming task.

7.2.3 Bloom's Taxonomy in CS Education

When it comes to the assessment of learning processes, Bloom's taxonomy is one of the common frameworks educators follow [135, 220, 232]. In this framework [8, 29], Bloom identifies six levels of cognitive skills that educators should aim at fulfilling with their students. The levels are Remembering, Understanding, Applying, Analyzing, Evaluating, and Creating. These cognitive levels are ordered from low to high, simple to complex and concrete to abstract, and each is a prerequisite to the next. This classification is combined with a practical guideline that educators could use to evaluate the learning outcome of their students by forming questions with certain verbs. In this way, it stimulates the cognitive process of the required level of the taxonomy. In CS education there appear two main usages of the taxonomy. First, the use of Bloom's taxonomy as a tool to measure the learning of students and how they perform in introductory courses in particular. Some research chose to build the assessment from scratch depending on the taxonomy. This includes the work of Whalley et al. [232] who assessed the reading and comprehension skills of students in introductory programming courses, creating a set of questions that conform to Bloom's taxonomy. The results show that the students performed consistently with the cognitive difficulty levels indicated by the taxonomy. Similar is the work of Thompson et al. [220] who created another set of programming questions per the main categories of the taxonomy, discussing each item and showing educators how to interpret the results. Both works have been insightful to our research. Second, other researchers have applied the taxonomy to evaluate existing programming exams or courses. Lister [135] argues that the taxonomy should be used as a framework of assessment, not learning since it provides a reliable tool with a standard level of assessment. Despite its popularity among researchers, there seems a consensus that applying the taxonomy to programming questions is challenging since a programming problem consists of several building concepts which makes isolating the problem to one cognitive category a hard job to do [82, 220, 232]. Although the challenging task to map the assessment items to Bloom's cognitive levels will always depend on the interpretation of the educators, the taxonomy should still provide a valuable tool to explore the cognitive processes involved in any programming exercise.

7.3 METHODOLOGY

The goal of this study is to answer an overarching research question: how does reading code aloud during lessons affect the students' learning of programming concepts? To this end, we designed and ran a controlled experiment with primary school students. In this section, we describe the setup and design of the experiment in addition to the theoretical basis we use for the assessment.

7.3.1 SETUP

We provided Python lessons to 49 primary school students in the Netherlands. We split the participants into a control and an experimental group. Both groups received the same lessons: three lessons of 1.5 hours each given by the researchers, one lesson per week. We gave the lessons to the groups subsequently: first the experimental group, followed by a break, followed by the control group. The students knew they were going to learn programming during the lessons but they were not aware of the experiment's goal. We asked the consent of the parents to collect the anonymous data needed for the research.

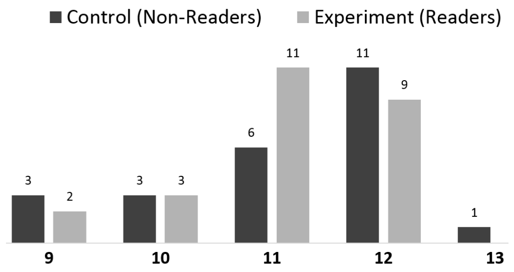


Figure 7.1. Age in years versus count of participants per group, mean=11.12 years. Both groups have equal age means

7.3.2 PARTICIPANTS

Participants are 49 students of one primary school in Rotterdam, the Netherlands. The programming lessons are provided as part of extracurricular activities arranged by the school, taking place during school days in a computer lab at the school. As shown in Figure 7.1, a total of 49 school children between 9 and 13 years with an average age of 11.12 years participated in the study. Participants were 28 boys, 20 girls, and 1 participant who chose not to specify their gender. The control group consisted of 24 children (age average=11.167

years, 6 girls - 17 boys - 1 unspecified), while the experimental group consisted of 25 children (age average=11.08 years, 14 girls, 11 boys). We could not control the split of groups since they are school classes hence the non-balance in gender.

7.3.3 LESSON DESIGN AND MATERIALS

Each lesson starts by introducing a small working program. One teacher shows a program on the interactive white-board explaining the code per line and highlighting the concepts included. The lessons include the following concepts primarily:

Variables Setting and retrieving a variable's value

Lists Creating lists of integers and strings, accessing and modifying lists through built-in functions

For-Loops Using loops for repeating certain operations

Function use Calling built-in functions and using functions from packages.

During the program explanation, the teachers encourage the students to express their thoughts on what the code does via interactive questions, such as *What do you think happens if we change this value?* According to [193], reading text aloud aiming at improving the comprehension should be combined with *"active and engaging discussions about the text"*. Following, the students are instructed to work in pairs to carry out specific exercises according to the lessons' material. During the lessons, an online compiler for Python (Repl.it[2]) was used. The final assessment questions are on-line [3].

7.3.4 RA DESIGN AND IMPLEMENTATION

Understandably, there exists no guideline on how to read code. When reading code, however, people tend to find that there are ambiguous words, symbols, and even punctuation, and vocalizing them is both challenging and subjective [104]. Consider an example as simple as the variable assignment $a = 10$, is it vocalized as *"a is ten"*, *"a equals ten"*, *"a gets 10"* or *"set a to ten"*. In this experiment, we follow a similar approach to [20, 104] where the code is read as if the person is telling another beginner student what to type into a computer. For both the experimental and the control group, the instructor read the code aloud to the students during and following the explanation of a concept within the code, a for-loop for instance. Only the students in the experimental group, however, were asked to repeat the reading activity: all-together and aloud. We consistently read all keywords, symbols identifier names and punctuation marks that are essential to the working of a program, for example quotation marks, brackets, colons and white spaces necessary for indentation. The full list of what and how we vocalized code during the lessons are presented in Table 7.1. We call this list the vocalization *guideline*, and we use it later to answer RQ3 which investigates the extent to which the students follow the taught guideline later during the assessment. We do this investigation for both groups since all the students in both groups listened to the teacher vocalizing the code during the lesson, but only the experimental group's participants performed it themselves.

[2]https://repl.it/repls
[3]http://bit.ly/2EhAmB0

Table 7.1. The vocalization guideline used during the lessons

Vocalization Item	**Description**	**Code**	**How Code was Vocalized**
V1	Setting a variable value	temperature = 8	temperature **is** eight
V2	Function-calling with round brackets	for i in range(10): temperature = temperature + 1	for i in range **open round bracket** ten **close round bracket**
V3	For-loop colon		**colon**
V4	For-loop indentation space		**space space**
V5	Plus sign in expressions		temperature is temperature **plus** one
V6	Symbols in identifiers (underscore)	healthy_food = ['apple', 'banana']	healthy **underscore** food is
V7	List square bracket (open)		**open square bracket**
V8	Strings single quotation (begin)		**single quotation** apple **single quotation**
V9	Comma separating list items		**comma**
V8 (Repeated)	Strings single quotation (end)		**single quotation** banana **single quotation**
V7 (Repeated)	List square bracket (close)		**close square bracket**
V10	Function use: calling a function from a package with the dot	food = random.choice(healthy_food)	food is random **dot** choice **open round bracket** healthy underscore food **close round bracket**

7.3.5 Assessment

We choose to assess only the two basic levels of Bloom's taxonomy: Remembering and Understanding. According to Lister [135] the two categories are sufficient for beginners when we want to assess the effectiveness of their code reading. When relating these two categories to programming assessment, Thompson et al. [220] provide useful insights into how to interpret them into programming assessment terms. Remembering can be related to activities centered around identifying a programming construct or recalling the implementation of a concept in a piece of code. For example by *"recall the syntax rules for that construct and use those rules to recognize that construct in the provided code"*[220]. For the Understanding category, it includes translating an algorithm from one form to another, plus explaining or presenting an example of an algorithm or a design pattern. For example, tracing a piece of code into its expected output. Multiple choice questions are suitable to assess these two basic levels for beginners [135, 232]. We developed an 11-question final assessment exam: 9 are multiple choice questions, one is of fill-in type in addition to vocalizing the code snippet, and one only requires the student to vocalize a code snippet. We aimed that the questions cover i) all of the programming concepts we taught (see section 7.3.3), and ii) for each concept to have a question assessing the two targeted levels of Bloom's taxonomy. Table 7.2 shows the questions and their mappings to Bloom's levels.

Following the Vocalization Guideline

We ask the students in both groups to answer two vocalization questions (Question 9 and 11). The students need to write down in words how they would vocalize a code snippet to another beginner student. Although the students in the control group did not read the code aloud themselves, they listened to the instructor performing the RA. Therefore, we ask both groups to answer these questions. We use the students' answers to address RQ3.

Table 7.2. The list of questions and their corresponding Bloom's cognitive level

#	Concept(s)	Bloom's level	Prerequisite Knowledge	Student's Action(s) to Answer
1	List Create/-Modify	Remembering	The syntax to create a list of string literals	Replace syntactically incorrect line by a correct option
2	Variables	Remembering	The syntax to increase an integer variable's value	Replace an empty line with a syntactically correct option
3	Function use	Remembering	The syntax to call a function with a variable parameter	Replace syntactically incorrect line by a correct option
4	Function use	Remembering	The syntax to call the print function with a string literal	
5	Function use & Variables	Remembering	The correct syntax to print a variable's value	
6	Sequential execution & Variables	Remembering	Same indentation for each line of a Python block	Identify/recognize/locate the cause of the error from choices
7	For-loop	Understanding	For-loop syntax Indentation effect on lines being within/outside a loop	Trace and predict the outcome of a for-loop with a print statement within, followed by another print statement
8	For-loop	Remembering	For-loop syntax	Identify/recognize the syntactically correct for-loop to get a specific outcome
9	For-loop & Variables (with Vocalize)	Understanding	For-loop syntax Indentation effect on lines being within/outside a loop	Trace and predict the outcome of a loop that increases the value of a variable. Then write in words how you would vocalize the code.
10	List Create/-Modify & Function use	Understanding	The syntax of List creation, List access & modification using built-in functions	Trace code and interpret its use in one of low-,medium- or high natural language descriptions
11	Vocalize only	-	Vocalize code as if your are reading it to a friend	Write, in words, on the answer sheet how you would vocalize the code snippet

7.4 RESULTS

In this section, we provide the answers to our research questions.

7.4.1 RQ1: WHAT IS THE EFFECT OF RA ON THE REMEMBERING-LEVEL?

To answer this question, we investigate the answers to the questions in the Remembering-level (7 questions) (see Table 7.2). The control group has a mean of 3.58 while the experimental group has a mean of 4.56. To test the equality of means we use the Mann-Whitney U Test since the sample size is relatively small and the presence of some outliers. The results (Table 7.3) show that the difference between the control and experimental groups is significant (p =0.003). The effect size r= 0.42 which indicates a large effect [57, 94].
Age factor: There is no relationship between the student's age and the Remembering-level score across the two groups. All age groups have equal means in the two experimental groups.

Table 7.3. The difference by group in the answer score means to each category of questions.

	Remembering-level Score 7 Questions		Understanding-level Score 3 Questions	
	Readers n=25	Non-Readers n=24	Readers n=25	Non-Readers n=24
Mean	4.56	3.58	0.90	0.92
Std. Dev.	1.00	1.28	0.76	0.75
Mann Whitney *U*	156.5	443.5	304.5	295.5
z-score	2.97		0.09	
(2-tailed) *p*	0.003		0.93	
Significant	Yes ($r = 0.42$)		No	

7.4.2 RQ2: WHAT IS THE EFFECT OF RA ON THE UNDERSTANDING-LEVEL?

To answer this question, we investigate the answers to the questions in the Understanding-level (3 questions) (see Table 7.2). The control group has a mean of 0.92 while the experimental group has a mean of 0.90. Similar to RQ1, we use the Mann-Whitney U Test to check the equality of the means. The test results (Table 7.3) show that the difference between the control and experimental groups is not significant ($p = 0.93$).

