Automated Identification and Application of Code Refactoring in Scratch to Promote the Culture Quality from the Ground up

Peeratham Techapalokul

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Eli Tilevich, Chair
Stephen Edwards
Felienne Hermans
Dennis Kafura
Francisco Servant

May 7, 2020
Blacksburg, Virginia

Keywords: code quality, code smells, refactoring, Scratch, block-based programming, introductory computing education, novice programmers

Automated Identification and Application of Code Refactoring in Scratch to Promote the Culture Quality from the Ground up

Peeratham Techapalokul

(ABSTRACT)

Much of software engineering research and practice is concerned with improving software quality. While enormous prior efforts have focused on improving the quality of programs, this dissertation instead provides the means to educate the next generation of programmers who care deeply about software quality. If they embrace the culture of quality, these programmers would be positioned to drastically improve the quality of the software ecosystem. This dissertation describes novel methodologies, techniques, and tools for introducing novice programmers to software quality and its systematic improvement. This research builds on the success of Scratch, a popular novice-oriented block-based programming language, to support the learning of code quality and its improvement. This dissertation improves the understanding of quality problems of novice programmers, creates analysis and quality improvement technologies, and develops instructional strategies for teaching quality improvement. The contributions of this dissertation are as follows. (1) We identify twelve code smells endemic to Scratch, show their prevalence in a large representative codebase, and demonstrate how they hinder project reuse and communal learning. (2) We introduce four new refactorings for Scratch, develop an infrastructure to support them in the Scratch programming environment, and evaluate their effectiveness for the target audience. (3) We study the impact of introducing code quality concepts alongside the fundamentals of programming with and without automated refactoring support. Our findings confirm that it is not only feasible but also advantageous to promote the culture of quality from the ground up. The contributions of this dissertation can benefit both novice programmers and introductory computing educators.

Automated Identification and Application of Code Refactoring in Scratch to Promote the Culture Quality from the Ground up

Peeratham Techapalokul

(GENERAL AUDIENCE ABSTRACT)

Software remains one of the most defect-prone artifacts across all engineering disciplines. Much of software engineering research and practice is concerned with improving software quality. While enormous prior efforts have focused on improving the quality of programs, this dissertation instead provides the means to educate the next generation of programmers who care deeply about software quality. If they embrace the culture of quality, these programmers would be positioned to drastically improve the quality of the software ecosystem, akin to professionals in traditional engineering disciplines. This dissertation describes novel methodologies, techniques, and tools for introducing novice programmers to software quality and its systematic improvement. This research builds on the success of Scratch, a popular visual programming language for teaching introductory students, to support the learning of code quality and its improvement. This dissertation improves the understanding of quality problems of novice programmers, creates analysis and quality improvement technologies, and develops instructional strategies for teaching quality improvement. This dissertation contributes (1) a large-scale study of recurring quality problems in Scratch projects and how these problems hinder communal learning, (2) four new refactorings, quality improving behavior-preserving program transformations, as well as their implementation and evaluation, (3) a study of the impact of introducing code quality concepts alongside the fundamentals of programming with and without automated refactoring support. Our findings confirm that it is not only feasible but also advantageous to promote the culture of quality from the ground up. The contributions of this dissertation can benefit both novice programmers and introductory computing educators.

Acknowledgments

I would like to express my gratitude to all those who contributed to the successful completion of this dissertation[1].

First, I would like to express my gratitude to my advisor, Dr. Eli Tilevich, for his continued guidance and support. When almost six years ago, he suggested refactoring for block-based languages as a possible dissertation topic, I was seriously doubtful. I am so glad that he managed to convince me to follow this topic after all. This unlikely combination turned out to be extremely motivating intellectually, so my dissertation research ended up something that I have enjoyed working on. I often remind myself how much I take for granted to have his feedback and guidance whenever needed. His knowledge and experience have been invaluable from helping me to communicate effectively to preventing me from doing unnecessary work. I appreciate that he lets me learn how to write through my apprenticeship with him, which inspires me to write well and makes me enjoy writing more. He guided and supported me through the many great opportunities to gain experience and improve myself. I am very thankful for his friendship and good memories including playing ping pong and laughing at his jokes. He has made my Ph.D. experience less stressful and more enjoyable.

I would like to thank the many people who have helped make this dissertation possible in small and big ways. I would like to thank all my committee members, Drs. Steven Edwards, Felienne Hermans, Dennis Kafura, and Francisco Servant, for their valuable insights and guidance that helped improve the technical quality of this dissertation. I would like to thank Dr. Simin Hall for her helpful and reassuring guidance and feedback, and Dr. Feli-

[1]This material is based upon work supported by the National Science Foundation (NSF) under Grant No. DUE-1712131, "Systematic Quality Analysis and Improvement for Block-Based Software: Promoting the Culture of Quality From the Ground Up." Any opinions, findings, and conclusions or recommendations expressed herein are those of the author(s) and do not necessarily reflect the views of NSF.

enne Hermans for her kindness and willingness to test my early research tool in her class. Thanks to Dr. Franklyn Turbak, my colleagues and anonymous reviewers who helped provide constructive feedback on the drafts of several conference papers that this dissertation is based upon. I am fortunate to work with several talented colleagues who helped develop the research tools used in this dissertation, especially Prapti Khawas who I have the opportunity to mentor. She has contributed greatly to the research infrastructure.

I am thankful to have wonderful lab mates who are very supportive and we keep one another motivated. Thanks to my Thai friends at Virginia Tech and the University of Virginia for keeping me company outside of study.

I am grateful for the support from the Thai Government Scholarship Program for the educational opportunity to study abroad to the highest degree after high school graduation. I also would like to recognize my high school teachers in Thailand who supported and provided opportunities for me to discover my potential.

I am very grateful for my family for their constant love and support. I am indebted to my parents, for their tireless dedication to my sister and me. I thank my sister for always making time to talk, listen to and support each other. I would not be able to make it this far without them.

Finally, I would like to thank my loving and caring Arisa for always being there to listen, support and encourage me to do my best. I am very grateful to have her in my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software makes up the very fabric of modern society, which is increasingly driven by software-based products and services. Despite its wide utilization, software has one of the highest defect rates across all engineering artifacts because of its poor quality [41, Chapter 1]. As quality problems plague the modern society's software infrastructure, computing educators are increasingly recognizing the importance of developing students' knowledge, attitudes and practices centered around software quality.

Block-based programming languages, such as Scratch [75] and AppInventor [107], have shown to be a highly effective tool for introducing introductory learners to computing [6]. As it turns out, similarly to any non-trivial software projects, block-based programs are rife with recurring code quality problems [1, 37, 63]. Poor code quality is shown to negatively impact novice programmers' ability to understand and modify programs [34]. It has been observed that novice programmers in this domain start forming programming habits early [59, 78], so introductory computing education has a unique role to play in promoting *the culture of quality* from the ground up.

The long-term objective of this dissertation research is to drastically improve the quality of software infrastructure. Unlike prior approaches that focus on *the programs*, we focus on *the programmers*, at the time when we start introducing them to the computing discipline. The thesis of this dissertation is that *it is feasible and advantageous to identify code smells in and provide automated refactoring for Scratch in order to support the introduction of software*

quality concepts alongside the fundamentals of programming.

This dissertation research provides the means for introducing novice programmers to code quality and its improvement practices. We build on the success of Scratch, one of the most popular block-based programming languages, to reach a broad target audience of novice programmers. To achieve our stated objectives, we have pursued three key interrelated research thrusts: (1) understand recurring code quality problems, endemic to novice Scratch programmers; (2) create code quality improvement techniques and tools; (3) study the impacts of introducing code quality concepts alongside the fundamentals of programming.

To study code quality problems and their systematic improvement in introductory computing context, this research leverages two well-known software engineering concepts: (1) *code smells*, structural patterns in the source code indicative of possible quality problems (2) *refactoring*, a code improvement technique associated with removing code smells by transforming a program while preserving its behavior [26].

The remainder of this chapter provides an overview of this dissertation's organization.

1.1 Understanding Code Quality Problems

By improving the understanding of code quality problems and their negative impact, we strive to raise the research community's awareness about code quality issues in this domain and inform the efforts to address them. In Chapter 3: Code Quality Problems in Scratch and Their Impacts, we discuss our work on identifying the presence of code smells, whose prevalence and severity we assessed in a large and representative dataset of over one million Scratch projects.

RQ1: What known code smells described in the literature are applicable in the context

of Scratch?

RQ2: What can the analysis of popular Scratch projects teach us about the state of software quality in this programming domain?

To shed light on how code quality problems may impact communal learning, we analyzed 512 popular Scratch projects and studied the relationship between quality problems and code modifications in their remixes.

RQ3: How prevalent and severe are different code smells in the general Scratch code base?

Furthermore, we sought to deepen our understanding of novice programmers' proneness to introducing recurring quality problems. In Chapter 4: Code Quality Problem Trends Among Novice Programmers, we analyze a longitudinal data analysis of close to 4,000 projects created by 116 Scratch programmers. We examined how different factors such as novice programmers' exposure to core programming constructs impact how prone novice programmers are to introducing code smells in their projects.

RQ4: Does the code quality improve, as novice programmers gain experience?

RQ5: How persistent are poor coding practices, as novice programmers gain experience?

RQ6: Which Scratch constructs and idioms reduce the prevalence of code smells in projects written by novice programmers?

1.2 Code Quality Improvement Techniques and Tools

To support refactoring practices, automated refactoring tools have become a standard part of modern integrated development environments (IDEs). However, the automated refactoring support for Scratch and other novice programming environments remains woefully lacking,

providing novice programmers with few rudimentary refactorings, such as renaming variables. While we can teach novice programmers to carry out the refactoring transformations by hand, novice programmers could be discouraged from improving code quality because of the tedious and error-prone nature of manual refactoring.

In Chapter 5: Systematic Code Quality Improvement for Scratch, we present a catalog of Scratch refactorings that addresses some of the highly recurring quality problems. For each refactoring, we formalized its definition in the form of refactoring constraints and transformations considering the language features of Scratch. We present our approach to enhance Scratch programming environment with a refactoring infrastructure that facilitate program analysis and transformation implementations of the introduced refactorings.

To ascertain the effectiveness of refactorings, we evaluated their applicability and impacts on the refactored projects' code quality.

RQ7: How applicable is each introduced refactoring?

RQ8: How do the refactorings impact code quality?

We add automated refactoring support to Scratch programming environment and conduct a between-subjects study with 24 participants to evaluate how automated refactoring support impacts novice programmers' willingness to improve code quality.

RQ9: Does the presence of automated refactoring support affect how willing novice programmers are to refactor their code?

1.3 Introducing Code Quality Concepts Alongside the Fundamentals of Programming

As computing educators have been recognizing the importance of software quality concepts in introductory curricula, it remains unclear how suitable and useful it is to introduce this topic to novice programmers.

In Chapter 6: Introducing Code Quality Concepts Alongside the Fundamentals of Programming, we present a controlled study that involves 24 adult beginner programmers. The study introduces them simultaneously to procedural abstraction and a refactoring that removes code duplication. The study seeks to understand the impact of introducing code quality concepts alongside the fundamentals of programming and the role of automated refactoring.

RQ10: Do novice programmers perceive enjoyment and difficulty differently when learning how to use procedural abstraction as compared to learning how to refactor code duplication?

RQ11: Does introducing novice programmers to the Extract Custom Block refactoring help them understand the concept of custom blocks?

RQ12: Does the preference for manual or automated refactoring methods change and why, as novice programmers progress from learning refactoring to intending to refactor in the future?

RQ13: Does introducing code duplication and its refactoring to novice programmers help them understand the concepts of software understandability and modifiability?

This dissertation research is based on the following publications:

- Chapter 3: Peeratham Techapalokul and Eli Tilevich. **"Understanding recurring quality problems and their impact on code sharing in block-based software."** (*Visual Language and Human-Centric Computing (VL/HCC) 2017*) [96]
- Chapter 4: Peeratham Techapalokul and Eli Tilevich. **"Novice programmers and software quality: Trends and implications"** (*Software Engineering Education and Training 2017*) [95]
- Chapter 5: Peeratham Techapalokul and Eli Tilevich. **"Code quality improvement for all: Automated refactoring for Scratch"** (*VL/HCC* 2019)[97]
- Chapter 6: Peeratham Techapalokul, Simin Hall, and Eli Tilevich. **"Teaching the Culture of Quality from the Ground Up: Novice-Tailored Quality Improvement for Scratch Programmers"** (*The American Society for Engineering Education Annual Conference & Exposition 2020*) [72]

Chapter 2

Literature Review

This chapter reviews the most closely related work on code smells, refactoring, and educational approaches to improving code quality for novice programmers. To provide the context of this dissertation research, this chapter begins with the background of software quality and Scratch.