7.4.3 RQ3: HOW DO STUDENTS FOLLOW THE VOCALIZATION GUIDELINE WHEN THEY READ CODE LATER?

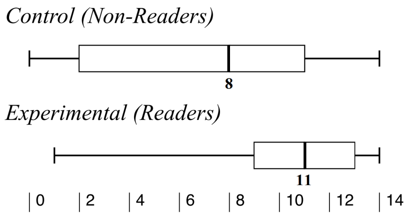


Figure 7.2. The vocalization score means by group

To answer this question we analyze students' answers to the vocalizing questions (Question 9 and 11 in Table 7.2), where we asked students to write down, in the answer paper, how would they vocalize two small code snippets.

The vocalization guideline is the way we chose to vocalize the code snippets provided during the lessons. It is summarized in Table 7.1. We grade the student's answers following the guideline; a point is given every time the guideline is followed, and the maximum possible is 14 points.

FOLLOWING THE GUIDELINE:

As expected there exists a significant difference between the two groups in following the vocalization guideline (see Figure 7.2). This is expected because of the intervention we did in the experimental group. The experimental group who read the code aloud themselves scored an average of 10.20, while the students in the control group, who only listened to the code being read, scored an average of 6.79. The Mann-Whitney test suggests the difference between the two means is significant (U=168.5, p= 0.009, r= 0.375 (a medium to large effect)).

MOST AND LEAST FOLLOWED VOCALIZATIONS

We analyzed the responses to the vocalization questions to assess how the vocalization guidelines were followed in both groups (Table 7.4). We notice that some vocalizations are frequent in both control and experimental groups especially the variable assignment (is), comma and single quotation mark. However, the colon in for-loop goes from one of the most frequent, in the experimental group, to the one of the least frequent in the control group. This difference can be linked to the intervention exercise making a lasting memory for the participants in the experimental group.

EFFECT OF FOLLOWING THE GUIDELINE:

Within one group, we analyzed whether following the guideline affects the answers to either Remembering- or Understanding-level questions. Results so far showed that students in the experimental group are more likely to score higher in following the vocalization guideline, and at the same time are more likely to score higher in the Remembering-level score, than the students in the control group. However, comparing the students' score within the experimental group itself does not show a relationship between following the vocalization guideline and the score in either the Remembering- or Understanding-level category. The control group, however, reveals a different behavior. Results show that students within the control group who scored higher on the vocalization questions scored higher on the Understanding-level questions (see Figure 7.3). According to the ANOVA test, the vocalization varies across the quantiles of the Understanding-level score (F=8.232, p=0.002). We highlight again that students in this group were not instructed to repeat the reading of the code, they only listened to the instructor reading aloud the code snippets.

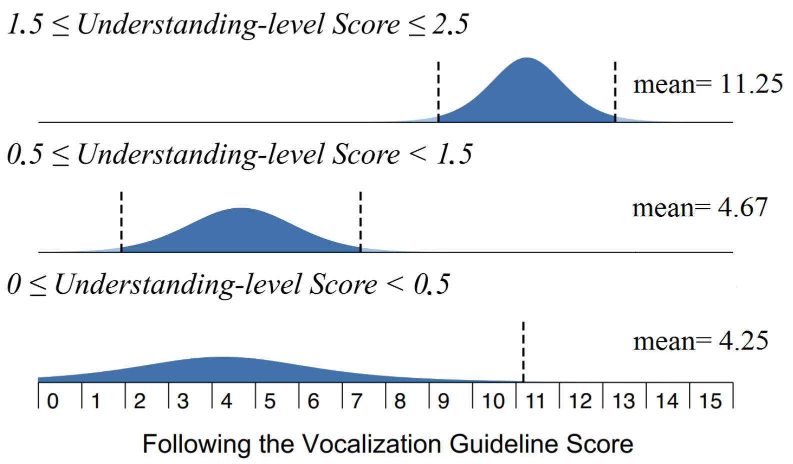


Figure 7.3. The variance of following the vocalization guideline among the participants in the control group and and the relation with the score on Understanding-level questions

Table 7.4. The most and least followed vocalizations following the guideline in Table 7.1

Most Followed		**Least Followed**	
The Experimental Group (n=25)			
V3 - Colon in for-loop (:)	22	V7 - List square brackets	11
V1 - Variable assignment (is)	21	V4 - Space indentation in for-loops	16
V9 - Comma (,) V8 - String single quotation (') V10 - Dot in function call	19	V5 - Plus sign	17
The Control Group (n=24)			
V1 - Variable assignment	14.25	V4 - Space indentation in for-loops	5
V6 - Underscore in variable names	14	V7 - List square brackets	6
V9 - Comma (,) V8 - String single quotation (')	13	V3 - Colon in for-loop (:) V2 - Round bracket for function call V10 - Dot in function call V5 - Plus sign	11

7.5 Discussion

7.5.1 Reflection and explanation of the results

The main finding in this study is the significant effect RA has on remembering the syntax of the programming constructs taught to the students. We believe this result encourages teachers in primary schools to practice code vocalization as pedagogy in their programming classes. While learning how to program is unique and can be difficult, it is still a learning process. We can, therefore, use pedagogies from other domains to help make programming easier to learn for younger students in particular. With that in mind, we can explain the effect of RA we observe in this study from two angles. First, RA improves the learning environment by utilizing a familiar technique to young students. This subsequently raises focus and attention of the students. When attention is gained and sustained learning can happen as *"attention is a prerequisite for learning"* [121, p. 3]. Secondly, RA helps in automating the retrieval of basic knowledge required for cognitive development [17]. According to the neo-Piagetian theories of cognitive development [144], students in their initial phase of learning programming are at the sensorimotor stage. At that stage, students mostly struggle in interpreting the semantics of the code they read, which affects their performance in tracing tasks in particular [216, 231]. Practicing RA could potentially help in reducing the struggle because it automates the remembering of the language constructs, and helps the student moving faster to the next development phases.

7.5.2 RA and the granularity of the vocalization

There is currently no standard guideline specifying how constructs of Python, or other programming languages, should be vocalized. As presented in Section 7.2.2, there are various granularities and strategies one can read code with. In this study, however, we follow a specific technique to vocalization (Section 7.3.4) which can be considered of a low granularity, focusing on syntax rather than semantics or relations within the code. Nevertheless, when teaching young and novice students, the RA method we follow could help teachers create a benchmark where students know how to call all the elements in their programming environment. We see from the answers of the control group students that there exist some variances in calling specific symbols. For example, calling the single (') a*"single quotation"*,*"apostrophe"*, or *"upper comma"*. This variance shows the challenges that beginners face to identify symbols in the first place. An extra cognitive effort is spent on remembering rather than on conceptual understanding. We hypothesize that higher granularities of vocalization of code structures or semantics can be integrated into the following phases. To determine the best vocalization method teachers should start with, however, is out of this study's scope and is an opportunity for future research.

7.5.3 Threats to validity

Our study involves some threats to its validity. First, the split of the participants into the two groups might have influenced the results. The split was introduced by the school structure; i.e., per class. However, we randomly selected one class as the experimental group and the other as the control group. The second threat, the authors being the teachers at the same time could introduce a bias in favor of the experimental group. To reduce the effect of such bias we ensured that both groups studied the same materials over the same amount of time with the same teacher. The main teacher was accompanied by another teacher who among other things observed the teaching given to the two groups. By these steps, we ensured that the only difference between the two groups would be the reading-aloud method. Third threat, a wrongful assignment of a question to one of Bloom's cognitive levels by the authors. This is a common challenge for researchers in similar studies [220, 232], and future experiments will lead to refining this process. Finally, a threat to the external validity of our study is the difficulty to generalize its results. This is, however, an inherent issue in similar studies with small sample size [178]. To overcome this threat we should replicate the study across different participants in the future.

7.6 CONCLUSIONS AND FUTURE WORK

Our study aims at measuring the effect of reading code aloud during programming lessons on comprehension. We perform a controlled experiment with 49 school students aged between 9 and 13. We assess the students' comprehension of basic programming concepts after three Python programming lessons. The assessment is based on Bloom's taxonomy and focused on the of remembering and understanding levels. The results show that the participants in the experimental group score significantly higher in remembering-level questions. However, the two groups perform similarly in understanding-level questions. Furthermore, we observe that the participants in both groups vocalize specific constructs more often than others. For example, the variable's assignment (is) and punctuation symbols (comma, underscore and quotation mark). Our results suggest that using RA for teaching programming in schools will contribute to improving comprehension among young students. In particular, it will improve remembering the syntax, paving the way to spending more cognitive effort on the higher level understanding of the concepts. For future work, we aim at experimenting with different RA approaches with different code granularities to find the best approach to improve code comprehension at this age.

8 Conclusion

In this dissertation, we investigate how end-user programmers perceive challenges while building their software programs. We look at two groups of end-user programmers: spreadsheet users and school-age children. With the unique features of each group, they still face challenges stemming from three overarching sources. First, the lack of system support that allows the end-users to build programs that are error-free and easy to understand and modify. Second, the missing of a collaborative process where building programs is shared and reviewed among peers. Finally, the lack of programming education and knowledge prior to building the programs.

Our investigation consisted of the following main activities: We worked with spreadsheet users to solve real-life challenges they face in building and maintaining business-critical spreadsheets. During our studies, we analyzed spreadsheet data, interviewed their users and contributed to developing software-support that allow end-users to overcome challenges.

We worked with school-age children from 4 to 18 years, as a special end-user group, on what challenges they face while learning introductory programming languages. In these studies, we provided lesson materials, collected data via questionnaires, observations and assessment to identify the challenges children face and how they perceive them. In Chapter 7 we offered a novel approach to help children overcome one of these challenges.

For the spreadsheet users, we investigated two main challenges related to spreadsheet performance and data extraction. Spreadsheet users in our case studies showed the need for external expertise to overcome these challenges and automate the processes. We provided software solutions that in one case reduced the execution times of a spreadsheet model, and in the second case, extract and transform cross-table data from a set of spreadsheets. In both cases our solutions were provided as external components to the spreadsheet environment. Therefore, end-users did not have to learn new knowledge or migrate out of spreadsheets.

Following, the challenges that school-age children experience depend on the programming environment and context. In Scratch, we showed in Chapter 4 that the current naming patterns of identifiers indicate that young end-users use longer names compared to other mainstream programming languages such as Java, PHP, and JavaScript. Besides, naming patterns observed show frequent use of features that are specific to Scratch such as the use of white spaces. Those two observations could impede the future transition towards the mainstream textual languages. On the other hand, we highlighted that single letter names constitute ca. 4% of the overall identifier names in the analyzed dataset. This usage, while still lower than the observed situation in professional programming environments, indicates that Scratch end-users could have difficulties maintaining and comprehending programs that include a lot of single-letter variables. In robotics and games that are intended to teach programming, we reported in Chapter 5 some of the challenges faced by preschoolers, an even younger target group for programming environments. For instance, the difficulty to relate the spatial characteristics of objects to the person's position. In Chapter 6, we identified some of the misconceptions that children hold that are related to basic programming concepts. The effect of these misconceptions last long and therefore could be a barrier to any programming experience in the future. Finally, we investigated how young students read code, which is a core skill that relates to comprehension in a textual introductory programming language. We found, in Chapter 7, that young students have difficulties reading some of the code constructs and symbols in Python. We evaluated there a novel approach based on reading code-aloud and its impact on comprehension. Reading code aloud, we found, helped students in performing better when the task required the mastery of the basic cognitive skill of remembering.

8.1 Contributions

The contributions of this dissertation are:

- An approach to improving a large-scale spreadsheet model evaluated in a case study.
- An approach to identify, extract and transform cross-table data from a set of spreadsheets into a relational database format, and a case study that evaluates the approach.
- The identification of naming characteristics of variables and procedures in Scratch programs.
- A comparison between the naming characteristics in Scratch and mainstream programming languages.
- An overview of challenges that preschoolers face when working with programming and robotic environments.
- The identification of programming misconceptions that school-age children hold in Scratch.
- An approach to systematically read code aloud.
- An evaluation of the effect of reading code aloud in classrooms on code comprehension in Python.

8.2 Revisiting the research questions

Given the results presented in the previous chapters, we look back at the main research questions of this dissertation.

8.2.1 [RQ1.A] How do spreadsheet users deal with challenges they face when working with spreadsheets?

Spreadsheet professionals face a variety of challenges throughout the different phases of building a spreadsheet. In this dissertation, we highlighted two challenges that spreadsheet professionals face in an enterprise and large-scale business environment.

Performance issues In Chapter 2 we report how a critical spreadsheet model suffered from performance issues that caused long and excessive execution times, which led to disruption to business processes. The spreadsheet model, a Monte Carlo simulation, was mostly developed in the VBA scripting part of the spreadsheet. By nature, the spreadsheet performed a high number of iterations to predict the outcome of a certain variable.

Data extraction and migration In Chapter 3, spreadsheets used over a long period contained important information for the organization. The data was semi-structured in a cross-table structure. Current migration tools, including the built-in features in spreadsheets, are not able to handle the extraction and migration properly. Therefore, there was a need to extract and transform the information reliably and efficiently.

In both cases, spreadsheet users currently handle those challenges by first trying to overcome them using available spreadsheet features and their domain-specific programming knowledge. In the case of the Monte Carlo model, spreadsheet users attempted to modify the code to speed up the execution. At some point, they accepted the long-execution times as part of the business process, but the model's instability and its unexpected faults during these runs pushed towards another more sustainable solution. For the data extraction, spreadsheet users in the second case were able to perform this process but only manually. However, a manual process was time-consuming. For those two reasons, spreadsheet users asked for help from external parties because the needed solutions (parallelization and extraction automation) required skills and knowledge outside the team's expertise.

8.2.2 [RQ1.B] How can these challenges be addressed?

Inspired by the work on End-user programming engineering, one way to overcome these challenges is to provide end-user programmers with tool- and system-support that help the users to produce the needed results. In this regard, we provided in Chapter 2 and Chapter 3 technical approaches to overcome the performance and data extraction challenges. For the spreadsheet model with long execution times, we provided an approach based on parallelizing part of the model's workload to a High-Performance Cluster of machines. For the data extraction, we provided an approach based on the automatic identification of labels and data cells, followed by transforming the identified cells to a collection of data tuples. In both cases, we provided a software solution for end-user programmers based on these approaches.

Providing these software-based solutions to end-users was neither easy nor straightforward. From the studies performed, we recorded and identified two challenges that spreadsheet users face when the tool-support is provided. First, is the lack of knowledge on using the tool. For example, in Chapter 2, a parallelization-based solution we introduced to spreadsheet users who had no prior knowledge of how parallel computing works. This caused a reported 21% of the project hours being spent on providing help to spreadsheet users to run the model. The application of the parallelization meant that changes to existing code structures are inevitable, which in turn required time for end-users to adapt to the new VBA code-base.

The second challenge is the limited generalization of software-based solutions to spreadsheet problems. Spreadsheets provide a high level of flexibility and freedom to end-users. Without the application of strict control measures, it becomes difficult for one solution to attend to a variety of spreadsheet models. A semi-automatic software solution that takes the input of the end-user into account is the best option when more generalization is needed.

8.2.3 [RQ2.A] HOW DO SCHOOL-AGE CHILDREN DEAL WITH CHALLENGES THEY FACE WHEN LEARNING TO PROGRAM?

In the studies we performed with school-age children we highlighted several challenges that could be a barrier to producing better programs and developing Computational Thinking skills. These challenges are:

Naming practices In Chapter 4, we showed that end-user programmers in Scratch follow, similar to professional developers, a *"not recommended"* approach to naming in single-letter variables. This makes the comprehension of programs by other users more difficult. Other naming patterns are specific to Scratch such as longer identifier names with white spaces and numeric variable names. This could make the users' transition to another programming environment in the future particularly challenging since the mainstream programming languages do not apply these patterns.

Spatial Awareness In Chapter 5 we observed that preschool children face particular challenges in their understanding and identification of the spatial characteristics of an object especially when it contradicts with the position of the child itself.

Programming Misconceptions In Chapter 6 we showed that school-age children who program in Scratch have misconceptions related to basic programming concepts and the computer role in programming in general.

Code reading In Chapter 7 we show that the way school-age children have difficulties when they read Python code, especially when reading symbols. The reading style affects their code comprehension.

We believe that previous programming experience, especially in the age group we targeted, shape the future perceptions of programming and the more general Computer Science. Children vary in their way of perceiving these challenges. Most of these challenges are foreseen in the future, and as a child, you have less depth in understanding the effect of the challenge. However, we see variations of how children deal with these challenges. Some children have a huge intrinsic motivation towards the generic computer technology, and they see programming as under this. Therefore, they are more engaged when they face a challenge, asking for assistance and feedback, and consequently picking from there to continue working on their programs. Other children with less motivation, are more prone to disappointment and frustration when they perceive some difficulty in their programming activities. It is crucial for them to see the computer interacting with their programs as anticipated all the time. These children, therefore, require extra scaffolding and assistance to help them overcome the sense of difficulty.

8.2.4 [RQ2.B] How can these challenges be addressed?

In general, the challenges we presented earlier aim at providing more awareness into the difficulty of programming and the possible barriers that school-age programmers are facing. The awareness is important for different stakeholders in programming research. We believe educators will find the identification of programming misconceptions that students hold beneficial. This could be supported by creating misconceptions-aware programming materials in classrooms and online tutorials.

The awareness of the different naming patterns in Scratch can be an important trigger for programming language designers to integrate some of these patterns into their languages so that the transition would be more natural for the end-user programmers in the future. Finally, the awareness of challenges related to object spatial positions could be relevant to visual language designers, including spreadsheets, to provide more support and information when the object location is important to understanding the program context.

In Chapter 7, we presented one practical suggestion to help in overcoming the code reading challenge. The systematic guidelines-strict reading-aloud of code snippets could be a way to help students, in the short term, understand programs better. In the long run, this could contribute to a better experience for the end-user programmers in peer-programming or collaborative program development where reading code is essential between peers.

8.2.5 Comparing the two groups

We look here at some striking similarities and differences between the two groups when we consider the challenges each one faces.

First, we revisit the end-user learning barriers introduced by Ko et al. [125] to see how the difficulties reported in this dissertation fall within the barriers. Table 8.1 shows which group indicates difficulties belonging to which barrier. In the parallelization solution provided to solve the performance issues, spreadsheet users struggled, in the beginning, to understand how the parallelization components work (Coordination) and how to use the software solution (Use). Before asking for help, spreadsheet users identified the existence of the problem but could not solve it with their current expertise (Design) and could not invest in the time to search and find useful information (Information). For the School-age children, they have misconceptions about programming and computer technology in general (Understanding). Also, we notice the difficulty children have in using specific programming constructs which related to alternative decisions and functions for instance (Design). Moreover, we observed in our experiments multiple times that students with an idea are stuck to execute it because they do not know the function yet, or they ask for a feature that is too advanced compared to their current level and requires extra materials (Selection).

Similarities

Performance and optimization of spreadsheets: We explored how spreadsheet users handled the performance problem. They faced challenges in identifying the time-consuming part of a spreadsheet and most importantly the possible alternatives that could enhance the execution time, namely parallelization. We believe the perception of parallelization is one of the main weaknesses of spreadsheet users. How can this be related to school-age children? In this dissertation, Chapter 6 shows that the difficulty to understand the sequential execution of programs as one of the frequent misconceptions among school students. In other studies, Weintrop *et al.* conclude that there is a scarce in computing material aimed at teaching primary school students program synchronization, parallel- and event-based programming. Even in high school courses, the focus is on *"non-temporal aspects of programming (like algorithms, sequencing, state, etc.)"*.

Spatial awareness in spreadsheets: In Chapter 5, we observed how preschoolers struggle with the spatial characteristic of objects when controlling robotic and game characters. There is a growing body of knowledge on understanding the relation between spatial awareness and developing computation skills. However, there is still more to be done to precisely understand how *"spatial skills training influences computer science learning"*. When we compare that to spreadsheet users, spatial awareness is part of the difficulty of working in a visual-based environment like spreadsheets. Various studies aimed at helping spreadsheet end-user programmers comprehend their spreadsheet models by inferring the spatial relations between cells and labels. In this dissertation, Chapter 3 shows the challenges the end-users face to extract information out of semi-structured data. To form a meaningful piece of information in that case, it requires moving on the horizontal and vertical axes, in addition to considering the hierarchy of labels as well. This is a typical everyday process that spreadsheet end-users perform which makes the spatial inference a mandatory skill to have.

DIFFERENCES

Learning focus: Most programming environments that aim at children as their audience have learning programming as one goal. They integrate this goal in their environments by providing visual-based elements. In Scratch, for example, there are the blocks to eliminate the syntax difficulty, colors to differentiate between concepts, tutorials accessible starting from the same development canvas and the possibility to look at and use shared projects. Children also are in the phase where learning is encouraged by the surrounding environment: parents at home and educators at schools encourage and allow time for learning activities including in or via programming. For spreadsheet users, this is not the case. We highlighted in the introduction of this dissertation how end-user programmers are *always* in a state of cost-benefit evaluation to decide whether they perform a programming activity. Part of this evaluation analysis concerns the time and effort they need to look for relevant and necessary conceptual and tool knowledge. They need to strike a balance between learning and the responsibilities of their work. Besides, spreadsheet IDEs do not provide learning elements from within, focusing instead on letting users add new functionalities.

Table 8.1. End-user learning barriers from Ko et. al and whether they apply to spreadsheet users and school-age children

Learning barrier	Brief description	Spreadsheet users	School-age children
Design barrier	The difficulties in using programming problem-solving methods and concepts	X	X
Selection barrier	While end-user know what to do, they lack the knowledge of what software, library and function to use		X
Coordination barrier	The difficulty to understand how various programming components can be combined to achieve one higher-level goal	X	
Use barrier	The difficulties stemming from ambiguous and vague User-Interface components	X	
Understanding barrier	Invalid assumptions or misconceptions held before working on a program		X
Information barrier	The difficulty to find useful information that could help in verifying or writing programs	X	

8.3 FUTURE WORK

While performing this research and writing this dissertation two areas emerged where researchers should put more focus on in future.

Computational Thinking Development: Research shows that learning to program contributes to the development of Computational Thinking. With the introduction of programming to younger age, there are initial research that aims at investigating the development of Computational Thinking learning trajectoriesit [184–186]. However, more research is needed on understanding what programming concepts to introduce at what age and by using what programming environment. This is important since any inappropriate match to children's cognitive abilities risks building negative and wrong conceptions about programming and computer science at an early age. Despite being provided with good intentions, a lot of programming teaching at schools is ad-hoc and lacks the underlying support of previous research results. Teachers, therefore, require clear guidelines of how to design and introduce programming materials that link to a trajectory of conceptual development per age group.

Computational Thinking Transferability: While computational skills are naturally developed when learning how to program, we still lack the knowledge of how Computational Thinking skills transfer to other domains and to contexts other than programming. Earlier research indicates that such transfer requires explicit scaffolding and instruction in the targeted context [169]. This is an important aspect, especially for end-user programming research. How would learning Scratch affect the development of spreadsheets, or other end-user programming skills? And to what extent can we use spreadsheets to develop Computational Thinking skills? We foresee here an opportunity to introduce spreadsheets to younger students as early as the beginning of high-school (12-13 years of age). That brings, however, the need for more research on, for example, the linkage between specific Computational Thinking skills and problem-solving in spreadsheets, and the development of professional teachers skills in spreadsheet programming.