2.1 Software Quality

A complex and multifaceted concept, software quality commonly describes different properties and perspectives [46]. This dissertation uses the terms **software quality** and **code quality** interchangeably to refer to the *internal* quality characteristics of source code, as defined by Steve McConnell's Code Complete, one of the most well-known guides to writing high-quality software. In his classification, the *external* quality characteristics are those that are important to end users, such as correctness, reliability, and usability. In contrast, the *internal* properties are those that programmers care about, such as maintainability, reusability and understandability [58, pg. 464]. Poor code quality has been shown to negatively impact correctness and reliability [13, 32, 48, 49, 83].

Software Quality Improvement: The research literature describes various methods for improving software quality, including testing, verification, and code reviews. A widely used quality improvement method is *software inspection*—examining source code for design and

implementation defects, introduced in 1976 by Fagan [22]. Code inspection has been applied at different levels of granularity: *coding style* primarily refers to formatting (e.g., indentation, alignment, control structure) and can help to improve clarity and readability of the source code [42]. *Coding standards* codify industry-recognized best practices for programming styles and guidelines to facilitate team collaboration (e.g., C coding standards [92]). *Antipatterns*, the counterpart of *design patterns* [108], are high-level structural patterns that are used in the wrong context, thereby causing undesirable maintainability issues [12]. In this dissertation research, we apply a code inspection technique that examines source code for **code smells**, structural patterns indicative of possible implementation-level problems, such as code duplication, long method, and dead code. Section 2.4 discusses code smells in Scratch in greater detail.

2.2 Block-based programming

Block-based programming languages is a highly effective tool for introducing beginners to programming, as it eliminates the need to master syntax, the main hindrance faced by beginner programmers [6]. This programming paradigm allows programmers to create computer programs by snapping programming blocks together. Block shapes provide hints about compatibility among different block types to enforce semantic correctness. By learning to program with blocks, novice programmers quickly start mastering fundamental programming concepts right away as compared to those that learn with traditional text-based languages.

The intuitive and accessible interface of block-based programming languages make them an increasingly attractive tool for end-user programming. Examples in the research literature include programmable robots [104], smart environments [84], and clinical alert rules [50]. Experts in domains other than computing, end-user programmers write code as part of their

professional pursuits to solve important problems. Unlike student programs, end-user code executes real-world computational tasks. If written sloppily, end-user code becomes hard to understand and modify, with potentially serious harmful consequences. As a specific example, in a 2012 incident, J.P. Morgan Chase lost around $7 billion (USD) due in part to a bug in a spreadsheet program [25].

2.3 Scratch

Scratch [75] is not only known for its novice-friendly programming tool but also its growing online learning community of over 52 million members who collectively create and share over 50 million projects[1]. Scratch programmers engage in creative expression—creating non-trivial programs such as games, animation, and storytelling—as a way to learn programming. Furthermore, remixing a project—building off one's project and extending it—has been shown to help learners gain exposure to new concepts, an important part of Scratch's communal learning [18].

To make computer programming intuitive and accessible, Scratch adopts a block-based programming interface Scratch has been increasingly adopted in K-12 introductory programming curricula, workshops, and outreach programs, as reflected by the growing research studies in the literature [55, 56, 76, 77, 82, 101, 109].

2.3.1 Basics of Scratch

Figure 2.1 shows a screenshot of a Scratch project opened for editing in the Scratch programming environment. In the "Code" mode, located on the left panel, is the toolbox providing

[1] https://scratch.mit.edu/statistics/ (accessed February 25, 2020)

over 100 different blocks, grouped into different categories (e.g., Motion, Looks, Control, Operators, etc.). These blocks can be dragged to the scripting area located in the center. The top right panel shows the visual output of running the program and the bottom right panel lists all programming objects in the project. Clicking the Green flag button starts executing the program.

A Scratch project contains a single Stage object that also serves as the project's global container and scope. The Stage can contain zero or more graphic objects called *Sprites*. Both the Stage and Sprites are programmable objects, collectively referred to as *Scriptables*. A Scriptable contains a set of scripts and procedures used to program media elements (i.e., graphics called costumes, and sounds). Each script is a unit of functionality comprising a sequence of instruction blocks. A script is triggered by an event-type block, placed at the top of the script and serving as the script's entry point. Also referred to as "hat blocks", the event-type blocks can be either predefined (e.g., "When a mouse click") or user-defined (i.e., "when I receive <user-defined-message>"). The user-defined events allow scripts to communicate via a message passing mechanism across multiple sprites through *broadcast* and *receive* blocks.

2.3.2 Language Features

To lower the barrier to entry, Scratch has ruled out several common language features often introduced in introductory courses with text-based programming languages, including data types, return statements, inheritance, and polymorphism. Nevertheless, Scratch enables novice programmers to focus their learning on the fundamental programming concepts/constructs (e.g., variables, Boolean expressions, conditionals, iterations, user-defined procedure) as well as some high-level concepts, such as event-based programming, concurrency, and syn-

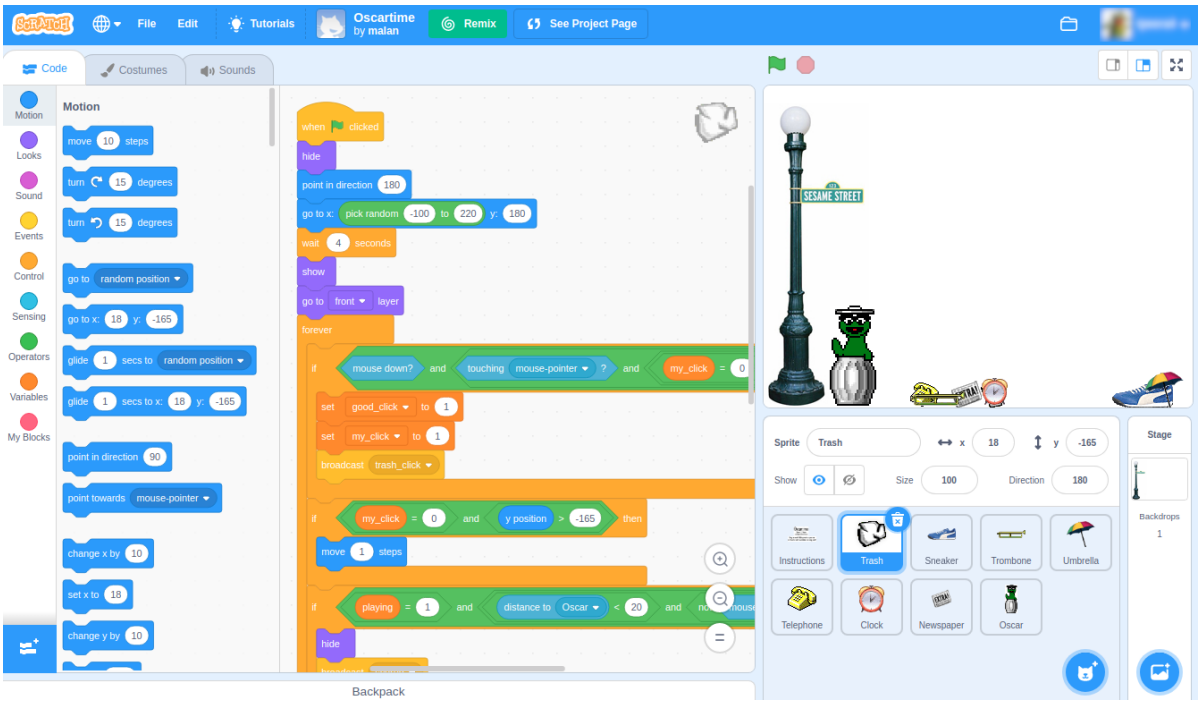


Figure 2.1: Scratch's Programming Environment

chronization.

Scratch programming constructs are simplified to make them easy to learn. It features a simple model of variables, in which their scope can be either global ("available for all sprites") or private ("available to this sprite only"). The name and scope of a variable must be specified upon its creation. Scratch features "custom block", a procedure construct that can accept parameters but cannot return a value. When a custom block is created (a definition script with "define <custom-block-name>" hat block), it belongs to the scriptable object currently being edited. The created custom block can then be called only from the code contained in the scriptable, to which the custom block belongs. Scratch provides an object cloning feature, a prototype-like mechanism that instantiates multiple objects based off a parent sprite. A clone object maintains its own copy of the parent sprite's local variables, while the parent's global variables remain accessible by the parent and all of its clones.

2.4 Code Smells

Popularized by Martin Fowler [26], code smells is a metaphor for describing structural patterns in the source code indicative of possible quality problems. Given distinctive names (e.g., Code Duplication, Long Method, Dead Code, etc.), code smells serve as a shared vocabulary for software developers to communicate about code quality problems. Expressing code quality problems as smells is practically beneficial as a way to indicate refactoring opportunities. Although code smells are most commonly used to identify quality problems at the code-level implementation, the concept has been applied more broadly to discuss quality problems in various aspects of software systems and domains (e.g., software services, usability, testing, Web, database, etc.) [85]. Code smells have been adopted widely as a practical software inspection technique, as they a amenable to automated detection. Since code smells were first formalized to enable their automated detection in Java programs [102], many works have focused on automatically detecting code smells with metrics-based and rule/heuristic-based approaches, the most common detection techniques [85].

2.4.1 Code Smells in Scratch

Quality Studies: Code smells have been applied in code quality studies for novice and end-user programming languages, such as Yahoo! Pipes web mashups [90] and spreadsheet formulas [36]. For Scratch, a few studies analyze code smells in programs written in Scratch and other block-based languages in general. Moreno and Robles analyze two code smells[2] in Scratch with an automated program analysis tool: default sprite naming and duplicate script. They found both code smells to be highly prevalent in the dataset of 100 Scratch projects [63]. Hermans et al. systematically study code smells in two educational block-based languages:

[2]referred to in their work as "bad programming practices"

Kodu and Lego Mindstorms EV3. By manually examining a small dataset of novice authored programs, they identify 11 common code smells (e.g., Dead Code, Duplicate Code, Feature Envy, Inappropriate Intimacy, Lazy Class, Long Method). Many of the identified smells correspond to object-oriented smells [37]. Aivaloglou and Hermans analyze a large dataset of over 250,000 to explore how young Scratch programmers code. Their results indicate a high occurrence of three code smells: large scripts, dead code, and duplicate code [1].

Causes and Impacts: Several studies seek to better understand the causes and impacts of code smells in Scratch. Robles et al. study the practice of software cloning among Scratch students. They found no particular computational concepts is associated with the absence of duplicated codes, while many students continue copying and pasting code, despite having been taught how to avoid it [78]. This undisciplined practice corroborates prior study findings that found introductory students programming in Java hardly resolve quality problems despite the presence of code quality tools [43]. Hermans and Aivaloglou conducted a controlled experiment to study novice Scratch programmers' performance to comprehend and modify smell afflicted programs. The results show that code smells negatively affect novice programmers trying to understand and extend existing code [34]. These results reinforce the prior findings that end-user programmers not only are aware of code smells but also prefer smell-free code [14, 36, 91].

Although these prior studies provide important empirical evidence about quality problems in Scratch, the state of code quality in its large codebase remains poorly understood. Because the types and frequency of code smells may depend on application domain [17] and language features [20], only a large scale study can comprehensively identify commonly recurring code smells endemic to Scratch. To raise the community's awareness of code quality issues in this domain, further studies should continue investigating the impact of poor code quality on Scratch's educational effectiveness.

2.5 Software Refactoring

The term *refactoring* was first introduced by Opdyke and Johnson [70], as a methodology for restructuring the source code while preserving the program behavior. The concept of refactoring was popularized by Fowler and Beck, whose book documents a catalog of object-oriented refactorings, each detailing why, when, and how to refactor common code smells [26]. Refactoring is an integral part of software development practices that maintain code quality [66, 110]. Software refactoring has been applied to multiple domains [60], and is commonly associated with code smells. When it comes to novice and end-user programming, refactoring has been applied only in few domains that include spreadsheets [4]) and pipe-like mashups [90].

In practice, software developers are encouraged to refactor with automated support, as refactoring by hand can be tedious and error-prone. Although automated refactoring support assists with analyzing and transforming the source code, it is still up to software developers to decide when, where, and how to refactor by specifying refactoring configurations. Integral to modern software development process, automated refactoring has become a standard feature of mainstream integrated development environments (IDEs), such as Eclipse [28] and IntelliJ IDE [39].

2.5.1 Automated refactoring for Blocks

Automated refactoring relies on analysis and transformation of the internal representation of the source code. For block-based programming environments, the structured blocks editor maintains an internal program representation, designed for rendering and interactively manipulating blocks, so composing a block-based program can be seen as directly manipulating the program's AST. Mainstream block-based programming environments, such as Scratch

and AppInventor leverage Blockly, a popular framework for structured blocks editing [27]. The blocks editor is a reusable component that can be customized to support different languages and integrated with the rest of the programming environment components (e.g., renderer).