Summary

The goal of this dissertation is to explore, understand, and mitigate when possible, the challenges that end-users face when creating their software programs. To gain a wider perspective of the challenges, we investigated two groups of end-users: spreadsheet users and school-age children learning to program.

In Chapters 2 and 3, we worked with spreadsheet users in two case studies. We identified two challenges related to the long execution time of some spreadsheets and the difficulty to extract semi-structured data from a set of spreadsheets. Subsequently, we followed software engineering-based methods to provide end-users with software solutions. We found that the parallelization of a spreadsheet load reduced the overall execution time in the first case. For the data extraction, we implemented a procedure based on an algorithm that performed previously well on standard tabular data. We modified the algorithm in a way that allowed the semi-structured crosstables, in particular, to be identified and extracted.

We followed the work on spreadsheets with the exploration of challenges that school-age children face while learning to program.

While working on spreadsheets with end-users, the expected outcome was to provide concrete solutions. The work with school-age children was however different and focused on exploring the difficulties and understanding their origins from one side and impact on their future interactions with programming.

Looking at preschoolers (Chapter 5), we observed children between 4 and 6 years old while performing programming activities within robotic and gaming contexts. We found that children execute sorting and classification tasks with reasonable ease. However, they had difficulties related to understanding the spatial characteristics of the objects they move around.

For the older age groups, we started by studying the identifiers' naming practices that Scratch programmers follow (Chapter 4). We, therefore, analyzed the variable and procedure names in 250,000 Scratch projects. We found that Scratch programmers use longer names compared to other mainstream programming languages such as Java. However, Scratch programmers still use short and single-letter variable names, where the most frequent names are *x*, *y*, and *i*. We also found that identifier names in Scratch projects quite often have features unique to Scratch such as having white spaces: 44% of the unique variable names and 34% of the projects in the dataset include at least one space. This familiarity with features that are specific to Scratch could make the transition to mainstream languages more difficult in the future.

We continued our focus on Scratch with the research work to identify programming misconceptions that school-age children possibly hold (Chapter 6). Programming misconceptions are partial and incorrect understanding of how some aspects of programming work. Misconceptions, in general, are known to be persistent and have a long effect. We found that school-age children indeed hold programming misconceptions. The top occurring three misconceptions belong to various programming aspects: the difficulty to understand the sequentiality of program execution, the difficulty to understand that a variable holds one value at a time, and the difficulty to understand the effect of input calls on programs' execution.

Beginning with programming for young children usually begins with visual-based programming languages to avoid difficult syntax. In the last study in this dissertation (Chapter 7), we explored how school-age children read code in Python, and followed it with a controlled experiment to assess the effects of reading code aloud on comprehension. The study found that children have difficulties reading some symbols and keywords. While there was no difference between the two groups in the score of the understanding cognitive ability, we found that the group who read code aloud scored better in programming questions that measured the remembering cognitive ability. Although the remembering level is the lowest in complexity within the learning cognitive skills, the results show that the syntax difficulty, in particular, could be lowered or even eliminated when the children follow code reading training in class.

Finally, when looking at the two groups we aimed at making a connection between younger children and older spreadsheet users. We provided, in Chapter 8, some similarities and differences based on the results as well as what we observed during the studies. Children are generally looked at as the future workforce. Current and future jobs increasingly demand Computational Thinking skills, and this demand is only going to grow as we become more and more relying on software technology. Therefore, children of today are expected to become the end-users of tomorrow. Therefore, it is our responsibility that we critically review how children learn to program and how they build their early software. While doing so, the aim should be to align programming activities by children cognitive abilities and to develop the global Computational Thinking skills within contexts other than programming. In this way not only we open the space for more creativity at work within the end-user paradigm but also avoid early challenges that could cause early misconceptions and withdrawal from computer science and technology-related topics.

SAMENVATTING

Het doel van dit proefschrift is de uitdagingen zijn waarmee eindgebruikers worden geconfronteerd bij het maken van hun softwareprogramma's te verkennen, te begrijpen en te beperken. Om een breder perspectief op de uitdagingen te krijgen, hebben we twee groepen eindgebruikers onderzocht: spreadsheetgebruikers en schoolgaande kinderen die leren programmeren. In hoofdstuk 2 en 3 werkten we met spreadsheetgebruikers in twee casestudies. We hebben twee uitdagingen geïdentificeerd met betrekking tot de lange berekeningstijd van sommige spreadsheets en de moeilijkheid om semi-gestructureerde gegevens uit een groep spreadsheets te extraheren. Vervolgens pasten we methodes uit de software engineering toe om eindgebruikers softwareoplossingen te bieden. We hebben geconstateerd dat de parallellisatie van een spreadsheetbelasting de algehele berekeningstijd in het eerste geval verkortte. Voor de data-extractie hebben we een procedure geïmplementeerd op basis van een algoritme dat goed presteerde op reguliere tabellen. We hebben het algoritme zodanig aangepast dat vooral de semi-gestructureerde kruistabellen konden worden geïdentificeerd en geëxtraheerd. We volgden dit werk over spreadsheets op, met het verkennen van uitdagingen waarmee kinderen in de schoolgaande leeftijd te maken krijgen tijdens het leren programmeren.

Tijdens het werken aan spreadsheets met eindgebruikers was het verwachte resultaat om concrete oplossingen te bieden. Het werk met schoolgaande kinderen was echter anders en was gericht op het verkennen van de moeilijkheden en het van één kant begrijpen van hun oorsprong en impact op hun toekomstige interacties met programmeren. Kijkend naar kleuters (hoofdstuk 5), zagen we kinderen tussen de 4 en 6 jaar oud tijdens het uitvoeren van programmeeractiviteiten binnen robot- en spellencontexten. We hebben gevonden dat kinderen met redelijk gemak sorteer- en classificatietaken uitvoeren. Zij echter had problemen met het begrijpen van de ruimtelijke kenmerken van de objecten die ze verplaatsen in de omgeving van. Voor de oudere leeftijdsgroepen zijn we begonnen met het bestuderen van de naamgevingspraktijken van de identificatoren die Scratch-programmeurs volgen (hoofdstuk 4). Daarom hebben we de namen van variabelen en procedures geanalyseerd in 250.000 Scratch-projecten. We hebben geconstateerd dat Scratch-programmeurs langere namen gebruiken in vergelijking met andere reguliere programmeertalen zoals Java. Scratch-programmeurs gebruiken echter nog steeds korte variabelenamen soms zelfs maar met 1 letter, waarbij de meest voorkomende namen x, y en i zijn. We hebben ook geconstateerd dat identificatienamen in Scratch-projecten vaak eigenschappen hebben die uniek zijn voor Scratch, zoals witte spaties: 44% van de unieke variabelenamen en 34% van de projecten in de gegevensset bevatten ten minste één spatie. Deze bekendheid met functies die specifiek zijn voor Scratch kan de overgang naar reguliere talen in de toekomst moeilijker maken. We hebben onze focus op Scratch voortgezet met het onderzoek om programmeer misconcepties te identificeren die kinderen in de schoolgaande leeftijd mogelijk hebben (hoofdstuk 6). Programmeerfouten zijn gedeeltelijk en onjuist begrip van hoe sommige aspecten van programmeren werken. Misvattingen staan in het algemeen bekend als persistent en hebben een lang effect. We ontdekten dat kinderen van schoolgaande leeftijd inderdaad misconcepties over programmeren hebben. De drie meest voorkomende misconcepties behoren tot verschillende programmeeraspecten: het begrijpen van de volgordelijkheid van programma's, de moeilijkheid om te begrijpen dat een variabele maar één waarde per keer heeft, en de moeilijkheid om het effect begrijpen van invoercommando’s op de uitvoering van programma's.

Beginnen met programmeren voor jonge kinderen begint meestal met visueel gebaseerde programmeertalen om moeilijke syntaxis te voorkomen. In de laatste studie in dit proefschrift (hoofdstuk 7) hebben we onderzocht hoe schoolgaande kinderen code lezen in Python en deze hebben gevolgd met een gecontroleerd experiment om de effecten van hardop lezen van code op het begrip te beoordelen. Uit het onderzoek bleek dat kinderen moeite hebben bepaalde symbolen en trefwoorden te lezen. Hoewel er geen verschil was tussen de twee groepen in de score van het inzicht in cognitief vermogen, vonden we dat de groep die code hardop voorleest beter scoorde in programmeervragen die de cognitieve vaardigheid ‘onthouden’ meten. Hoewel het herinneringsniveau het laagste is in complexiteit binnen de leercognitieve vaardigheden, tonen de resultaten aan dat met name de moeilijkheden met syntax kunnen worden verlaagd of zelfs geëlimineerd wanneer de kinderen in de klas oefenen met het lezen van code volgen.

Ten slotte, als we naar de twee groepen keken, wilden we een verband leggen tussen jongere kinderen en oudere spreadsheetgebruikers. We hebben in hoofdstuk 8 enkele overeenkomsten en verschillen gegeven op basis van de resultaten en wat we tijdens de studies hebben waargenomen. Kinderen worden over het algemeen gezien als het toekomstige personeelsbestand. Huidige en toekomstige banen vereisen in toenemende mate Computational Thinking-vaardigheden, en deze vraag zal alleen maar toenemen naarmate we meer en meer afhankelijk worden van softwaretechnologie. Daarom wordt van kinderen van vandaag verwacht dat ze de eindgebruikers van morgen worden. Daarom is het onze verantwoordelijkheid dat we kritisch beoordelen hoe kinderen leren programmeren en hoe ze hun eerste software schrijven. Daarbij zou het doel moeten zijn om de programmeeractiviteiten op elkaar af te stemmen op de cognitieve vaardigheden van kinderen en om de wereldwijde vaardigheden van Computational Thinking te ontwikkelen binnen andere contexten dan programmeren. Op deze manier openen we niet alleen de ruimte voor meer creativiteit in het eindgebruikersparadigma, maar vermijden we ook die uitdagingen die ertoe kunnen leiden dat kinderen misconcepties ontwikkelen of programmeren en technologiegerelateerde onderwerpen helemaal op te geven.

Bibliography

References

[1] Abraham, RobinErwig, M. (2007). Ucheck: A spreadsheet type checker for end users. *Journal of Visual Languages & Computing*, 18(1):71–95.

[2] Abramson, D., Roe, P., Kotler, L., and Mather, D. (2001). Activesheets: Supercomputing with spreadsheets. In *2001 High Performance Computing Symposium (HPC01), Advanced Simulation Technologies Conference*. Citeseer.

[3] Aghaee, S., Blackwell, A. F., Stillwell, D., and Kosinski, M. (2015). Personality and intrinsic motivational factors in end-user programming. *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*.

[4] Aivaloglou, E. and Hermans, F. (2016). How kids code and how we know: An exploratory study on the Scratch repository. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ICER '16. ACM.

[5] Aivaloglou, E., Hermans, F., Moreno-Leon, J., and Robles, G. (2017). A dataset of Scratch programs: Scraped, shaped and scored. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*.

[6] Aman, H., Amasaki, S., Sasaki, T., and Kawahara, M. (2015). Empirical analysis of change-proneness in methods having local variables with long names and comments. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE.

[7] Ambrosio, A. P., Almeida, L. S., Macedo, J., and Franco, A. H. R. (2014). Exploring core cognitive skills of computational thinking. In *Psychology of Programming Interest Group Annual Conference (PPIG) 2014*. Psychology of Programming Interest Group (Online).

[8] Anderson, L. W., Krathwohl, D., and Bloom, B. S. (2001). *A taxonomy for learning, teaching, and assessing: a revision of Bloom's taxonomy of educational objectives*. Longman, 2001.

[9] Anquetil, N. and Lethbridge, T. C. (1998). Assessing the relevance of identifier names in a legacy software system. In *Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '98. IBM Press.

[10] Arends, R. (2012). *Learning to teach*. McGraw-Hill.

[11] Arnaoudova, V., Di Penta, M., and Antoniol, G. (2016). Linguistic antipatterns: what they are and how developers perceive them. *Empirical Software Engineering*, 21(1):104–158.

[12] Arnaoudova, V., Di Penta, M., Antoniol, G., and Guéhéneuc, Y. (2013). A new family of software anti-patterns: Linguistic anti-patterns. In *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE.

[13] Atkinson, P. and Delamont, S. (2011). *SAGE Qualitative Research Methods*. SAGE Publications Ltd.

[14] Avidan, E. and Feitelson, D. G. (2017). Effects of variable names on comprehension an empirical study. In *IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*.

[15] Barendsen, E., Grgurina, N., and Tolboom, J. (2016). A new informatics curriculum for secondary education in the netherlands. In Brodnik, A. and Tort, F., editors, *Informatics in Schools: Improvement of Informatics Knowledge and Perception*. Springer International Publishing.

[16] Barr, D., Harrison, J., and Conery, L. (2011). Computational thinking: A digital age skill for everyone. *Learning & Leading with Technology*, 38(6):20–23.

[17] Barton, C. (2018). *How I wish I'd taught maths*. John Catt Educational Ltd.

[18] Baxter, R. (2007). Enterprise spreadsheet management: A necessary good. In *Proceedings of the EuSpRIG 2007 Symposium*, page 7.

[19] Begel, A. (2005). Programming by voice: A domain-specific application of speech recognition. In *Proceedings of the Speech Technology Track (SpeechTEK) in the Applied Voice Input/Output Society (AVIOS) Conference*. Trafford Publishing.

[20] Begel, A. and Graham, S. (2005). Spoken programs. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. IEEE.

[21] Belohlavek, R. (2008). Introduction to formal concept analysis. *Palacky University, Department of Computer Science, Olomouc*.

[22] Ben-Ari, M. (1998). Constructivism in computer science education. In *Proceedings of the Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '98. ACM.

[23] Ben Nasr, S., Becan, G., Acher, M., Ferreira Filho, J. B., Baudry, B., Sannier, N., and Davril, J.-M. (2015). Matrixminer: a red pill to architect informal product descriptions in the matrix. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*.

[24] Beniamini, G., Gingichashvili, S., Klein-Orbach, A., and Feitelson, D. G. (2017). Meaningful identifier names: the case of single-letter variables. In *IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE.

[25] Berglund, A. and Lister, R. (2010). Introductory programming and the didactic triangle. In *Proceedings of the Twelfth Australasian Conference on Computing Education - Volume 103*, ACE '10. Australian Computer Society, Inc.

[26] Bernhard Ganter, G. S. (2003). *Formal Concept Analysis: Methods and Applications in Computer Science*. TU Dresden (Online). URL: http://www.math.tu-dresden.de/ ganter/cl03/stumme/chapter1_2.pdf.

[27] Bitbrains.nl (2015). De bits en de brains achter uw bedrijfskritische applicaties - bitbrains. *Online. URL: www.bitbrains.nl. Redirects to the new company, Solvinity.*

[28] Blackwell, A. F. (2017). End-user developers - what are they like? *New Perspectives in End-User Development*, pages 121–135.

[29] Bloom, B. (1956). *Taxonomy of Educational Objectives: The Classification of Educational Goals*. D. McKay.

[30] Bochud, M. (2012). Estimating heritability from nuclear family and pedigree data. In Elston, R. C., Satagopan, J. M., and Sun, S., editors, *Statistical Human Genetics: Methods and Protocols*. Humana Press, Totowa, NJ.

[31] Boe, B., Hill, C., Len, M., Dreschler, G., Conrad, P., and Franklin, D. (2013). Hairball: lint-inspired static analysis of Scratch projects. *Proceeding of the 44th ACM technical symposium on Computer science education - SIGCSE '13*.

[32] Boulay, B. D. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):57–73.

[33] Brancheau, J. C. and Brown, C. V. (1993). The management of end-user computing: status and directions. *ACM Computing Surveys*, 25(4):437–482.

[34] Brennan, K., Balch, C., and Chung, M. (2014). *Creative Computing*. Harvard Graduate School of Education.

[35] Bus, A. G., van IJzendoorn, M. H., and Pellegrini, A. D. (1995). Joint book reading makes for success in learning to read: A meta-analysis on intergenerational transmission of literacy. *Review of Educational Research*, 65(1):1–21.

[36] Busch, T. (1995). Gender differences in self-efficacy and attitudes toward computers. *Journal of Educational Computing Research*, 12(2):147–158.

[37] Buse, R. P. L. and Weimer, W. (2010). Learning a metric for code readability. *IEEE Trans. Software Eng.*, 36(4):546–558.

[38] Butler, S., Wermelinger, M., Yu, Y., and Sharp, H. (2010). Exploring the influence of identifier names on code quality: An empirical study. In *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE.

[39] Calao, L. A., Moreno-Leon, J., Correa, H. E., and Robles, G. (2015). Developing mathematical thinking with Scratch. *Design for Teaching and Learning in a Networked World*, pages 17–27.

[40] Canfora, G., Cimitile, A., and Visaggio, C. (2003). Lessons learned about distributed pair programming: what are the knowledge needs to address? In *Proceedings. Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003.*, WET ICE 2003. IEEE.

[41] Caprile, B. and Tonella, P. (1999). Nomen est omen: analyzing the language of function identifiers. In *Sixth Working Conference on Reverse Engineering*. IEEE.

[42] Caprile, B. and Tonella, P. (2000). Restructuring program identifier names. In *Proceedings 2000 International Conference on Software Maintenance*. IEEE.

[43] Caracciolo, A., Chis, A., Spasojević, B., and Lungu, M. (2014). Pangea: A workbench for statically analyzing multi-language software corpora. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE.

[44] CBS. CBS Statline: Het centraal bureau voor de statistiek (CBS) Statline. Online. URL(in Dutch): https://opendata.cbs.nl/statline#/CBS/nl/.

[45] Chambers, C. and Erwig, M. (2010). Reasoning about spreadsheets with labels and dimensions. *Journal of Visual Languages & Computing*, 21(5):249–262.

[46] Chambers, C., Erwig, M., and Luckey, M. (2010). Sheetdiff: A tool for identifying changes in spreadsheets. *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*.

[47] Chambers, C. and Scaffidi, C. (2015). Impact and utility of smell-driven performance tuning for end-user programmers. *Journal of Visual Languages & Computing*, 28:176–194.

[48] Chan, Y. and E.Storey, V. C. (1996). The use of spreadsheets in organizations: Determinants and consequences. *Information & Management*, 31(3):119–134.

[49] Chen, Z. (2016). *Information Extraction on Para-Relational Data*. PhD thesis, The University of Michigan.

[50] Chen, Z. and Cafarella, M. (2013). Automatic web spreadsheet data extraction. *Proceedings of the 3rd International Workshop on Semantic Search Over the Web - SS@ '13*.

[51] Chen, Z. and Cafarella, M. (2014). *A semiautomatic approach for accurate and low-effort spreadsheet data extraction*. Technical report, University of Michigan.

[52] Chen, Z., Cafarella, M., Chen, J., Prevo, D., and Zhuang, J. (2013). Senbazuru: a prototype spreadsheet database management system. *Proceedings of the VLDB Endowment*, 6(12).

[53] Cheney, P. H., Mann, R. I., and Amoroso, D. L. (1986). Organizational factors affecting the success of end-user computing. *Journal of Management Information Systems*, 3(1):65–80.

[54] Chitu, C., Inpuscatu, R. C., and Vintila, A. (2011). The integration of visual basic programming language into physics discipline. *eLearning & Software for Education*.

[55] Clarke, S. and Tobias, A. (1995). Corporate modelling in the UK: A survey. *OR Insight*, 8(3):15–20.

[56] ClusterSeven (2016). Clusterseven annual state of the spreadsheet 2016: The spreadsheet is here to stay (Online). URL: https://www.clusterseven.com/state-of-spreadsheet-report-2016/. Technical report, ClusterSeven.

[57] Cohen, B. H. (2008). *Explaining Psychological Statistics*. John Wiley & Sons, 3 edition.

[58] Coley, J. D. and Tanner, K. D. (2012). Common origins of diverse misconceptions: Cognitive principles and the development of biology thinking. *CBE?Life Sciences Education*, 11(3):209–215. PMID: 22949417.

[59] Costabile, M. F., Mussio, P., Provenza, L. P., and Piccinno, A. (2008). Advanced visual systems supporting unwitting eud. *Proceedings of the working conference on Advanced visual interfaces - AVI '08*.

[60] Coster, N., Leon, L., Kalbers, L., and Abraham, D. (2011). Controls over spreadsheets for financial reporting in practice. In *Proceedings of EuSpRIG 2011 Conference "Spreadsheet Governance Policy and Practice" ISBN: 978-0-9566256-9-4*. arXiv preprint arXiv:1111.6887.

[61] Croll, G. J. (2005a). The importance and criticality of spreadsheets in the city of london. *Proc. European Spreadsheet Risks Int. Grp. (EuSpRIG) 2005 82-92 ISBN:1-902724-16-X*.

[62] Croll, G. J. (2005b). The importance and criticality of spreadsheets in the city of London. In *In Proceedings of European Spreadsheet Risk Interest Group 2005 Conference, EuSpRIG, London, UK*.

[63] Cunha, J., Erwig, M., and Saraiva, J. (2010). Automatically inferring classsheet models from spreadsheets. *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*.

[64] Cunha, J., Saraiva, J., and Visser, J. (2008). From spreadsheets to relational databases and back. *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation - PEPM '09*.

[65] Cypher, A. and Smith, D. C. (1995). Kidsim: end user programming of simulations. *Conference companion on Human factors in computing systems - CHI '95*.

[66] Dajsuren, Y., van den Brand, M. G. J., Serebrenik, A., and Roubtsov, S. (2013). Simulink models are also software: Modularity assessment. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '13. ACM.

[67] Dann, W., Cooper, S., and Pausch, R. (2006). *Learning to program with Alice*. Prentice Hall.

[68] Dann, W., Cosgrove, D., Slater, D., Culyba, D., and Cooper, S. (2012). Mediated transfer: Alice 3 to Java. In *SIGCSE*. ACM.

[69] Dasgupta, S. and Hill, B. (2017). Learning to code in localized programming languages. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, L@S '17. ACM.

[70] Davis, R., Kafai, Y., Vasudevan, V., and Lee, E. (2013). The education arcade: Crafting, remixing, and playing with controllers for Scratch games. In *International Conference on Interaction Design and Children*. ACM.

[71] Deißenböck, F. and Pizka, M. (2005). Concise and consistent naming [software system identifier naming]. In *13th International Workshop on Program Comprehension (IWPC'05)*. IEEE.

[72] Deschambault, R. (2011). Thinking-aloud as talking-in-interaction: Reinterpreting how l2 lexical inferencing gets done. *Language Learning*, 62(1):266–301.

[73] Desilets, A. (2001). Voicegrip: A tool for programming-by-voice. *International Journal of Speech Technology*, 4(2):103–116.

[74] Doukakis, D., Grigoriadou, M., and Tsaganou, G. (2007). Understanding the programming variable concept with animated interactive analogies. In *Proceedings of the 8th Hellenic European Research on Computer Mathematics & its Applications*, HERCMA '07.

[75] Du Boulay, B., O'Shea, T., and Monk, J. (1999). The black box inside the glass box: presenting computing concepts to novices. *International Journal of Human-Computer Studies*, 51(2):265–277.

[76] Edwards, E., Elliott, J., and Bruckman, A. (2001). Aquamoose 3d: Math learning in a 3d multi-user virtual world. In *CHI '01 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '01. ACM.

[77] EuSpRIG. European spreadsheet risks interest group - spreadsheet risk management and solutions conference. Online. URL: http://www.eusprig.org/.

[78] Fateman, R. (1998). How can we speak math. *Journal of Symbolic Computation*, 25(2).