In these programming environments, refactoring is only minimally supported and exposed as a part of the main blocks editing features. These environments are limited to few Blockly built-in rudimentary refactoring functionalities: renaming variables and changing function signatures (i.e., add parameters and change their order). The implementation of these refactorings is limited as well, often characterized by a simple match-and-replace strategy. Overall, the Blockly framework implements minimal built-in semantic analysis capabilities, and leaves it up to language designers to implement the necessary analysis and transformation support. If a block-based language environment is to support more advanced refactorings, such as *Extract Procedure*, it would need to be enhanced with a powerful refactoring infrastructure, comprising both program analyses and transformation.

2.5.2 Techniques and Approaches

An important part of applying refactoring is ensuring that the program transformations are behavior-preserving by checking refactoring constraints. The most common approach is *preconditions*, first introduced in Opdyke's Ph.D. thesis [70]. With the precondition approach, a set of conditions is checked before the program transformations. In addition to preconditions, the research literature documents other common constraint checking techniques, such as *Invariants*, the conditions that need to be valid before and after refactoring and *Postconditions*, conditions that must be valid after the refactoring [60]. Specifying these refactoring constraints must take into consideration the target programming language' syntax and se-

mantics.

Refactoring engines operate on some internal program representations with a common one being *a program graph*. As discussed in [71], program graph is an AST augmented with semantics edges that express various relationships (e.g., reference binding, def-use chains, etc.). This representation provides the flexibility in analyses and transformations, in which additional semantics edges are introduced as required by a given analysis.

Although the underlying representation of block-based programs closely follows the abstract syntax tree (AST), it might not be appropriate for program analyses and transformations required by refactoring. To support advanced refactorings, block-based programming environments have to be enhanced with more complex program analysis and transformation facilities. Nevertheless, several facets of the refactoring infrastructure for block-based programming languages can build on the prior applications of analysis techniques (e.g., control flow and data flow) to automated refactoring despite its dominant text-based context.

Refactoring User Interfaces: When it comes to designing refactoring for novice programmers, few studies provide the insights about usability concerns and design approaches. It has been shown that even experienced programmers are overwhelmed by the refactoring interfaces of modern IDEs due to their usability problems [66]. Prior research suggests the benefits of combining code smell detection and automated refactoring to not only reuse the common analysis functionality in both components [67], but also to make refactoring more usable [94]. When providing refactoring support, these prior findings highlight several important design considerations that are likely to be amplified in the context of supporting novice programmers.

2.6 Software Quality and Introductory Computing

The research literature shows that software quality topics are increasingly integrated into computing education. Much of CS education research focuses on software testing [15, 21, 38, 52, 88], which largely limits the scope of software quality to correctness, so quality aspects as understandability and maintainability receive insufficient attention. A recent study by an ITiCSE'17 Working Group points out that CS students possess a low level of skills and knowledge about code quality, concluding that CS education programs should discuss this topic more [8]. Overall, the research literature treats code quality as a topic reserved for more advanced computing learners, with only few studies targeting introductory learners.

2.6.1 Code Quality Education for Scratch

The dissertation work of Monroy-Hernández and Resnick, the developer of Scratch remixing, introduced the issue of quality problems in Scratch. In particular, a Scratch project lacking modularity can negatively impact the likelihood of the project being remixed and built on [62]. As Scratch is being adopted for classroom use, educators start to observe code quality issues in student projects. Meerbaum-Salant et al. identify two programming practices among students programming in Scratch that are at odds with conventional practices: 1) extremely bottom-up programming and 2) excessive program decomposition [59]. Since these early works, several studies have explored code quality issues in Scratch, particularly those that analyze Scratch projects for code smells, discussed in Section 2.4.

Some prior efforts give feedback about code quality to encourage Scratch programmers to improve the quality of their code. Boe et al. developed Hairball, an analysis tool that detects improper Scratch coding practices, such as missing variable initialization, unsafe uses of broadcast/receive events that cause infinite loops [7]. Moreno-León et al. developed

Dr.Scratch, an online analysis tool that analyzes Scratch projects for code smells and bad practices, such as duplication, dead code, sprite naming, and uninitialized local variables, in addition to their computational thinking score [64].

A few recent works explore how software engineering principles and practices that promote code quality can be integrated into the introductory CS curriculum using Scratch. Hermans and Aivaloglou concluded that it is feasible and useful to teach K-12 students software engineering principles and practices, including code duplication and how to avoid it [35]. When learning software development skills in a Scratch project-based workshop, K-6 learners responded positively to the covered material, thus indicating that even introductory students can be receptive to software engineering ideas, introduced at the appropriate level [31]. Rose et al. developed Pirate Plunder, a game-based intervention that teaches learners about custom blocks to remove code duplication in a Scratch-like environment. Although different from our approach, their work reports encouraging results that show how learners can internalize code improvement skills Having played the game, learners were observed to be more likely to apply custom blocks to reuse code when programming in Scratch, as compared to non-game control groups [80].

Code quality and its improvement practices are only starting to get incorporated into introductory computing curricula. The findings of these prior works provide encouraging results indicating that it is feasible and useful to teach code quality to the novice audience of Scratch programmers. For code quality to become a standard part of introductory CS education, the topic needs to be taught alongside the fundamentals of programming. Additionally, any educational interventions intended for this target audience must be congruent with Constructivism. This educational philosophy underlies and guides the design and pedagogy of Scratch and other educational block-based programming languages. In Constructivist-based learning, learners are encouraged to actively self-construct knowledge through experiential

and unconstrained learning, a learning style that some educators believe to be inconsistent with following any systematic programming practices.

Chapter 3

Code Quality Problems in Scratch and Their Impacts

3.1 Introduction

The key functions of modern society critically depend on software-based systems. Finance, transportation, communication, government, defense—all rely on software to manage and carry out day-to-day operations. A key factor that determines the utility and safety of any software-based system is software quality. In this respect, poor software quality is known not only to increase the development and maintenance costs, but also be conducive to causing software defects [32, 41].

Block-based languages have become an important entryway to the world of software development for CS learners and end-user programmers alike. Although one may argue that block-based programs are too simple to warrant any quality concerns, the issue at hand is the formation of good habits that promote solid software engineering practices, as block-based programmers move forward in their computing journeys. In any case, software quality is known to be inversely correlated with the effort required to understand, modify, and evolve a software system [69, 111]. In that light, improving software quality is an important process, with the assessment of quality problems being the critical first step in this process.

See Appendix A.1 for *QualityHound*, our end-user analysis tool based on this work

In addition to the societal impact, poor software quality can hinder learning enabled by code sharing, an important learning activity for novice programmers. For example, Scratch, the language we focus on here, has a large and engaging online community, whose slogan is "Imagine, Program, Share." This slogan reflects a vision of sharing—called *remixing* in Scratch—being a central tenet of this learning community [62]. Yet, the majority of Scratch projects we studied have had a limited success in allowing others to extend them, rendering these projects less "remixable," which we define as how easy a project is to be extended and modified by others. As we have discovered, software quality can be an important factor affecting whether a project is remixable.

In this work, we document recurring quality problems in block-based programs written in Scratch by leveraging the well-recognized software quality assessment methodology of *code smells* [26]. In essence, a code smell documents a recurring pattern of design and/or implementation choices that indicate the symptoms of software quality problems. We study how the presence of code smells affects remixed projects in terms of their "remixability." In fact, some code smells that we studied have shown statistically significant effects on how remixable a project is. The intuition behind this insight is simple: for a project to be inviting for other programmers to remix and extend, it has to be easy to understand and modify, a property hindered by the presence of some code smells.

To establish a practical benchmark for the thresholds at which the presence of code smells starts hindering remixing, our study focuses on popular remixed projects, whose remixes have been substantially extended. We then use these benchmark-based thresholds to determine the severity of the discovered code smells in our subject dataset, which comprises close to 600K projects.

The low-risk category of smell severity indicates the level of quality at which a project is likely to be remixed and extended. As our results show, some code smells with high

prevalence have a larger percentage of projects in the very high risk category compared to the benchmark projects. Conversely, successful remixed projects exemplify how high software quality can help uphold the fundamental sharing principle of the learning community of novice programmers.

The goal of our study is to answer the following research questions:

- ***RQ1:*** *What known code smells described in the literature are applicable in the context of Scratch?*

  Motivation: The incidence of code smells is commonly influenced by certain programming language features. Although block-based languages share many similarities, they tend to differ with respect to their feature-sets. As a result, recurring quality problems afflict programs in these languages in dissimilar ways. This work contributes a catalog of code smells specific to Scratch, thus far not thoroughly compiled and documented, benefiting all the stakeholders in Scratch and giving insights to the stakeholders in other block-based languages.
- ***RQ2:*** *What can the analysis of popular Scratch projects teach us about the state of software quality in this programming domain?*

  Motivation: Popular projects can strongly impact the software development practices of novice programmers, who commonly remix these projects, thus using them as an active learning resource. Understanding the software quality of popular projects can provide insights about how quality affects code remixing and also make it possible to derive practical software quality benchmarks.
- ***RQ3:*** *How prevalent and severe are different code smells in the general Scratch code base?*

  Motivation: Knowing how prevalent each type of code smell is in a large population of Scratch programs can provide helpful hints for the tool builders, whose aim is to

enable block-based developers to improve the quality of their projects. When providing refactoring tools support, one should pay special attention to the code smells with high prevalence and severity.

This chapter makes the following contributions:

1. **A catalog of code smells for Scratch** We present a catalog of code smells for Scratch, drawn from the research literature on recurring software quality problems.
2. **An assessment of how the presence and density of code smells affect the likelihood of a popular Scratch project being remixed and extended** We propose the *Script Addition* metric as a simple heuristic to identify Scratch projects that are likely to exhibit high comprehensibility and extensibility. We statistically evaluate whether the proposed metric can be relied on to predict the rate of incidence of code smells.
3. **A large-scale assessment of the prevalence and severity of code smells in Scratch projects** We present our findings on the prevalence and severity of code smells in a large dataset of close to 600K Scratch projects by using the percentile-based risk thresholds derived from the subset of popular projects whose *Script Addition* metric is ranked in the top 25%.

Chapter Organization: The remainder of the chapter is organized as follows: Section 3.2 presents our approach to cataloging smells, and the identified catalog as well as explains how we designed our quality assessment study. Section 3.3 explains the results of our study. Section 3.4 presents the discussion of our results and the threats to validity. Finally, Section 3.5 presents future work direction and concluding remarks.

3.2 Methodology

We derive our catalog of code smells for block-based software by examining the research literature on this topic, considering smells in different types of programming domains. In particular, the code smells we consider in this work are derived from the following categories:

- **Classic code smells:** These language independent code smells have been identified and documented a long time ago, as they universally occur in all types of software domains. The smells in this category include *Duplicate Code* and *Long Script*.
- **Object-oriented code smells:** Scratch supports a limited form of the object-oriented programming style with *scriptable objects*. In a way, each scriptable embodies an object with an encapsulated `private` state that can only be modified in response to receiving external events. In essence, scriptables can be seen as single-instance objects, whose interactions with each other are also subject to the kinds of smells usually found in object-oriented software. In fact, some of the smells in this category have already been identified in the literature. For example, OO-inspired block-based smells, such as *Feature Envy* and *Inappropriate Intimacy* have been identified in reference [37]. The smells in this category include *Middle Man*.
- **End-user code smells:** Some end-user code smells in the literature [90] are also applicable to Scratch. The smells in this category include *Duplicate String* and *Uncommunicative Name*.
- **Interface code smells:** Some of the smells are unique to block-based software. Although many of these smells share similarities with ones in other categories, certain aspects of Scratch design make these smells unique for this domain. For example, the IDE support for naming sprites is partly responsible for introducing one of these smells, *Uncommunicative Name* [63]. The smells in this category include *No-op*, *Broad*

Variable Scope, and *Undefined Block*.

Designating a recurring software quality problem as a smell is a subjective decision. Software domains are known to have unique quality problems [102]. Informed by this insight, we take a conservative approach to introducing new smells, rather preferring to focus on the known smells, identified earlier in each of the categories listed above. In other words, we deliberately disregard potential code smells that are domain-specific, instead focusing on the general smells that commonly occur in Scratch programs. We also disregard those code smells that are unavoidable, possibly due to the limitations of the language. For example, *Duplicate Code* across scriptable objects is not considered because scripts and custom blocks cannot be shared among scriptables. *Long Parameter* is another example deemed irrelevant, as Scratch provides no way to construct data objects as object-oriented languages do to address this problem.

3.2.1 Datasets

Scratch currently has over 17.3 million users and over 21 million projects shared[1]. We study Scratch because of its popularity and the accessibility of a large set of projects that this educational / end-user programming community has made publicly available. Our analysis relies on two datasets of Scratch projects, which serve different purposes in our study as described next.

A Large dataset of Scratch projects The first dataset consists of 1,066,308 shared Scratch projects randomly collected during April-July 2016. This dataset will be used in the assessment of the prevalence and severity of code smells in Scratch programs in Section 3.3.2.