[79] Fay, A. L. and Klahr, D. (1996). Knowing about guessing and guessing about knowing: Preschoolers' understanding of indeterminacy. *Child Development*, 67(2):689.

[80] Ferry, T. R., Fouad, N. A., and Smith, P. L. (2000). The role of family context in a social cognitive model for career-related choice behavior: A math and science perspective. *Journal of Vocational Behavior*, 57(3):348–364.

[81] Fessakis, G., Gouli, E., and Mavroudi, E. (2013). Problem solving by 5-6 years old kindergarten children in a computer programming environment: A case study. *Computers & Education*, 63:87–97.

[82] Fitzgerald, S., Simon, B., and Thomas, L. (2005). Strategies that students use to trace code: An analysis based in grounded theory. In *Proceedings of the First International Workshop on Computing Education Research*, ICER '05. ACM.

[83] Fowler, M., Beck, K., Brant, J., and Opdyke, W. (1999). *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1 edition.

[84] Francioni, J. M. and Smith, A. C. (2002). Computer science accessibility for students with visual disabilities. In *ACM SIGCSE Bulletin*, volume 34. ACM.

[85] Fung, P., Brayshaw, M., Du Boulay, B., and Elsom-Cook, M. (1990). Towards a taxonomy of novices' misconceptions of the prolog interpreter. *Instructional Science*, 19(4-5):311–336.

[86] Gabriel, K. R. (1969). Simultaneous test procedures—some theory of multiple comparisons. *The Annals Mathematical Statistics*, 40(1):224–250.

[87] Glassman, E. L., Fischer, L., Scott, J., and Miller, R. C. (2015). Foobaz: Variable name feedback for student code at scale. In *Annual ACM Symposium on User Interface Software & Technology*. ACM.

[88] Goldman, K., Gross, P., Heeren, C., Herman, G., Kaczmarczyk, L., Loui, M. C., and Zilles, C. (2008). Identifying important and difficult concepts in introductory computing courses using a delphi process. *ACM SIGCSE Bulletin*, 40(1):256.

[89] Greher, G. R. and Heines, J. M. (2014). *Computational Thinking in Sound*. Oxford University Press.

[90] Griddynamics.com. Blog | grid dynamics. Online. URL: http://www.griddynamics.com/solutions/excel-hpc.html.

[91] Grossman, T. A. (2002). *Spreadsheet Engineering: A Research Framework*.

[92] Grover, S., Pea, R., and Cooper, S. (2015). Designing for deeper learning in a blended computer science course for middle school students. *Computer Science Education*, 25(2):199–237.

[93] Hale, A. D., Hawkins, R. O., Sheeley, W., Reynolds, J. R., Jenkins, S., Schmitt, A. J., and Martin, D. A. (2010). An investigation of silent versus aloud reading comprehension of elementary students using maze assessment procedures. *Psychology in the Schools*, 48(1):4–13.

[94] Hattie, J. (2009). *Visible learning*. Routledge.

[95] Hermans, F. (2013). *Analyzing and visualizing spreadsheets*. PhD thesis, Technische Universiteit Delft.

[96] Hermans, F. and Aivaloglou, E. (2016). Do code smells hamper novice programming? a controlled experiment on Scratch programs. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*.

[97] Hermans, F. and Aivaloglou, E. (2017). Teaching software engineering principles to k-12 students: A MOOC on Scratch. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering and Education Track*, ICSE-SEET '17. IEEE Press.

[98] Hermans, F. and Dig, D. (2014). Bumblebee: a refactoring environment for spreadsheet formulas. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*.

[99] Hermans, F., Jansen, B., Roy, S., Aivaloglou, E., Swidan, A., and Hoepelman, D. (2016). Spreadsheets are code: An overview of software engineering approaches applied to spreadsheets. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE.

[100] Hermans, F. and Murphy-Hill, E. (2015). Enron's spreadsheets and related emails: A dataset and analysis. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*.

[101] Hermans, F., Pinzger, M., and van Deursen, A. (2010). Automatically extracting class diagrams from spreadsheets. In *Proceedings of the 24th European Conference on Object-oriented Programming ECOOP'10*. Springer Berlin Heidelberge.

[102] Hermans, F., Pinzger, M., and van Deursen, A. (2011). Supporting professional spreadsheet users by generating leveled dataflow diagrams. *Proceeding of the 33rd international conference on Software engineering - ICSE '11*.

[103] Hermans, F., Pinzger, M., and van Deursen, A. (2014). Detecting and refactoring code smells in spreadsheet formulas. *Empirical Software Engineering*, 20(2):549–575.

[104] Hermans, F., Swidan, A., and Aivaloglou, E. (2018). Code phonology: an exploration into the vocalization of code. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18. ACM.

[105] Hill, B. M. and Monroy-Hernández, A. (2013). The remixing dilemma. *American Behavioral Scientist*, 57(5):643–663.

[106] Hofmeister, J., Siegmund, J., and Holt, D. V. (2017). Shorter identifier names take longer to comprehend. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE.

[107] Hong, H., Wang, J., and Moghadam, S. H. (2016). K-12 computer science education across the U.S. In Brodnik, A. and Tort, F., editors, *Informatics in Schools: Improvement of Informatics Knowledge and Perception*. Springer International Publishing.

[108] Høst, E. W. (2011). *Meaningful Method Names*. PhD thesis, University of Oslo, Norway.

[109] Jansen, B. and Hermans, F. (2015). Code smells in spreadsheet formulas revisited on an industrial dataset. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE.

[110] Jenkins, T. (2002). On the difficulty of learning to program. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, volume 4. LTSN-ICS.

[111] Joet, G., Usher, E. L., and Bressoux, P. (2011). Sources of self-efficacy: An investigation of elementary school students in france. *Journal of Educational Psychology*, 103(3):649–663.

[112] Johansson, R. (2007). On case study methodology. *Open House International*, 32(3):48.

[113] John, O. P. and Srivastrava, S. (1999). *The Big-Five Trait Taxonomy: History, Measurement, and Theoretical Perspectives*. Guilford Press, 2 edition.

[114] Jones, M. and Scaffidi, C. (2011). Obstacles and opportunities with using visual and domain-specific languages in scientific programming. *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*.

[115] Jongeling, R., Sarkar, P., Datta, S., and Serebrenik, A. (2017). On negative results when using sentiment analysis tools for software engineering research. *Empirical Software Engineering*, 22(5):2543–2584.

[116] Kaloti-Hallak, F., Armoni, M., and Ben-Ari, M. M. (2015). Students' attitudes and motivation during robotics activities. In *Proceedings of the Workshop in Primary and Secondary Computing Education*, WiPSCE '15. ACM.

[117] Kato, T., Kambayashi, Y., and Kodama, Y. (2016). Data mining of students' behaviors in programming exercises. In Uskov, V. L., Howlett, R. J., and Jain, L. C., editors, *Smart Education and e-Learning 2016*. Springer International Publishing.

[118] Kelleher, C. and Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, 37(2):83–137.

[119] Kelleher, C. and Pausch, R. (2006). Lessons learned from designing a programming system to support middle school girls creating animated stories. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*. IEEE.

[120] Kelleher, C., Pausch, R., and Kiesler, S. (2007). Storytelling alice motivates middle school girls to learn computer programming. *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '07*.

[121] Keller, J. M. (1987). Development and use of the arcs model of instructional design. *Journal of Instructional Development*, 10(3):2–10.

[122] Kery, M. B., Radensky, M., Arya, M., John, B. E., and Myers, B. A. (2018). The story in the notebook: Exploratory data science using a literate programming tool. In *Proceedings of the 2018 Conference on Human Factors in Computing Systems*, CHI '18. ACM.

[123] Klopfer, E. (2003). Technologies to support the creation of complex systems models?using starlogo software with students. *Biosystems*, 71(1-2):111–122.

[124] Ko, A. J., Myers, B., Rosson, M. B., Rothermel, G., Shaw, M., Wiedenbeck, S., Abraham, R., Beckwith, L., Blackwell, A., and Burnett, M. e. a. (2011). The state of the art in end-user software engineering. *ACM Computing Surveys*, 43(3):1–44.

[125] Ko, A. J., Myers, B. A., and Aung, H. H. Six learning barriers in end-user programming systems. *2004 IEEE Symposium on Visual Languages - Human Centric Computing*.

[126] Kölling, M., Brown, N. C. C., and Altadmri, A. (2015). Frame-based editing: Easing the transition from blocks to text-based programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education*, WiPSCE '15. ACM.

[127] Konietschke, F., Hothorn, L. A., and Brunner, E. (2012). Rank-based multiple test procedures and simultaneous confidence intervals. *Electronic Journal of Statistics*, 6:738–759.

[128] Konietschke, F., Placzek, M., Schaarschmidt, F., and Hothorn, L. (2015). nparcomp: An R software package for nonparametric multiple comparisons and simultaneous confidence intervals. *Journal of Statistical Software*, 64(9):1–17.

[129] Kragler, S., Martin, L., and Schreier, V. (2015). Investigating young children's use of reading strategies: A longitudinal study. *Reading Psychology*, 36(5):445–472.

[130] Kruck, S. E., Maher, J. J., and Barkhi, R. (2003). Framework for cognitive skill acquisition and spreadsheet training. *Journal of Organizational and End User Computing*, 15(1):20–37.

[131] Kurvinen, E., Hellgren, N., Kaila, E., Laakso, M., and Salakoski, T. (2016). Programming misconceptions in an introductory level programming course exam. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16. ACM.

[132] Kyriacou, C. (2014). Filtered - microsoft excel versus apple's numbers, who prevails? Online. URL: https://excelwithbusiness.com/blog/microsoft-excel-versus-apples-numbers-who-prevails.

[133] Lawrie, D., Morrell, C., Feild, H., and Binkley, D. (2007). Effective identifier names for comprehension and memory. *ISSE*, 3(4):303–318.

[134] LAWSON, B., BAKER, K., POWELL, S., and FOSTERJOHNSON, L. (2009). A comparison of spreadsheet users with different levels of experience? *Omega*, 37(3):579–590.

[135] Lister, R. (2006). Computer science teachers as amateurs, students and researchers. In *Proceeding of the Fifth Koli Calling Conference on Computer Science Education, Baltic Sea Conference on Computing Education Research*. Turku Centre for Computer Science.

[136] Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B., and Thomas, L. (2004). A multinational study of reading and tracing skills in novice programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '04. ACM.

[137] Lungu, M. and Kurs, J. (2013). On planning an evaluation of the impact of identifier names on the readability and quality of smalltalk programs. In *2013 2nd International Workshop on User Evaluations for Software Engineering Researchers (USER)*. IEEE.

[138] Ma, L. (2007). *Investigating and improving novice programmers mental models of programming concepts*. Phd thesis, University of Strathclyde, UK.

[139] Ma, L., Ferguson, J., Roper, M., and Wood, M. (2011). Investigating and improving the models of programming concepts held by novice programmers. *Computer Science Education*, 21(1):57–80.

[140] Magnani, M., Rizopoulos, N., Mc.Brien, P., and Montesi, D. (2005). *Schema Integration Based on Uncertain Semantic Mappings*, pages 31–46. Springer Berlin Heidelberg, Berlin, Heidelberg.

[141] Magnenat, S., Shin, J., Riedo, F., Siegwart, R., and Ben-Ari, M. (2014). Teaching a core cs concept through robotics. In *Proceedings of the 2014 Conference on Innovation &#38; Technology in Computer Science Education*, ITiCSE '14. ACM.

[142] Manovich, L., Malina, R. F., and Cubitt, S. (2001). *The language of new media*. MIT press.

[143] Martin, L. E. and Kragler, S. (2011). Becoming a self-regulated reader: A study of primary-grade students' reading strategies. *Literacy Research and Instruction*, 50(2):89–104.

[144] Mascolo, M. F. (2015). Neo-piagetian theories of cognitive development. *International Encyclopedia of the Social & Behavioral Sciences*, pages 501–510.

[145] Matsuzawa, Y., Ohata, T., Sugiura, M., and Sakai, S. (2015). Language migration in non-CS introductory programming through mutual language translation environment. In *SIGCSE*. ACM.

[146] McCallum, R. S., Sharp, S., Bell, S. M., and George, T. (2004). Silent versus oral reading comprehension and efficiency. *Psychology in the Schools*, 41(2):241–246.

[147] McLeod, S. (2015). Preoperational stage. Online. URL: https://www.simplypsychology.org/preoperational.html.

[148] Meerbaum-Salant, O., Armoni, M., and Ben-Ari, M. M. (2010). Learning computer science concepts with Scratch. In *Proceedings of the Sixth International Workshop on Computing Education Research*, ICER '10. ACM.

[149] Mitchell, P. and Ziegler, F. (2013). *Fundamentals of Developmental Psychology*. Taylor & Francis Ltd, 1 edition.

[150] Moreno, J. and Robles, G. (2014). Automatic detection of bad programming habits in Scratch: A preliminary study. In *Frontiers in Education Conference*. IEEE.

[151] Moreno-Leon, J. and Robles, G. (2015a). Analyze your scratch projects with dr. scratch and assess your computational thinking skills. In *Scratch Conference*, pages 12–15.

[152] Moreno-Leon, J. and Robles, G. (2015b). Dr.Scratch: a web tool to automatically evaluate Scratch projects. *Proceedings of the Workshop in Primary and Secondary Computing Education - WiPSCE '15*.

[153] Moreno-León, J., Román-González, M., Harteveld, C., and Robles, G. (2017). On the automatic assessment of computational thinking skills: A comparison with human experts. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '17. ACM.

[154] Morgado, L., Cruz, M., and Kahn, K. (2006). Radia perlman: A pioneer of young children computer programming. *Current developments in technology-assisted education. Proceedings of m-ICTE*, III:1903–1908.

[155] Morgado, L. C. (2005). *Framework for computer programming in preschool and kindergarten*. PhD thesis, Universidade de Tras-os-Montes e Alto Douro.

[156] Nadiminti, K., Chiu, Y.-F., Teoh, N., Luther, A., Venugopal, S., Buyya, R., and Street, B. (2004). Excelgrid: A .net plug-in for outsourcing excel spreadsheet workload to enterprise and global grids. In *Proceedings of the 12th International Conference on Advanced Computing and Communication (ADCOM 2004*.

[157] Nelson, R. R. and Cheney, P. H. (1987). Training end users: An exploratory study. *MIS Quarterly*, 11(4):547.

[158] Neubert, K. and Brunner, E. (2007). A studentized permutation test for the non-parametric Behrens-Fisher problem. *Computational Statistics & Data Analysis*, 51(10):5192–5204.

[159] Niess, M., Sadri, P., and Lee, K. (2007). Dynamic spreadsheets as learning technology tools: Developing teachers' technology pedagogical content knowledge (TPCK). *American Educational Research Association*.

[160] NWO (2015). From ás soon as possibleťo ímmediateÓpen Access. Online. URL: https://www.nwo.nl/en/news-and-events/news/2015/from-as-soon-as-possible-to-immediate-open-access.html.

[161] Orlando, A. and Politano, M. (2010). Pension funds risk analysis: stochastic solvency in a management perspective. *Problems and Perspectives in Management*, Volume 8(Issue 3).

[162] Pane, J. F. (1998). Designing a programming system for children with a focus on usability. *CHI 98 conference summary on Human factors in computing systems - CHI '98.*

[163] Panko, R. R. (2006). Spreadsheets and sarbanes-oxley: Regulations, risks, and control frameworks. *Communications of the Association for Information Systems*, Vol. 17 Article 29.

[164] Panko, R. R. (2007). Recommended practices for spreadsheet testing. *arXiv preprint arXiv:0712.0109.*

[165] Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc.

[166] Papert, S. and Harel, I. (1991). *Situating Constructionism in "Constructionism" ISBN 0893917869.* Ablex.

[167] Parnin, C. (2011). Subvocalization - toward hearing the inner thoughts of developers. In *2011 IEEE 19th International Conference on Program Comprehension*. IEEE.

[168] Pea, R. D. (1986). Language-independent conceptual ?bugs? in novice programming. *Journal of Educational Computing Research*, 2(1):25–36.

[169] Perkins, D. N. and Salomon, G. (1989). Are cognitive skills context-bound? *Educational Researcher*, 18(1):16–25.

[170] Perrone-Bertolotti, M., Rapin, L., Lachaux, J.-P., Baciu, M., and L?venbruck, H. (2014). What is that little voice inside my head? inner speech phenomenology, its role in cognitive performance, and its relation to self-monitoring. *Behavioural Brain Research*, 261:220–239.

[171] Peterson, J. and Haynes, G. (2017). Integrating computer science into music education. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17. ACM.

[172] Petre, M. and Blackwell, A. F. (2007). Children as unwitting end-user programmers. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007).*

[173] Piaget, J. (1964). Part i: Cognitive development in children: Piaget development and learning. *Journal of Research in Science Teaching*, 2(3):176–186.

[174] Pichitlamken, J., Kajkamhaeng, S., Uthayopas, P., and Kaewpuang, R. (2010). High performance spreadsheet simulation on a desktop grid. *Journal of Simulation*, 5(4):266–278.

[175] Price, D. E., Dahlstrom, D. A., Newton, B., and Zachary, J. L. (2002). Off to see the wizard: using a "wizard of oz" study to learn how to design a spoken language interface for programming. In *32nd Annual Frontiers in Education*. IEEE.

[176] Price, T. W. (2018). isnap: Automatic hints and feedback for block-based programming. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education - SIGCSE '18*.

[177] Prior, S. M. and Welling, K. A. (2001). "read in your head": A vygotskian analysis of the transition from oral to silent reading. *Reading Psychology*, 22(1):1–15.

[178] Purswell, K. E. and Ray, D. C. (2014). Research with small samples. *Counseling Outcome Research and Evaluation*, 5(2):116–126.

[179] Putnam, R., Sleeman, D., Baxter, J., and Kuspa, L. (1986). A summary of misconceptions of high school basic programmers. *Journal of Educational Computing Research*, 2(4):459–472.

[180] Rajalingham, K., Chadwick, D., Knight, B., and Edwards, D. (2000). Quality control in spreadsheets: a software engineering-based approach to spreadsheet development. *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*.

[181] Rastogi, A. and Nagappan, N. (2016). Forking and the sustainability of the developer community participation - an empirical investigation on outcomes and reasons. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE.

[182] Raymond, E. S. (1996). *The New Hacker's Dictionary*. MIT Press.

[183] Rayner, K., Foorman, B. R., Perfetti, C. A., Pesetsky, D., and Seidenberg, M. S. (2002). How should reading be taught? *Scientific American*, 286(3):84–91.

[184] Rich, K. M., Binkowski, T. A., Strickland, C., and Franklin, D. (2018). Decomposition: A k-8 computational thinking learning trajectory. In *Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER)*, ICER '18, pages 124–132, New York, NY, USA. ACM.

[185] Rich, K. M., Strickland, C., Binkowski, T. A., and Franklin, D. (2019). A k-8 debugging learning trajectory derived from research literature. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE)*, SIGCSE '19, pages 745–751, New York, NY, USA. ACM.

[186] Rich, K. M., Strickland, C., Binkowski, T. A., Moran, C., and Franklin, D. (2017). K-8 learning trajectories derived from research literature: Sequence, repetition, conditionals. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER)*, ICER '17, pages 182–190, New York, NY, USA. ACM.

[187] Robles, G., Moreno-León, J., Aivaloglou, E., and Hermans, F. (2017). Software clones in Scratch projects: on the presence of copy-and-paste in computational thinking learning. In *2017 IEEE 11th International Workshop on Software Clones (IWSC)*. IEEE.

[188] Rother, K. (2017). Cleaning up code. In *Pro Python Best Practices*. Apress, Berkeley, CA.

[189] Rothermel, K. J., Cook, C. R., Burnett, M. M., Schonfeld, J., Green, T. R. G., and Rothermel, G. (2000). Wysiwyt testing in the spreadsheet paradigm. *Proceedings of the 22nd international conference on Software engineering - ICSE '00.*

[190] Roy, S., Hermans, F., Aivaloglou, E., Winter, J., and Deursen, A. v. (2016). Evaluating automatic spreadsheet metadata extraction on a large set of responses from MOOC participants. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE.

[191] Sáez-López, J.-M., Román-González, M., and Vázquez-Cano, E. (2016). Visual programming languages integrated across the curriculum in elementary school: A two year case study using Scratch in five schools. *Computers & Education*, 97:129–141.

[192] Sannier, N., Acher, M., and Baudry, B. (2013). From comparison matrix to variability model: The wikipedia case study. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE.

[193] Santoro, L. E., Chard, D. J., Howard, L., and Baker, S. K. (2008). Making the very most of classroom read-alouds to promote comprehension and vocabulary. *The Reading Teacher*, 61(5):396–408.

[194] Scaffidi, C. (2017). Workers who use spreadsheets and who program earn more than similar workers who do neither. *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).*

[195] Scalabrino, S., Vásquez, M. L., Poshyvanyk, D., and Oliveto, R. (2016). Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE.

[196] Scanniello, G. and Risi, M. (2013). Dealing with faults in source code: Abbreviated vs. full-word identifier names. In *2013 IEEE International Conference on Software Maintenance*. IEEE.

[197] Schmeier, M. and Bijman, R. (2017). *Effectief rekenonderwijs op de basisschool (in Dutch)*. Uitgeverij Pica, 1st edition edition.

[198] Seiter, L. and Foreman, B. (2013). Modeling the learning progressions of computational thinking of primary grade students. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ICER '13. ACM.

[199] Sentance, S., Waite, J., Hodges, S., MacLeod, E., and Yeomans, L. (2017). "creating cool stuff": Pupils' experience of the bbc micro:bit. *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education - SIGCSE '17.*

[200] Sentence, B. (2008). A new approach to spreadsheet analytics management in financial markets. *arXiv preprint arXiv:0802.2932.*

[201] Shamoo, A. E. and Resnik, D. B. (2009). *Responsible conduct of research*. Oxford University Press.

[202] Sherrell, L. B., Robertson, J. J., and Sellers, T. W. (2005). Using software simulations as an aide in teaching combinatorics to high school students. *Journal of Computing Sciences in Colleges*, Volume 20(Issue 6):Pages 108–117.

[203] Shull, F. J., Carver, J. C., Vegas, S., and Juristo, N. (2008). The role of replications in empirical software engineering. *Empirical Software Engineering*, 13(2):211–218.

[204] Simon (2011). Assignment and sequence: Why some students can't recognise a simple swap. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, Koli Calling '11. ACM.

[205] Simon and Snowdon, S. (2011). Explaining program code: Giving students the answer helps - but only just. In *Proceedings of the Seventh International Workshop on Computing Education Research*, ICER '11. ACM.

[206] Sleeman, D., Putnam, R., Baxter, J., and Kuspa, L. (1986). Pascal and high school students: A study of errors. *Journal of Educational Computing Research*, 2(1):5–23.

[207] Snell, L. and Cunningham, M. J. (2000). An investigation into programming by voice and development of a toolkit for writing voice-controlled applications. *M Eng. Report. Imperial college of Science, Technology and Medicine, London.*

[208] Sorva, J. (2008). The same but different. students' understandings of primitive and object variables. In *Proceedings of the 8th International Conference on Computing Education Research*, Koli '08. ACM.

[209] Sorva, J. (2012). Visual program simulation in introductory programming education. PhD Thesis, Aalto University.

[210] Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education*, 13(2):1–31.

[211] Speier, C. and Brown, C. V. (1997). Differences in end-user computing support and control across user departments. *Information & Management*, 32(2):85–99.