[1]https://scratch.mit.edu/statistics/ (accessed March 2017)

Statistic	N	Mean	St. Dev	Min	Pctl(25)	Median	Pctl(75)	Max
# of blocks	1009192	140.95	571.88	0	9.44	27.93	75.7	25255
# of procedures	1009192	0.87	7.09	0	0	0	0	947
# of scripts	1009192	16.76	60.11	0	1.29	3.97	11.06	8020
# of sprites	1009192	5.9	9.78	1	1.75	2.95	5.52	595
# of variables	1009192	2.78	14.84	0	0	0	0.84	1121
script length	4461713	12.84	24.13	0	4	7	13	1279

Table 3.1: Basic summary statistics of large dataset of over one million Scratch projects.

Out of these projects, we were able to successfully parse and analyze 1,009,192 projects. The disregarded projects were the ones that our parser and analyzer could not handle.

Data exclusion: For the large dataset, the summary statistics in Table 3.1 suggests the majority of projects are very small and may not exhibit code smells that we are interested in. To yield meaningful results, we consider projects of sufficient size (20 blocks or more) when assessing the prevalence of code smells.

Based on this criteria, we parsed and filtered the initial project dataset, identifying 594,988 projects deemed as worthy to be analyzed for the presence of smells. In effect, excluding projects on this principle also disregards non-programming projects that contain little interesting programming logic referred to in the literature as "coloring-contests" [18].

Statistic	N	Mean	St. Dev.	Min	Pctl(25)	Median	Pctl(75)	Max
Views	519	80,317.9	115,952.1	15,780	32,559.5	48,887	83,379	1,916,543
Favorites	519	2,632.9	1,705.8	631	1,579	2,032	3,103.5	13,203
Loves	519	3,180.4	2,006.9	734	1,966.5	2,479	3,775.5	16,419
Remixes	519	593.4	1,347.1	26	137.5	265	505	22,589
Sprites	519	18.9	28.9	0	5	12	23	401
Scripts	519	109.4	246.6	1	23	56	116	3,869
Blocks	519	1,217.2	1,748.0	2	222	621	1,383	14,801

Table 3.2: Basic summary statistics of 519 popular Scratch projects

Top popular Scratch projects For the second dataset, we consider 620 popular projects, hosted on the Scratch website[2], as the initial dataset. Out of this dataset, we select the parse-able projects with at least 80 remixes, resulting in 519 projects used for the study. Our reasoning behind this selection procedure is that for a project to get remixed, it has to enjoy popularity.

Table 3.2 shows the basic summary statistics of this dataset. As seen, popular projects have a high number of views, marked as "favorite" and "loved" by others, with many remixes. These projects come from a wide range of sizes, based on the number of blocks used (comparable to the lines of code metric).

3.2.2 Computing code smell metrics

Next, we describe how the code smell metrics used in this study are computed.

Automated smell analysis

We develop an automated code smell analysis tool for Scratch. We leverage a widely used parser generator, Java Compiler Compiler (JavaCC), that takes a grammar as input and automatically generates a parser. As input to JavaCC, we pass the grammar describing the internal representation of Scratch programs. The resulting parser reconstructs the abstract syntax tree (AST) objects from the source code of Scratch projects, already structured using an AST-like representation in the JSON format. We leverage JastAdd [33], a Java-based system for compiler construction that supports the development of analysis tools. Because of the excellent tooling support for compiler-based projects in Java, we used this language to write all our analysis routines.

[2]https://scratch.mit.edu/explore/projects/all/popular (as of March 2017)

Calculating code smells metrics

There are three possible options for calculating code smells metrics: (1) density per 100 blocks, (2) percentage of smell instance relative to program elements of interest, and (3) instance counts. We make use of all these options, depending on the code smell under study. For example, we use density to adjust for the project size in certain code smells whose incidence tends to increase with the size of the source code. For instance, *Duplicate Code* with the density of 3 means an average of 3 Duplicate Code instances per hundred blocks in the project. We use percentage for those code smells whose incidence tends to grow proportionately to the number of the program elements that can be afflicted by the smell. For example, *Long Script* with 25% means that, on average, 1 out of 4 scripts in the project suffers from the smell. For other code smells with a rare incidence rate expected, we use smell instance counts. We occasionally refer to the three types of smell metrics collectively in the rest of the chapter as *code smell incidence rate*. Section 3.3.1 defines the studied code smells as well as the options used for calculating them.

Code smell analysis parameters

Certain code smells require specifying parameters. Changing these parameter will lead to different results, thus affecting the reproducibility of this work. Although the analysis routines for the majority of analyzed smells require no parameterization, the routines that search for the presence of Duplicate Code, Duplicate Script, Long Script and Inappropriate Intimacy can be parameterized with different sensitivity levels.

For DC, our analysis implementation is based on [40]. The analysis disregards duplicate code segments if they happen to be parts of larger duplicate segments. Our assumption is that the size of a duplicate segment is directly proportional to how harmful the impact of

Metric (Associated smell)	Description	N	MED	Pctl(70)
scriptLength (LS)	Vertical length (number of blocks) of a script	4,461,713	7	11
foreignAttrAccessNum (FE)	# of accesses made by a considered sprite to read external sprites' local variables	66,725	2	4
duplicatedStringGroupSize (DS)	Number of times a certain duplicate string repeats	126,475	3	4
stringLength (DS)	Number of characters contained in a duplicate string	190,881	10	18

Table 3.3: Thresholds for certain smell detection

this smell is. Hence, smaller duplicate segments within the larger duplicate segments can be safely ignored without compromising the accuracy of the insights derived from our analysis. This consideration is referred to in the clone detection literature as "clone quality" [40].

Specifically, we consider subtree clones that include nested blocks (e.g., the `while-loop` statement, etc.) and fragment clones (i.e., varying sequences of simple blocks and subtrees), whose minimum size is 8 AST nodes. The analysis considers sequences of up to 10 blocks, which can comprise both simple blocks and subtrees.

For other code smells, we mitigate such subjectivity in choosing the thresholds by relying on a data-driven approach. The two passes of the analysis are required with the first pass extracting the necessary software metrics in the large dataset to determine the appropriate threshold values (e.g., string length and script length across all projects in the large analysis dataset in the case of DS and LS, respectively). We obtain the threshold values similarly to the approach introduced by Lanza and Marinescu [51]. Specifically, we consider the threshold values at the 70^{th} percentile as extreme. Table 3.3 presents the thresholds derived from the large analysis dataset.

3.2.3 Script Addition Metric

This metric measures the differences between the original project and its remix. We extract the added code by using a third-party JSON diff tool, which structurally compares the JSON representations of the original project's source code and that of its remixes. The tool produces a comprehensive list of all basic changes (i.e., add, remove, move, replace, and copy), applying which would transform the original JSON file to that of the remix version.

To reliably approximate the actual extent of code changes between the original projects and their remixes, we filter *remove*, *move*, and *copy* to focus on the *add* and *replace* operations. The *replace* operation replaces a script segment with a new one. Although changes made to the JSON source file can be mapped to 4 basic code element types (i.e., scriptable, script, block, literal value), we found changes with a script as the unit of change to be the most meaningful modification. In other words, we do not count block additions to existing scripts. However, we do count all scripts in the new Sprites added to a project. To calculate the *Script Addition* metric, we sum the add and replace operations, and then divide the result by the total number of blocks, so as to adjust for different project sizes. We use the median of Script Addition to represent the average change rate calculated for each pair of original projects and their remixes. For projects with a high number of remixes, we sample 100 remixes randomly to compute this metric.

3.2.4 Large-scale assessment of software quality

We conduct a large-scale analysis of over one million Scratch projects leveraging the super-computing facility at our institution. To process a large volume of computationally intensive program analysis tasks, the infrastructure makes use of the Hadoop MapReduce framework [19]. Our Java-based analysis tool integrates naturally with Hadoop, achieving the required

scalability by batch analyzing multiple projects concurrently. The analysis results expressed in the JSON format are stored in the MongoDB database for subsequent data analytics, for which we utilize Apache Spark[114], a cluster computing framework for interactive data analysis. Figure 3.1 gives a high-level overview of our analysis infrastructure.

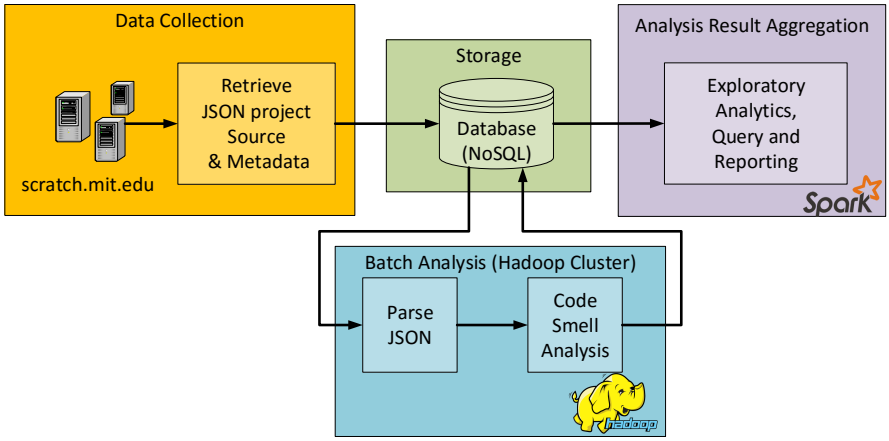


Figure 3.1: Large-scale smell analysis infrastructure for block-based software.

3.3 Results

In this section, we presents the results for each of the research questions posed.

3.3.1 RQ1: What known code smells described in the literature are applicable in the context of Scratch?

We present a catalog of 12 Scratch code smells. For each identified smell, we also point out its variants in other linguistic contexts. When presenting various cut-off thresholds for counting a code pattern as a smell, we make use of the statistical thresholds presented and

explained in Section 3.3. We derive our statistics-based thresholds from a large dataset of Scratch projects following the approach first presented by [51].

Catalog of code smells in Scratch

Classic code smells

1. **Duplicate Code (DC):** A fragment of code is duplicated as a way to reuse existing functionality of code at multiple locations in the program. Other contexts: [90]
2. **Long Script (LS):** An unreasonably long script can suggest inadequate decomposition and hinder code readability. A script is considered too long if there are more than 11 blocks measured vertically. Other contexts: [26, 37]

Object-oriented code smells

3. **Feature Envy (FE)**: A data producing script is in a different sprite from the one that uses the data. If there is a 1-to-1 relationship between such sprites, they should not be separated, having to talk to each other via a global variable. A better design would have a single sprite, with scripts communicating with each other via a local variable. Other contexts: [26, 37]
4. **Inappropriate Intimacy (II)**: A sprite can check on other sprites' attributes through sensor blocks. However, excessively reading of other sprite's private variables (at least 4) can lead to high coupling between sprites. Other contexts: [26, 37]
5. **Middle Man (MM)**: A long chain of broadcast-receive can be used to pass a message from one script to another. However, using this abstraction to simply delegate work without actions is considered a code smell. Other contexts: [26]

End-user code smells

6. **Duplicate String (DS)**: String values are considered duplicate if the same value of at least 18 characters is used in at least 4 different places. Other contexts: [90]

7. **Uncommunicative Name (UN)**: Although other programming entities could suffer a similar effect, this smell focuses particularly on a problematic naming of a sprite —"Sprite" which is highly common name since it is the default name that the Scratch programming environment gives to generic sprites at the creation time. The evidence of this smell is presented in the work by Moreno and Robles[63]. Other contexts: [26]

Interface code smells

8. **Broad Variable Scope (BV)**: A variable is marked as broad scope when the variable is made visible to all sprites, but is only used in one sprite. By following commonly accepted design practices, variables should be made local to the scope that uses it. Proper variable scope helps improve comprehensibility as it tells to which sprite the variable belongs. Too many global variables can also be confusing for programmers trying to find the right variable in the script palette and the drop-down menus. Example of smells in other contexts: [23]

9. **No-op (NO)**: A user event-based script that performs nothing can be removed. A common occurrence is event-handling code with no action associated with it. Other contexts: [37]

Interface code smells (continued)

10. **Undefined Block (UB)**: Scripts can be copied from different projects using the Scratch programming environment feature called "backpack". The scripts with calls to a custom block without its definition will be rendered as undefined blocks, thus ceasing to contribute any useful functionality to the project. This smell is commonly introduced when custom block definitions are not copied and placed first. The rationale of this code smell is similar to *No-op*.

11. **Unreachable Code (UC)**: An unreachable script can be safely removed without affecting the program behavior. A script is considered unreachable if it is the receiver of a nonexistent message. This particular case is often caused by removing only the broadcast blocks without adjusting their corresponding receiver blocks. Note that our analysis disregards the fragment scripts not beginning with event blocks, as they are commonly used by Scratch programmers to experiment with code and to initialize persistent data. Other contexts: [37, 90]

12. **Unused Variable (UV)**: The Scratch programming environment lets a programmer declare variables in the data palette before they can be used in the scripting area. However, the programming environment provides no support to check if the declared variables are unused and can be safely removed. The rationale of this code smell is similar to that of *Unreachable Code*.

	N	Stdv	Min	Median	Max	Pctl.70.	Pctl.80.	Pctl.90.
BV	51.00	14.60	0.00	15.48	50.00	24.70	32.86	38.57
UC	59.00	0.93	0.00	0.00	5.68	0.00	0.15	0.42
DC	59.00	0.46	0.00	0.44	1.98	0.70	0.87	1.08
DS	25.00	23.09	0.00	0.00	100.00	0.00	0.00	28.57
FE	59.00	5.36	0.00	0.00	27.00	0.00	0.00	6.37
II	59.00	0.25	0.00	0.00	1.00	0.00	0.00	0.00
LS	59.00	10.33	0.00	9.18	40.00	14.40	18.90	29.21
MM	59.00	9.50	0.00	0.83	36.00	6.20	11.20	21.20
NO	59.00	20.38	0.00	2.79	101.00	7.77	19.20	37.70
UN	59.00	34.95	0.00	6.51	100.00	38.25	61.61	89.38
UB	59.00	4.09	0.00	0.00	21.00	0.00	0.73	4.20
UV	51.00	25.24	0.00	14.64	95.24	30.83	44.95	61.25

Table 3.4: Summary statistics of code smell of projects in the benchmarks and their 70^{th}, 80^{th}, and 90^{th} percentile values

3.3.2 RQ2: What can the analysis of popular Scratch projects teach us about the state of software quality in this programming domain?

Scratch projects can be cloned or "remixed" in the Scratch terminology. The remixes can be traced back to their original projects. In this study, we examine whether popular projects with high code changes in their remixes tend to exhibit higher software quality.

RQ2.1: *Script Addition* metric value and project's code quality We consider popular projects of medium sizes, as defined as being in the range of between the 25^{th} and the 75^{th} percentiles (200-1,400 blocks) of all project sizes. We consider two project subsets that we refer to as:

1. *high-change projects*: 59 popular remixed projects whose *script Addition* metric is ranked in the top 25;

	Low changes (25%)		High changes (Top 25%)		
	Median	SD	Median	SD	P-values
BV	16.67	33.94	15.48	14.54	0.323
UC	0.00	0.99	0.00	0.92	0.997
DC	0.54	0.48	0.44	0.46	**0.044**
DS	0.00	28.62	0.00	22.69	0.202
FE	0.00	4.53	0.00	5.32	0.795
II	0.00	0.21	0.00	0.25	0.707
LS	15.38	17.58	9.18	10.29	**0.003**
MM	0.00	13.64	1.00	9.46	0.980
NO	2.00	18.89	3.00	20.94	0.968
UN	27.78	44.50	6.51	34.84	**0.011**
UB	0.00	3.77	0.00	4.77	0.847
UV	5.56	24.41	15.28	25.02	0.999

Table 3.5: The comparison of studied code smell metrics between projects with remixed code changes in the bottom 25% (< 0.1) vs. top 25% (> 0.82)

2. *low-change projects*: 87 popular remixed projects whose *script Addition* metric is ranked in the bottom 25;

Table 3.5 reports on the medians of the studied code smell metric values for each of these subsets.

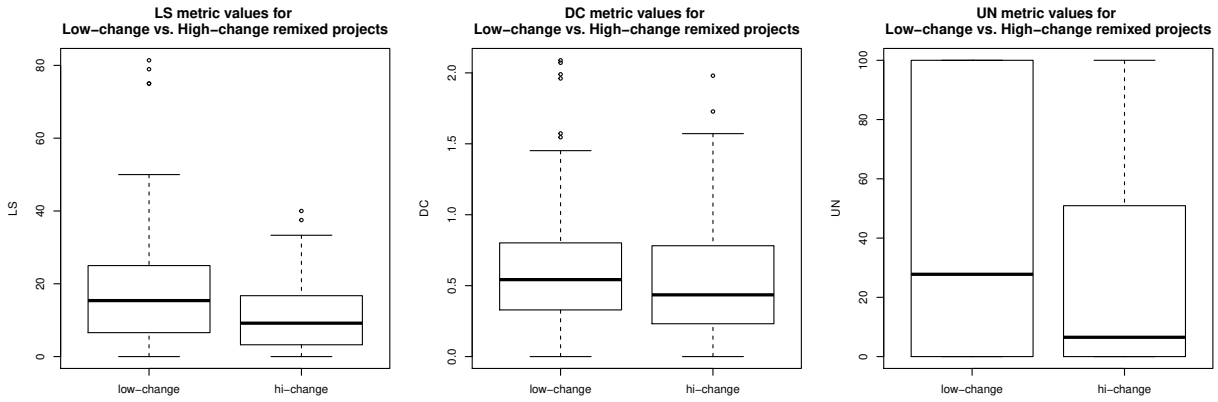


Figure 3.2: Incidence rate of code smells LS, DC, and UN in the two subsets (High-change vs. Low-change) of popular Scratch projects

We visualize the smell incidence in the two subsets using boxplots, and test if they are

drawn from the same distribution (null hypothesis) using a one-tailed Wilcoxon rank sum test[3]. The null hypothesis is that both of these distributions should be the same. Figure 3.2 shows boxplots of the LS, DC, and UN smell metrics for the remixed projects, whose *Script Addition* metrics are in the bottom 25% and the remixed projects whose *Script Addition* metrics are in the top 25%. The boxplots show the trends of a lower smell incidence rate in the *high-change* subset. We omitted the boxplots comparing other code smells in the two subsets for brevity, as they do not show any clear trends and their test values are not statistically significant.

The non-parametric Wilcoxon rank sum test (with continuity correction) for LS, DC, and UN conclusively rejects the null hypothesis that the two subsets (low-change remixed projects and high-change remixed projects) are drawn from the same distribution. Table 3.5 presents the corresponding effect size (difference of medians) and p-values. As can be observed, the medians of UN and LS smell incidence rates in the high-change subset are dramatically lower than that of the low-change subset. DC has a significant but smaller effect size, based on the median difference of the smell incidence rate.

3.3.3 RQ3: How prevalent and severe are different code smells in the general Scratch code base?

We study the prevalence of code smells in the dataset of 594,988 projects, selected from the initial 1,009,192 projects as containing more than 20 blocks as described in 3.2.1. For each code smell, we count the number of projects as being afflicted by the smell if at least one code smell instance is found. We report on the prevalence of smells as the percentage of the smell afflicted projects over the total number of projects analyzed for the smell of interest.

[3]The Wilcoxon rank sum test is a non-parametric test that is not sensitive to outliers, as it does not assume any distribution of sample data.

Smell	N	%Prevalence	%Low risk	%Moderate risk	%High risk	%Very high risk
LS	594,988	51.00	71.13	5.03	8.55	15.30
UN	591,621	48.00	67.36	8.57	5.97	18.10
DC	594,988	35.00	75.86	3.11	3.26	17.77
BV	328,016	30.00	54.86	3.09	3.85	38.20
NO	594,988	28.00	93.71	3.24	1.09	1.96
UV	328,016	20.00	77.51	5.25	6.73	10.50
MM	594,988	17.00	94.02	2.17	1.75	2.06
DS	276,891	13.00	71.71	0.00	14.00	14.29
UC	594,988	9.00	91.30	1.51	1.42	5.77
UB	594,988	8.00	92.29	0.00	5.60	2.11
FE	594,988	5.00	94.63	0.00	2.89	2.48
II	594,988	4.00	95.88	0.00	0.00	4.12

Table 3.6: Prevalence and severity of code smells in the large dataset

Please note that the number of projects considered for each code smell can vary since certain code smells may not be applicable in some projects and thus we exclude them from the calculation. For example, variable-related code smells (e.g., BV and UV) are not applicable for projects that declare no variables.

Based on our analysis, LS, UN, DC, and BV show high prevalence ($> 30\%$); NO, UV, and MM show moderate prevalence $[10\%, 30\%]$; and DS, UC, UB, FE, and II show low prevalence ($< 10\%$). Additionally, we assess the severity of quality problems by categorizing the projects into increasing risk levels. Code smells with low incidence rate may not be harmful. Hence, one must determine the thresholds exceeding which should classify a project as being at risk. Since choosing threshold values can be subjective, we rely on the approach first introduced by [2] that establishes benchmarks for deriving thresholds. Having shown high *Script Addition* metrics associated with likely high quality projects, we use the high-change subset consisting of 59 projects, described in 3.3.2, as the benchmark for deriving practical threshold values of different smell risks.

The summary statistics of the *high-change* subset as well as the 70, 80 and 90th percentiles,

used as the basis for determining the risk intervals are presented in Table 3.4. We base our intervals on [2] to categorize the severity of quality problems using their percentile values: low (0–70%), moderate risk (70-−80%), high risk (80-−90%), and very-high risk (> 90%). Table 3.6 presents the prevalence and the severity of the analyzed code smells.

3.4 Discussion

In this section, we present the discussion of the findings and the threats to validity.

Code changes in the remixes as software quality indication: The low-change projects exhibit software quality that is noticeably worse than their high-change counterparts in the presence of a high incidence rate of *Long Script*, *Uncommunicative Name*, and *Duplicate Code*, known to hinder code comprehensibility and extensibility. Being hard to understand and extend, these projects tend to discourage high-change remixes.

This empirical evidence shows that poor software quality can negatively impact the educational effectiveness of communal learning by hindering project remixing. Remixing is a common practice among Scratch programmers and has been shown to help novice programmers get exposed to new programming constructs and concepts [18].

State of software quality in average block-based projects: For the smells with high prevalence—BV, UN, DC, LS and DS—a larger percentage of average projects is in the very high risk category, which is at odds with the software quality exhibited by the popular projects with high remixability. If the quality of an average block-based project is similar to that of popular projects, we expect to see about 10% of the population distributed into each of the moderate, high, and very high risk categories. However, our results show that average projects have been afflicted by code smells differently. That is, BV–38.2%,

UN–18.1%, DC–17.8%, LS–15.3%, and DS–14.3% are clearly in the very high risk category. Scratch programmers may be simply unaware of these code smells and their harmful effect on program quality. The remaining smells afflict the analyzed projects less commonly.

Threats to validity: The validity of our analysis results may be threatened by several factors. The documented code smells may not cover all code smells, which are known to be subjective, while additional code smells can appear as new language features and development tools are being introduced. The selection of the benchmark projects may not be appropriate for all types of projects (e.g., long scripts are a common feature of storytelling projects). Our benchmark and its derived thresholds aim at representing a broad category of projects, so as to avoid the manual inspection required to include all types of projects. We establish thresholds by observing the impact on the comprehensibility and extensibility, as guided by our *Script Addition* metric. In other words, these thresholds may not be applicable for studying other aspects of software quality (e.g., reusability, maintainability, etc.). To mitigate the risk of the analyzer producing erroneous results, we manually sample if their subsets adhere to the specified analysis metrics.

3.5 Conclusions and Future Work

This work sheds light on the state of quality in block-based software. We provide empirical evidence of quality problems negatively affecting the likelihood of Scratch projects to be remixed and extended. Our large-scale study assesses not only the prevalence of Scratch code smells, but also their severity, presenting conclusive evidence of recurring quality problems in this domain. The results of this work can inform future efforts to support quality improvement practices in block-based programming environments that are aligned with the actual needs of this community.

Chapter 4

Code Quality Problem Trends Among Novice Programmers

4.1 Introduction

To be fully prepared for the challenges of producing high quality software in the real world, students must have been introduced to software quality as part of the curriculum. Nevertheless, the educational community is split on the question of when the right time is to start introducing software quality. Postponing the topic until later in the curriculum often grooms introductory computing learners into undisciplined software development practices. As Will Durant eloquently articulated: "We are what we repeatedly do. Excellence, then, is not an act, but a habit." To truly embrace this principle, we should integrate software quality into the computing curriculum, starting from the first programming course.

Prior studies have uncovered the high prevalence of recurring code quality problems in programs written by introductory programmers [1, 98]. This finding calls for a serious reevaluation of the importance of software quality in introductory CS education. However, the research community possesses limited knowledge about the software quality issues of novice programmers. Closing this knowledge gap has potential to provide valuable insights for computing educators, informing the efforts aimed at creating novel educational interventions that integrate the software quality concepts and practices into the CS curriculum.

In this work, we study a large longitudinal dataset of software artifacts produced by introductory computing learners. Our study's goal is to answer the following research questions:

- **RQ4:** Does the code quality improve, as novice programmers gain experience?
- **RQ5:** How persistent are poor coding practices, as novice programmers gain experience?
- **RQ6:** Which Scratch constructs and idioms reduce the prevalence of code smells in projects written by novice programmers?

To identify quality problems, we adopt the terminology of *code smells*, coding patterns known to indicate possible poor design or implementation choices, the term made popular by Fowler's refactoring book [26]. We analyze a longitudinal dataset of 3,810 Scratch projects, written by a distinct group of 116 novice programmers. For these projects and their programmers, we compute a set of relevant explanatory variables, which comprise the measurements and metrics that potentially associate with the presence of code smells, including programming proficiency and basic programming abstractions and constructs. We apply survival analysis to identify the relationship between a set of explanatory variables and the risk of novice programmers introducing code smells into their programs.