[212] Swidan, A., Serebrenik, A., and Hermans, F. (2017). How do Scratch programmers name variables and procedures? In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE.

[213] Takang, A. A., Grubb, P. A., and Macredie, R. D. (1996). The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4(3):143–167.

[214] Takeuchi, O., Ikeda, M., and Mizumoto, A. (2012). Reading aloud activity in l2 and cerebral activation. *RELC Journal*, 43(2):151–167.

[215] Teague, D., Corney, M., Ahadi, A., and Lister, R. (2013). A qualitative think aloud study of the early neo-piagetian stages of reasoning in novice programmers. In *Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136*, ACE '13. Australian Computer Society, Inc.

[216] Teague, D. and Lister, R. (2014). Longitudinal think aloud study of a novice programmer. In *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148*, ACE '14, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

[217] Teasley, B. E. (1994). The effects of naming style and expertise on program comprehension. *International Journal of Human-Computer Studies*, 40(5):757–770.

[218] Techapalokul, P. and Tilevich, E. (2017). Quality Hound? an online code smell analyzer for Scratch programs. *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*.

[219] Technet.microsoft.com. How hpc services for excel work. Online. URL: https://technet.microsoft.com/en-us/library/ff877825

[220] Thompson, E., Luxton-Reilly, A., Whalley, J. L., Hu, M., and Robbins, P. (2008). Bloom's taxonomy for cs assessment. In *Proceedings of the Tenth Conference on Australasian Computing Education - Volume 78*, ACE '08, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

[221] Tilley, T. (2003). *Formal concept analysis applications to requirements engineering and design*. PhD thesis, The University of Queensland.

[222] Tramontana, P., Risi, M., and Scanniello, G. (2014). Studying abbreviated vs. full-word identifier names when dealing with faults: an external replication. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14. ACM.

[223] U.S., N. R. C. (2011). *Report of a Workshop of Pedagogical Aspects of Computational Thinking (Computational Thinking)*. National Academies Press, 1 edition.

[224] van Wesel, P., Lin, B., Robles, G., and Serebrenik, A. (2017). Reviewing career paths of the openstack developers. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE.

[225] van Zyl, S., Mentz, E., and Havenga, M. (2016). Lessons learned from teaching Scratch as an introduction to object-oriented programming in Delphi. *African Journal of Research in Mathematics, Science and Technology Education*, 20(2):131–141.

[226] Vasilescu, B., Capiluppi, A., and Serebrenik, A. (2014a). Gender, representation and online participation: A quantitative study. *Interacting with Computers*, 26(5):488–511.

[227] Vasilescu, B., Serebrenik, A., Goeminne, M., and Mens, T. (2014b). On the variation and specialisation of workload—A case study of the Gnome ecosystem community. *Empirical Software Engineering*, 19(4):955–1008.

[228] Vbdepend.com (2015). Vbdepend achieve higher vb6/vba code quality. Online. URL: http://www.vbdepend.com/Metrics#MetricsOnMethods.

[229] Vitek, D. *Requirements Traceability Matrix Template*. Online File. URL: http://www2a.cdc.gov/cdcup/library/templates/CDC_UP_Requirements_Traceability_Matrix_Template.xls.

[230] Webb, H. C. (2011). Injecting computational thinking into career explorations for middle school girls. *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*.

[231] Whalley, J. and Kasto, N. (2014). A qualitative think-aloud study of novice programmers' code writing strategies. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, ITiCSE '14. ACM.

[232] Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P. K. A., and Prasad, C. (2006). An australasian study of reading and comprehension skills in novice programmers, using the bloom and solo taxonomies. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, ACE '06, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

[233] Wilson, A., Hainey, T., and Connolly, T. (2012). Evaluation of computer games developed by primary school children to gauge understanding of programming concepts. In *6th European Conference on Games-Based Learning (ECGBL)*, pages 4–5.

[234] Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, 49(3):33.

[235] Winston, W. L. (2001). Executive education opportunities millions of analysts need training in spreadsheet modeling, optimization, Monte Carlo simulation and data analysis. *OR MS TODAY*, 28(4):36–39.

[236] Wittwer, J. (2004). Monte Carlo simulation basics. Online, URL: https://www.vertex42.com/ExcelArticles/mc/MonteCarloSimulation.html.

[237] Woelfle, M., Olliaro, P., and Todd, M. H. (2011). Open science is a research accelerator. *Nature Chemistry*, 3(10):745–748.

[238] Wolz, U., Stone, M., Pulimood, S. M., and Pearson, K. (2010). Computational thinking via interactive journalism in middle school. *Proceedings of the 41st ACM technical symposium on Computer science education - SIGCSE '10*.

[239] Yoon, C. Y. (2009). Measures of perceived end-user computing competency in an organizational computing environment. *Knowledge-Based Systems*, 22(6):471–476.

[240] Yu, Y., Wang, H., Yin, G., and Wang, T. (2016). Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Inf. & Softw. Technology*, 74:204–218.

[241] Zimmerman, D. W. and Zumbo, B. D. (1992). Parametric alternatives to the Student t test under violation of normality and homogeneity of variance. *Perceptual and Motor Skills*, 74(3(1)):835–844.

CURRICULUM VITÆ

Alaaeddin Ayyoub SWIDAN

1985/02/18	Date of birth in Nablus, Palestine

EDUCATION

03/2015 - 02/2019	Ph.D. Candidate, Software Engineering Research Group Delft University of Technology, The Netherlands Thesis: *Challenges of End-user Programmers: Reflections from Two Groups of End-users* Supervisor: Dr. Felienne Hermans Promotor: Prof. Dr. Arie van Deursen
10/2013 - 09/2014	M.Sc. ICT Management Cardiff Metropolitan University, United Kingdom Thesis on *Spreadsheet Clone Detection*
09/2003 - 05/2008	Bachelor of Computer Engineering An Najah National University, Palestine

EXPERIENCE

02/2019 - 08/2019	Postdoctoral Researcher, Programming Education Lab (PERL) Leiden University, The Netherlands
11/2014 - 03/2015	Assistant IT Project Manager Office of the European Union Representative, Palestine
07/2008 - 09/2014	Integration Engineer Palestine Telecommunications Company, Palestine

ACADEMIC SERVICE

PC Member	Blocks and Beyond Workshop, 2019
Reviewer	TOCE, 2019 SIGCSE, 2018
Teaching Assistant	Introduction to Programming, Dr. Felienne Hermans, 2017

List of Publications

🗎 9. *Alaaeddin Swidan*, Felienne Hermans. The Effect of Reading Code Aloud on Comprehension: An Empirical Study with School Students. The ACM Global Computing Education Conference (CompEd). Chengdu, China, 2019

8. Felienne Hermans, *Alaaeddin Swidan*, Efthimia Aivaloglou. Code Phonology: An exploration into the vocalization of code. IEEE/ACM International Conference on Program Comprehension (ICPC). Gothenburg, Sweden, 2018

7. Felienne Hermans, *Alaaeddin Swidan*, Efthimia Aivaloglou, Marileen Smit. Thinking out of the box: comparing metaphors for variables in programming education. The Workshop in Primary and Secondary Computing Education (WiPSCE). Potsdam, Germany, 2018

🗎 6. *Alaaeddin Swidan*, Felienne Hermans, Marileen Smit. Programming Misconceptions for School Students. ACM International Computing Education Research (ICER). Espoo, Finland, 2018

🗎 5. *Alaaeddin Swidan*, Felienne Hermans. Programming Education to Preschoolers: Reflections and Observations from a Field Study. The 28th Annual Workshop of the Psychology of Programming Interest Group (PPIG). Delft, The Netherlands, 2017

🗎 4. *Alaaeddin Swidan*, Alexander Serebrenik, Felienne Hermans. How do Scratch Programmers Name Variables and Procedures?. IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM). Shanghai, China, 2017

🗎 3. *Alaaeddin Swidan*, Felienne Hermans. Semi-automatic extraction of cross-table data from a set of spreadsheets. The International Symposium on End User Development (IS-EUD). Eindhoven, The Netherlands, 2017

🗎 2. *Alaaeddin Swidan*, Felienne Hermans, Ruben Koesoemowidjojo. Improving the Performance of a Large Scale Spreadsheet: A Case Study. IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER). Osaka, Japan, 2016

1. Felienne Hermans, Bas Jansen, Sohon Roy, Efthimia Aivaloglou, *Alaaeddin Swidan*, David Hoepelman. Spreadsheets are Code: An Overview of Software Engineering Approaches applied to Spreadsheets. IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER). Osaka, Japan, 2016

🗎 Included in this thesis.

Acknowledgments

Many people helped and supported me to get here after four-years-and-a-half of starting the Ph.D. journey. I want to give them here the least I could provide: my thanks and heartful gratitude.

To my co-promotor and daily supervisor, Felienne: I find it impossible to get the right words to thank you! I still remember how welcoming and encouraging you were during the first Skype meeting we had while I was just a random master student. From that moment onwards I just got your complete support as a mentor and later as an official supervisor. The road has not always been smooth, but you always managed to inspire me to go further. I learned a lot from you, not only on the academic level. Heel erg bedankt!

To Arie, thank you for allowing me the opportunity to pursue a Ph.D. in your group. Thanks for the guidance you gave whenever it was needed.

For my collaborators, I thank you for the amount of work and support you gave me during the joint work. Thanks to Alexander, Fenia, Ruben, and Marileen. The acknowledgments cannot be complete without writing a few words for the most remarkable colleagues and friends. Fenia, you are amazing! Besides your great knowledge and experience that you happily give whenever needed, you have this unique personality that always cheers up and positively encourages people around you. We still remember the international dinner you hosted, which was the warmest start to our journey as a family in the Netherlands. Thanks for the countless times you gave me tips for places to go or things to do. Sohon, thanks for being nice to me all the time. We shared a lot during the past five years besides the office space, of course. I already miss our chats about food, our expat life, and politics. Bas, thanks for being my paranymph! You were at the office once a week, but you always had this incredible impact on discussions with your honest and clear thoughts, besides the excellent and detailed tips that you always provided when I continuously asked you for help. I learned a couple of things from you! Mozhan, you have this distinct caring personality that is always looking to help your friends, myself included, thanks for that! Shirley: Thanks for the many times you helped me in schools, your passion and dedication to programming education is an example to follow, and thanks a lot for assisting in the cover design. Anna: Thank you for the support, especially during the final thesis preparations. Thanks also for the rest of the SERG and PERL groups, I was lucky to be a part of such distinguished working environments and colleagues.

Our industry and community partners: I want to thank Bitbrains (later Solvinity) for financially supporting my research work in this dissertation. I want to give special thanks to Hein Brat in particular for showing the support and enthusiasm to the spreadsheet project from the very first moments.

I want to thank the team at Nemo Science Museum who opened the door for us to study school-age children who visited the museum for two weeks. My thanks also go to all schools, which allowed me the opportunity to give programming lessons and perform studies with their students. Finally, I want to thank all the student volunteers who helped me during my experiments. Special thanks to Munire for the help in classrooms and the data entry of students' assignments.

To my wife *Oraib*: it is that simple that nothing of this could have been done without you. Being there next to me in this journey gave me the inner-strength during the difficult times and pushed me to go forward and to overcome challenges. You made sure that you provide all the love and care to our two beautiful children whenever I was away for work. I love you, and I cannot thank you enough, this work is dedicated to you. To my *Yara* and *Ahmad*: I love you both beyond words. Your love and laughs always remind me of what is essential in life.

To my father and mother, thank you fo all the unconditional support and love. To my brothers and sisters thanks for being there for me whenever I needed you, I love you all. To my parents in law, I thank you for all the support and care you always give while being thousands of miles away. I am also grateful to my brothers- and sisters-in-law for their enormous support during the past five years. I am lucky to have such a big loving family. Thank you all!

Alaaeddin
Delft, September 2019