Our results indicate that for *all* levels of programming proficiency, students tend to retain an unchanged attitude toward software quality, irrespective of their current level of programming proficiency. Being exposed to certain programming concepts and constructs lowers the computing learners' vulnerability to introducing some code smells into their projects. However, introducing students to these concepts alone may be insufficient to convince them to embrace software quality. However, if software quality concepts were taught alongside the fundamentals of computing, students would acquire an awareness and practical skills, required to proficiently develop functional computing solutions, while also adhering to well-

established software design and implementation practices.

The rest of the chapter is structured as follows. Section 4.2 describes the methodology applied to answer the research questions. Section 4.3 explains our statistical analysis and its results. Section 4.4 interprets the results and their implications, and Section 4.5 concludes the chapter.

4.2 Methodology

In this section we describe our approach to data collection, the set of explanatory variables derived from the data we collected and *survival analysis*, the statistical analysis technique we apply in this work.

4.2.1 Extracting explanatory variables

The explanatory variables considered in this study are in the form of code smell metrics, basic programming proficiency and measurements of core programming construct usages.

Code smells: encode those coding patterns that are indicative of possible quality problems. Compared to coding styles, code smells are less subjective, and thus work well in the context of the strict visual syntax of block-based software. Amenable to automated analysis, code smells also allow the analysis heuristics to scale. In this work, we analyze projects authored by novice programmers for the incidence of 4 types of common code smells, which are defined in Table 4.1. Our previous work [98] provides additional information about these code smells.

To account for varying project sizes, we calculate a smell metric based on the percentage or density of smell instances, noted in the Table 4.1 (e.g., a density-based metric for *Duplicated*

Code Smell	Definition
Broad Var. Scope (BV)	A variable with its scope broader than its usage (100% of all variables)
Duplicated Code (DC)	Similar blocks in multiple places ($\geq$ 0.34 instance per 100 LOCs)
Long Script (LS)	A long script with LOCs $>$ 11 (33% instance of all scripts)
Uncommunicative Name (UN)	Poor naming started with "Sprite" (100% of all sprite names)

Table 4.1: Code smells studied in this work

Code, and a percentage-based metric for *Long Script*, etc.). We select the 75^{th} percentile of the smell metric values across all projects in the dataset as the threshold for classifying a project as having an "unacceptable" number of a given smell and being *afflicted* by it (e.g., the threshold for *BV* is 100%). It is these projects that serve as the *events of interest* in our survival analysis.

Programming proficiency: To measure programming proficiency of novice programmers, we leverage *Computational Thinking Score (CT Score)*, developed by Moreno and Robles and subsequently evaluated extensively [65]. We calculate six CT dimensions (i.e., abstraction, data representation, flow control, logic, parallelization, and synchronization), disregarding the interactivity dimension, as it is not directly relevant to the general programming proficiency we focus on in this work. The score in each dimension is in the range of 0 to 3, based on the proficiency inferred from the usage of different types of block constructs in the program, as briefly explained in Table 4.2. The total CT score is calculated by summing the partial scores assigned to several CT dimensions. The original work by Moreno and Robles [65] provides specific details how these scores are computed.

Programming vocabulary: We additionally extract explanatory variables based on the number of core programming constructs found in the projects that the subjects created. This

Dimension	Description
Abstraction	`whenCloned` > `procDef` > presence of scripts
Data Representation	Usage of list > variables > blocks
Flow Control	`doUntil` > `doRepeat` and `doForever` > presence of scripts
Logic	Logical operation > `IF-ELSE` > `IF`
Parallelization	Advanced event-based block > less advanced ones
Synchronization	`doWaitUntil` > broadcast and receive > wait

Table 4.2: Dr. Scratch's computational thinking dimensions. Description lists usage patterns in decreasing order of proficiency (from 3 to 1, ordered by >).

approach is based on *programming vocabulary*, as a way to measure learning progression in prior studies including [11] and [18]. This approach relies on the fact that different types of blocks can be mapped to certain computing concepts. In this approach, programming proficiency is directly correlated with the extent of blocks used (the greater variety of blocks used, the higher the proficiency). We consider the usage number of blocks such as procedures, variables and sensore blocks whose detailed descriptions can be found in Table 4.3.

4.2.2 Survival analysis

Originally, survival analysis originated in epidemiology, in which the time to the event-of-interest (survival time) was death. but later has been applied more broadly in different research fields including education (e.g., the effect of remixing or code sharing on student learning progress [18], factors influencing students in massive open online courses to drop [112, 113].). Survival analysis addresses the problem of incomplete information about the survival time, called *censoring*. The data may be missing because a study subject has not experienced the event of interest by the time the observation period ends, thus making the information about survival time incomplete. In this study, we apply survival analysis to study the effect of certain learner characteristics (e.g., poor quality of their past projects)

on the learners' risk of their project being afflicted by a particular code smell.

A standard way to visually explore and understand survival data is the *Kaplan-Meier* plot of survival against time for each study group; it considers one predictor variable at a time while taking into account the censored data. However, for extensive analysis, we use Cox proportional hazards [47], a popular model for multivariate survival analysis. The Cox's hazard ratio (HR) describes the relative likelihood of the event-of-interest by comparing event rates between different study groups, while adjusting for other significant variables. In this study, the ratios indicate how the *relative likelihood* of the event of interest (the presence of code smells in a project) changes relative to explanatory variables (e.g., numbers of past projects that contain smells).

For example, to study how exposing learners to the concept of procedure affects their risk of introducing duplicate code smell, the following are the possible values of hazard ratio:

- *HR = 1*: at any particular time, the event rates are the same in both groups (the factor has no effect).

- *HR = 0.5*: at any particular time, half as many learners, whose past projects use procedures, are likely to introduce the smell, as compared to the learners, whose past projects use procedures one fewer time.

- *HR = 2*: at any particular time, twice as many learners, whose past projects use procedures, are likely to introduce the smell, as compared to the learners, whose past projects use procedures one fewer time.

4.2.3 Data and Analysis

In this section, we describe our approach to data collection, and the set of explanatory variables derived from the data.

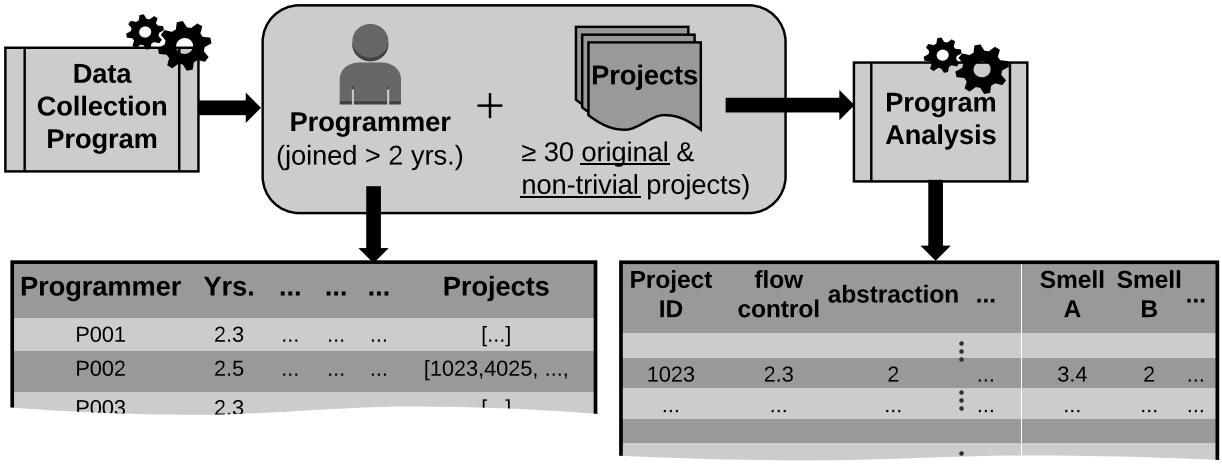


Figure 4.1: Overview of the data collection approach

Scratch provides a convenient access to each programmer's shared projects, and their last modification dates. These shared projects form a longitudinal dataset for our case study. Figure 4.1 provides an overview of the criteria for random selection of the programmers for the study. The first step sequentially checks each project ID for its validity and sharing status, leveraging the fact of Scratch project IDs being auto-generated integers in ascending order. The second step retrieves the author information for all projects whose ID is determined as valid and shared. The following conditions must be met for a user to be included:

1. The author has been a member of the Scratch community for at least two years (the difference between the date of sharing the last project and the date of the user joining the community)
2. The author must have created a minimum of 30 projects, excluding the remix projects, thereby ensuring that each included authors comes with sufficient longitudinal data

Furthermore, non-programming projects, which contain fewer than 20 blocks, are excluded as they may not be sufficiently complex to exhibit code smells and reliably reflect their programmers' proficiency level. This filtering is also applied to prevent common non-programming projects, referred to by Dasgupta et al.[18] as "coloring-contests."

Because Scratch repository only keeps the most recent modified version, all projects shared a long time ago but continuously modified would be excluded. To prevent this mistake, we include all projects, whose last modification date is within 60 days of their sharing date. This provision captures the author characteristics as they move across the different periods of the learning process. These inclusion and exclusion criteria are designed to limit the considered set of authors to those whose projects would yield useful insights for this study.

Once the project and programmer data are collected, we then compute the CT score for each dimension. To meaningfully measure program quality, we need to exclude projects authored by complete novices. We define this group as authors whose code never reaches the proficiency level of 2 across all dimensions. To filter out the cases of "an occasional display of proficiency," we only include those projects in which the high proficiency levels (2 and 3) are demonstrated at least three times. We found this heuristic to effectively identify the projects that are worth analyzing in our study. Assuming that programming proficiency is a continuously increasing metric as learners author additional projects, we include in our *analysis dataset* all the subsequent projects once the sought-for level of proficiency is demonstrated. We developed several automated program analysis routines to extract relevant metrics for each project. These metrics represent the explanatory and outcome variables of interest used in our analysis model.

Explanatory variables: Similar to simple linear regression, independent variables or explanatory variables are used to predict the dependent variable, or the outcome variable of interest. As mentioned above, the *Cox model* assumes the time independent property of the

Variable Name	Description
prior_{*smell*}	# of *smell* afflicted projects in the first 10 projects
CT_overall	Median CT scores across all dimensions
CT_{*dimension*}	Mean CT score for a *dimension*
numLocalVar	Median local variables
sensorBlock	Total number of times sensor blocks are used for accessing other sprites' private fields
numProc	Median number of custom blocks (procedures) created

Table 4.3: Predictors used in the analysis model

explanatory variables. We use the first 10 projects in the analysis set of each programmer as the baseline, with the assumption that in the very beginning, programming proficiency as well as programming habits and practices change little and are thus negligible. We compute the average values for the measurement and the metrics of the first 10 baseline projects to represent the baseline information for each programmer. For most of the measurement and metrics, we calculate the median since the metric data are skewed making arithmetic mean inappropriate. Table 4.4 gives a summary statistics of the baseline data.

Outcome variable: The number of projects until the smell afflicted project (exceed the 75^{th} percentile threshold) is used as the time to the event rather than the physical time, a similar approach used in [18]. The smell afflicted projects are identified from the 20 consecutive projects in the analysis dataset, following the initial set of 10 baseline projects. Since programmers may create multiple smell afflicted projects over time, we use a counting process [47], a common approach for modeling the recurring events. This model allows each smell event for the same programmer to be considered independent from each other and contributes to the risk analysis.

We study each code smell using a separate model for each. All statistical analysis are performed using R [74] and particularly the survival library [100]. Please refer to Appendix

A.2 for more information on how we apply survival analysis to analyze our data.

Statistic	Mean	St. Dev.	Min	Median	Max
prior_BVS	2.8	2.1	1	2	11
prior_DC	5.2	2.5	1	5	12
prior_LS	5.0	3.1	1	4	15
prior_UN	7.1	3.9	1	7	15
CT_overall	1.4	0.3	0.9	1.4	2.1
CT_parallel	0.9	0.3	0	1	2
CT_dataRep	1.6	0.6	1	1.5	3
CT_abstraction	1.4	0.7	0	1	3
CT_sync	2.2	0.8	0	2	3
CT_flowControl	2.1	0.4	1	2	3
CT_logic	1.6	1.4	0	1	3
numLocalVar	0.2	0.7	0	0	4
sensorBlock	6.2	17.4	0	0	113
numProc	0.2	0.9	0	0	7

Table 4.4: Baseline characteristics of the first 10 projects in the analysis dataset

4.3 Results

In this section, we describe the Cox model used to study the effect of different explanatory variables on the risk of a programmer introducing code smells. We study each code smell using a separate model. The Cox regression models allow adjustment of other explanatory variables in the model, while varying the explanatory variable of interest to ascertain its effect size.

Table 4.5 presents the models and their results. We describe each model, its results, and how the results answer each of the 3 research questions above as follows:

Model	Variable	Hazard Ratio	p-value
BV	prior_BV	1.11	0.005*
	CT_overall	0.48	0.059
	CT_dataRep	1.74	0.001*
	numLocalVar	0.96	0.819
	sensorBlock	0.99	0.045*
DC	prior_DC	1.1	0.000*
	CT_overall	1.08	0.698
	CT_flowControl	1.55	0.000*
	CT_abstraction	0.98	0.768
	numProc	1.02	0.642
LS	prior_LS	1.08	0.000*
	CT_overall	0.12	0.693
	CT_sync	0.78	0.004*
	numProc	0.98	0.725
UN	prior_UN	1.15	0.000*
	CT_overall	0.77	0.264

Table 4.5: The statistics of the effect of each predictor variable on the learners' risk of introducing smells.

4.3.1 RQ4: Does the code quality improve, as novice programmers gain experience?

To answer this research question, we include the programming proficiency score as a predictor in each of the models. We control for programming proficiency in each of our analysis models by using the average CT_overall. The results in Table 4.5 show that the CT_overall has no statistically significant effect on the likelihood of novice programmers introducing code smells into their projects. However, gaining programming proficiency in certain CT dimensions can increase the risk of code smells. Specifically, the results show that the increased score for CT_dataRep raises the likelihood of the learner's program to be afflicted by the *BV* smell. *CT_dataRep* has a hazard ratio of 1.74 with statistical significance at 5% level indicated by *. This result suggests 1.74 or 74% more risk relative to the group of learners whose scores are one fewer, when other variables are held constant. Additionally, the increased score of *CT_FlowControl* raises the likelihood of the projects being afflicted by *DC* smell.

4.3.2 RQ5: How persistent are poor coding practices, as novice programmers gain experience?

To answer this research question, we include the prior exposure of a smell, defined as the number of baseline projects being afflicted by the code smell (e.g. *prior_BV*). Table 4.5 shows how the increased prior exposure to each of the smells has a statistically significant effect on the novice programmers' proneness toward introducing the smell. For example, prior_BV with HR=1.11 can be interpreted as: for one additional *BV* afflicted project, the programmer has an increased risk of 1.11 times (11%) the risk of those having one fewer *BV* afflicted projects. The Kaplan-Meier curves in Figure 4.2 visualize varying risks faced by each learner group. The group of programmers with *BV* afflicted projects in the range (0-3]

Code Smell	Scratch Concept/Constructs
BV	Local variable to Sprite, Sensing blocks
DC	Loop iteration, Cloning
LS	Synchronization (broadcast/receive), Procedures
UN	N/A

Table 4.6: Code smells alongside programming concepts/constructs that can reduce their incidence

are less likely to introduce *BV* smells, considering that more projects created by this group remained unafflicted compared to the other two groups, across the observation period.

4.3.3 RQ6: Which Scratch constructs and idioms reduce the prevalence of code smells in projects written by novice programmers?

This research question explores how programming concepts/constructs, of which novice programmers may be unaware, can improve their program quality. We take into account some specific Scratch language features. Table 4.6 shows code smells and corresponding programming concepts/constructs that are relevant for minimizing each of the smell. These programming concepts/constructs serve as the explanatory variables and are also included in the analysis models.

The results in Table 4.5 show a few programming concepts/constructs appear to naturally induce quality improving practices among novice programmers. Specifically, novice programmers with higher CT_sync have a reduced risk of *LS* afflicted projects (i.e., HR=0.78, or a reduced risk of 22% relative to the group whose CT_sync is one fewer). This finding confirms the observation about scenario-based programming [29], which is captured by CT_sync. This programming style fosters modular thinking, which leads to short scripts.

Novice programmers, who have more exposure to the use of sensor blocks in the past, are slightly less likely to introduce the *BV* code smell (i.e., HR=0.99, or a reduced risk of 1% relative to the group less exposed to sensor blocks). Sensor blocks make it possible to read the local variables of other sprites, similar to getter methods to access private fields. However, having been exposed to local variables alone fails to translate into reducing the incidence of *BV* afflicted projects.

Our results show no association between the usage of procedures and the lower risk of *DC* afflicted projects. Upon further investigation, we discovered that very few novice programmers in the dataset have ever used procedures (custom blocks in Scratch).

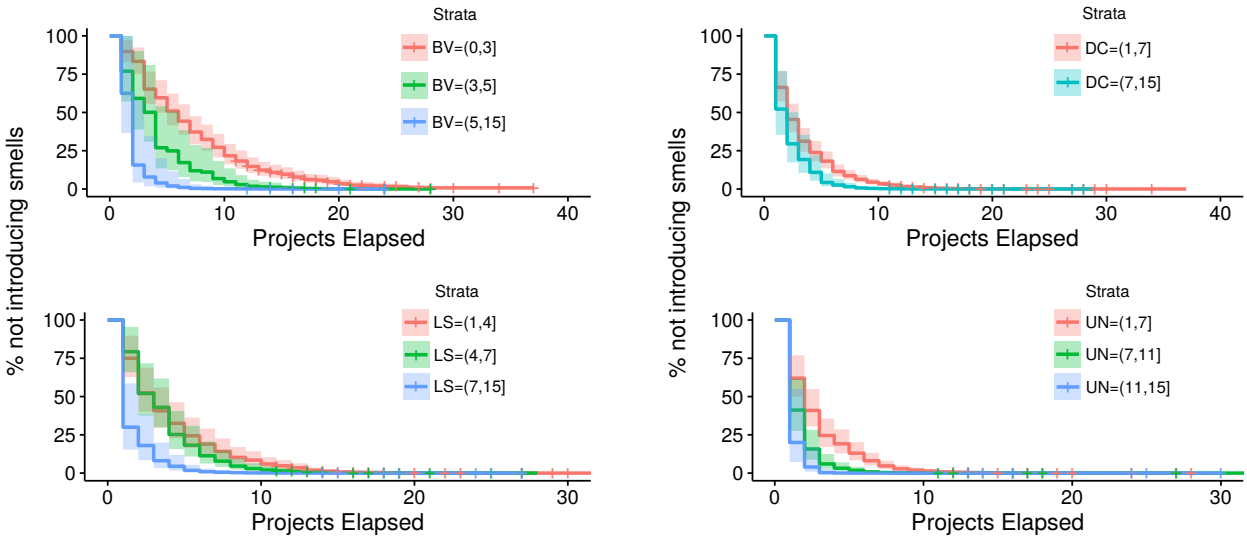


Figure 4.2: KM survival curves: Programmers with a high number of prior smell-afflicted projects incur higher risks of introducing the same smells.

4.4 Discussion

This section discusses our findings and their implications.

Informal learning Our results raise questions about the nature of programming practices fostered by informal programming learning environments, which have become increasingly popular in recent years. In these environments, introductory learners are encouraged to freely explore and learn on their own and from projects shared by others. This way, students move quickly to gain programming proficiency. However, as our results indicate, an increase in programming proficiency does not necessarily translate into proper programming practices, which emphasize the importance of software quality as an important objective.

Persistent poor quality practices Prone to introducing certain code smells into their code, novice programmers continue this trend, as their code continues to be afflicted with the same code smells. These poor quality practices likely need an educational intervention that focuses on how students can avoid introducing them in the first place. Discussing how certain programming practices are considered improper and how to improve upon them can be an effective pedagogical strategy for equipping students with knowledge and skills required to improve software quality.

Opportunities for educational intervention Using certain computing concepts and programming styles can be conducive to decreasing the vulnerability of introducing some code smells, as the protective effect of CT_sync on LS seems to suggest. This insight suggests that effective educational interventions can introduce known effective programming concepts and practices as a way to improve software quality.

Threats to validity Our results might not generalize beyond the context of the study (e.g., the studied subjects beyond the specified criteria, different programming languages, educational settings, etc.). Because of our dataset's properties , the results apply mostly in the context of informal CS education and block-based programming pedagogy. The validity

of our findings may be affected by other factors, not considered or impossible to measure in this study, such as age, the setting (i.e., formal / informal settings for CS Education).

4.5 Conclusion

Our ultimate objective is to design effective educational interventions to promote the culture of quality and quality improving practices among the introductory computing learners. As a first step in this effort, we conducted this study to understand software quality in the context of novice programmers. Specifically, we strive to understand all the different factors that may affect the software quality of the code written by introductory programmers.

We apply survival analysis to identify the effect of various factors on the programmers' risk of introducing recurring quality problems, known as *code smells*. Our findings show that novice programmers prone to introducing some smells continue to do so even as they gain experience.

These findings indicate the need of promoting the culture of quality from the ground up. To that end, novel educational interventions should be able to instill the importance of software quality in introductory computing learners. By incorporating these insights, novel educational interventions can be designed to seamlessly integrate the core computing concepts with disciplined software development practices, while ensuring that these topics are introduced at the level appropriate for introductory learners.

Chapter 5

Systematic Code Quality Improvement for Scratch

Block-based programming plays an essential role in realizing the vision of *CS for All* [87], which renders computing accessible to the broadest possible audience of programmers, many of whom are learners and end-users. A highly popular block-based programming language is Scratch, whose general design principles strive for "a low-barrier to entry" for beginners and "a high-ceiling" for maturing programmers to create increasingly sophisticated programs over time [75]. Nevertheless, the Scratch community's main focus thus far has been on its "low-barrier to entry," with new command blocks that accommodate a wider audience by rendering programming accessible and attractive. Perhaps as an unintended consequence, the efforts to push the "ceiling" higher have been somewhat deprioritized. Left to their own devices, experienced programmers do get to work on advanced sophisticated projects, but at the cost of their software growing uncontrollably in size and complexity, with the overall code quality becoming degraded over time.

As it turns out, the issues of code quality reduce the pedagogical effectiveness of block-based programming [34, 98] as well as the attractiveness of blocks for serious end-user programming pursuits [68]. Consequently, this programming community can no longer afford to neglect the issues of code quality and its systematic improvement. To that end, support for improving code quality can play an important role in elevating the "ceiling" of block-based

programming, while also transforming code quality improvement into a regular practice of novice programmers and end-user developers.

As first steps on the way to improving quality, several recent research efforts have focused on identifying recurring quality problems in block-based software [37, 98]. However, once faced with the identified quality problems, a programmer needs to decide whether to spend the time and effort required to fix them. In fact, evidence suggests novice programmers in this domain refrain from engaging in quality improvement practices [78], despite being sufficiently proficient in programming to improve their code. Across all levels of expertise, novice programmers have been observed introducing a high number of quality problems in their code [95].

Integrating quality improvement practices into the software development process can be prohibitively expensive for professional software developers, let alone novice programmers and end-users. For text-based languages, automated refactoring has become an indispensable quality improvement tool [66]. Refactoring systematically improves code quality, while keeping its functionality intact, ensured by its sophisticated program analyses and behavior-preserving program transformations. Alas, major block-based programming environments only support the most rudimentary `Rename` refactoring, which removes the *Uncommunicative Name* code smell [98]. However, other highly recurring quality problems (e.g., code duplication) make block-based projects hard to understand, modify, and reuse. To be able to improve the code quality of their projects, programmers need well-documented refactorings for Scratch and programming support, which both identifies refactoring opportunities and applies them automatically. Systematically supporting Scratch programmers in improving the code quality of their projects can increase the pedagogical and productivity benefits of this programming domain.

To address this problem, we added automated refactoring support to Scratch 3.0, validated

by implementing four refactorings that remove code smells, reported as *highly recurring* in prior studies [37, 98]. `Extract Custom Block` puts the repeated sequences of statement blocks into a new procedure, invoked in place of the sequences. `Extract Parent Sprite` removes duplicate sprites (programmable objects) by introducing a special construct that clones a sprite multiple times. `Extract Constant` replaces recurring constant expressions with a new variable, initialized to that expression. `Reduce Variable Scope` changes a variable scope accessibility from all sprites (default setting) to a given sprite.

With the exception of `Extract Custom Block`, these refactorings would be novel even without automated application. They provide the Scratch community with a vocabulary to communicate about semantic preserving transformations that improve code quality. These refactorings are also uniquely applicable to Scratch, as it is this language's domain-specific features and distinct programming practices that cause the code smells these refactorings remove. `Extract Custom Block` does have counterparts in text-based languages, but to correctly carry out this refactoring in Scratch requires verifying a different set of refactoring preconditions.

To determine the potential usefulness of the introduced refactorings, we apply them to a representative sample of 448 Scratch projects in the public domain. We ran a user study[1] to investigate whether the availability of our refactoring impact the studied participants improving code quality and their opinion about code quality and our refactoring tools.

The evaluation results show overall high applicability of the introduced refactorings when applied to the detected code smells previously shown to be highly prevalent in the Scratch codebase. Each refactoring positively impacts its respective software metrics, which improve various code quality attributes, including program size, comprehensibility, modifiability, and abstraction. Our user study results suggest that the presence of actionable improvement hints

[1]VT IRB approval was obtained for the human studies described in this study.

and the associated refactorings increase programmers' willingness to improve code quality. To the best of our knowledge, this work is the first effort to introduce automated refactoring to Scratch. By describing the design, implementation, and evaluation of our approach, this chapter makes the following contributions:

1. A catalog of four refactorings for Scratch that removes highly recurring code smells.
2. An intuitive user interface for refactoring, whose actionable and contextualized coding hints encourage programmers to engage in improving code quality.
3. An experimental study that evaluates the applicability and utility of the introduced refactorings.
4. A user study that investigates the impact of our refactoring tools on programmers.
5. A software architecture and reference implementation of a refactoring engine for Scratch 3.0, featuring program analyzers and automatic code transformation.

The rest of the chapter is structured as follows. Section 5.1 presents a catalog of Scratch refactorings. Section 5.2 describes our automated software refactoring support. Section 5.3 describes the methodology to evaluate our approach. Section 5.5 discusses the significance of the evaluation results. Section 5.6 presents concluding remarks and future work directions.

5.1 Refactoring Catalog

Next we present our refactoring catalog. For each refactoring, we list its rationale and preconditions. Due to space limitation, we only illustrate the affected program parts before and after the application of complex refactorings.

5.1.1 Extract Custom Block

This refactoring creates a procedure, whose body comprises the repeated code fragment being refactored, and replaces all occurrences of this fragment with the appropriately parameterized invocations of the created procedure.

Precondition: For behavioral preserving transformation, each argument to be parameterized must be a constant and can be parameterized. Note that some blocks accept a drop-down option value and cannot be parameterized. Finally, the extracted fragment must not contain the control flow terminating command (i.e., "`stop <this script>`" block).

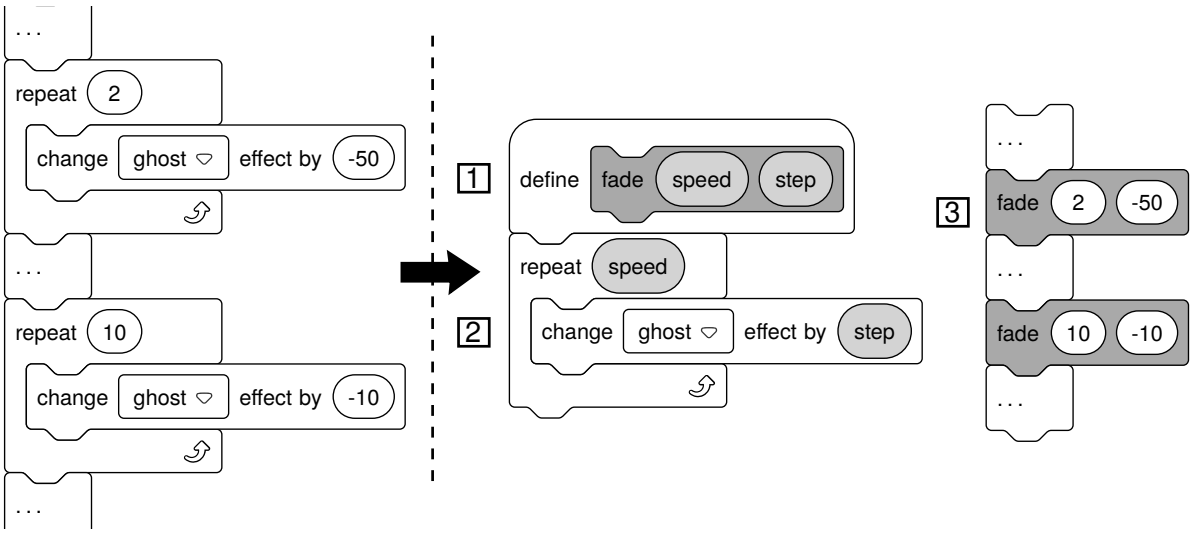


Figure 5.1: Extract Custom Block

5.1.2 Extract Parent Sprite

This refactoring removes duplicate sprites by extracting the parent sprite which instantiates its children clones using "`create clone of <target>`" block[2]. Encapsulated within a sprite, the same code is not only easier to modify, but is also amenable to other localized refactorings

[2] https://en.scratch-wiki.info/wiki/Cloning

(e.g., `Extract Custom Block`, `Extract Constant`, etc.). Automated refactoring cannot remove sprite duplicates in all cases. Some of the removals require adding boilerplate code, which would be hard to generate automatically and require advanced programming expertise to understand. Hence, the refactoring presented herein is applicable when sprite duplicates share similar code (usually at the beginning when a duplicate has just been introduced). Note that the "hide" block is immediately executed in the parent sprite, so as to emulate an invisible prototype object, whose only purpose is to clone visible children.

Preconditions: Sprite duplicates have identical set of scripts (exact code duplication, without variation in literals and identifiers (e.g. variable references)). Each sprite duplicate contains no scripts starting with the "`when I start as a clone`" block. Finally, this Each sprite duplicate uses a single costume.

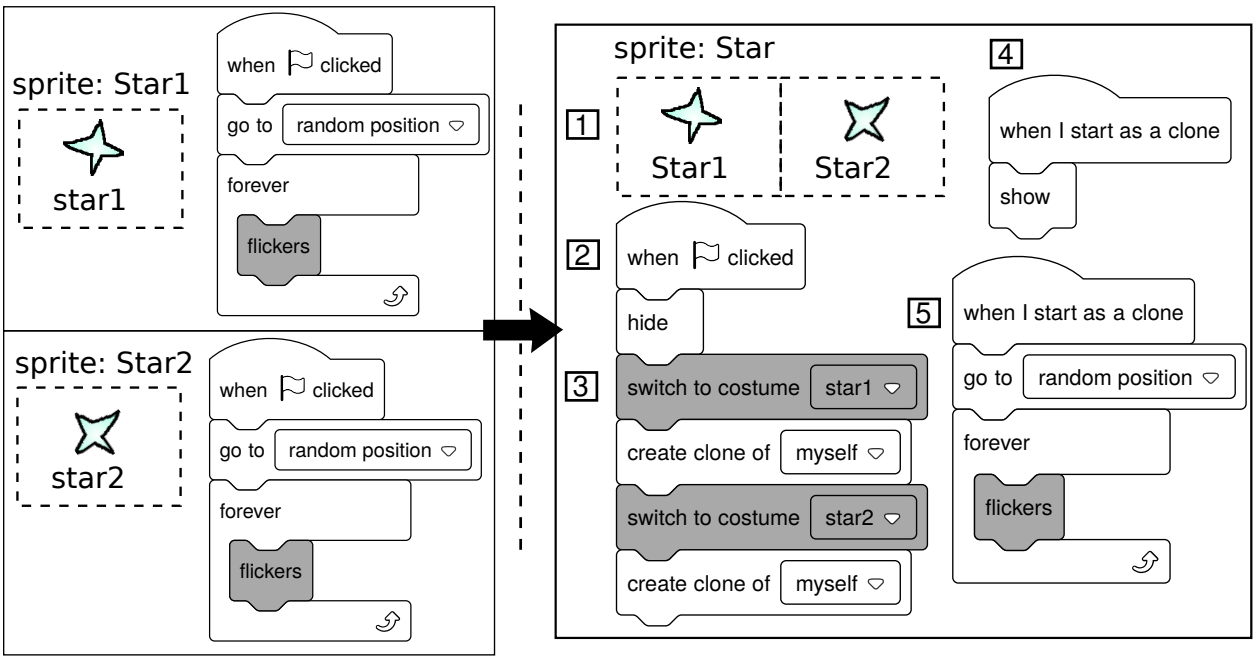


Figure 5.2: `Extract Parent Sprite`

5.1.3 Extract Constant

This refactoring replaces replicated constant values with a variable. Descriptively named variables improve comprehension and modifiability [26]. The only *precondition* is that the replicated values must be of type literal.

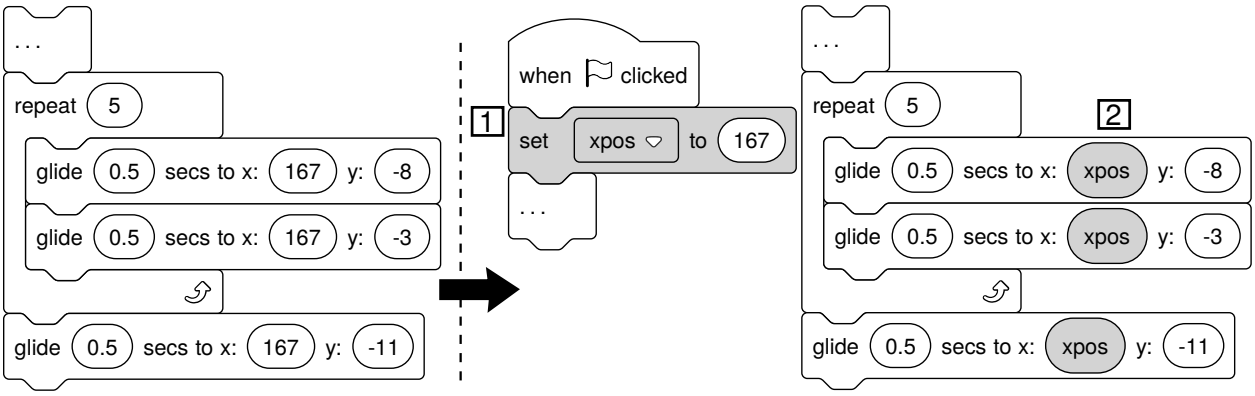


Figure 5.3: `Extract Constant`

5.1.4 Reduce Variable Scope

This refactoring changes the scope of an existing variable from being accessible to all sprites to only a given sprite. If global scope for a variable is not needed, reducing its scope improves the sprite's data encapsulation.

Preconditions: Only one sprite modifies the rescoped variable (though it can be read by multiple sprites)

5.2 Refactoring for Scratch

Although we introduce automated refactoring to Scratch, our general architecture and design can be applied to any block-based programming environment.

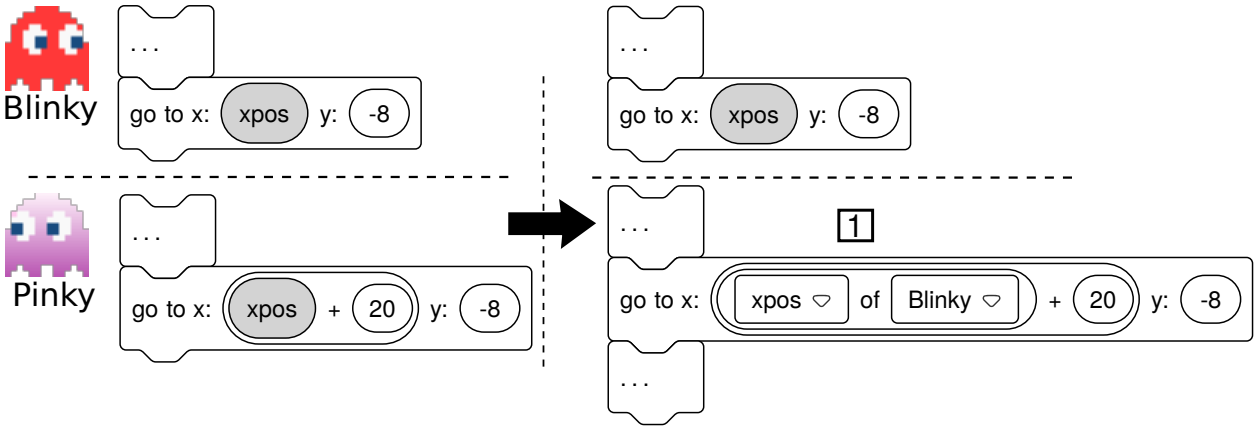


Figure 5.4: `Reduce Variable Scope`

Overall Architecture: Exposed as remote services, the required program analysis and transformation functionalities integrate non-intrusively. Passed a serialized form of the edited program as input, these services analyze and detect code smells, returning the computed refactoring transformations.

Implementation: Based on its input parameters, the *refactoring engine* analyzes and transforms the edited program. The refactoring parameters can be specified by the programmer or in our case automatically extracted from the smells (e.g., `Duplicate Code` → `Extract Custom Block` request). Before performing any transformations, the *refactoring engine* determines whether a given refactoring request satisfies all of its preconditions. In the transformation phase, the *refactoring engine* modifies the analysis AST, while recording each modification as a *transformation action* (see Figure 5.7 for examples). Having been transferred back to the client-side, this atomic sequence of actions is applied to the *program model*, maintained by the block-based programming environment. The actions are applied in the specified order, as each of them modifies program state.

Fig. 5.5 shows some transformation action types used to implement `Extract Constant`. Each action is persisted, so the client can replay the corresponding transformations on the client-side's *program model*. Our design assumes all program elements (both blocks and

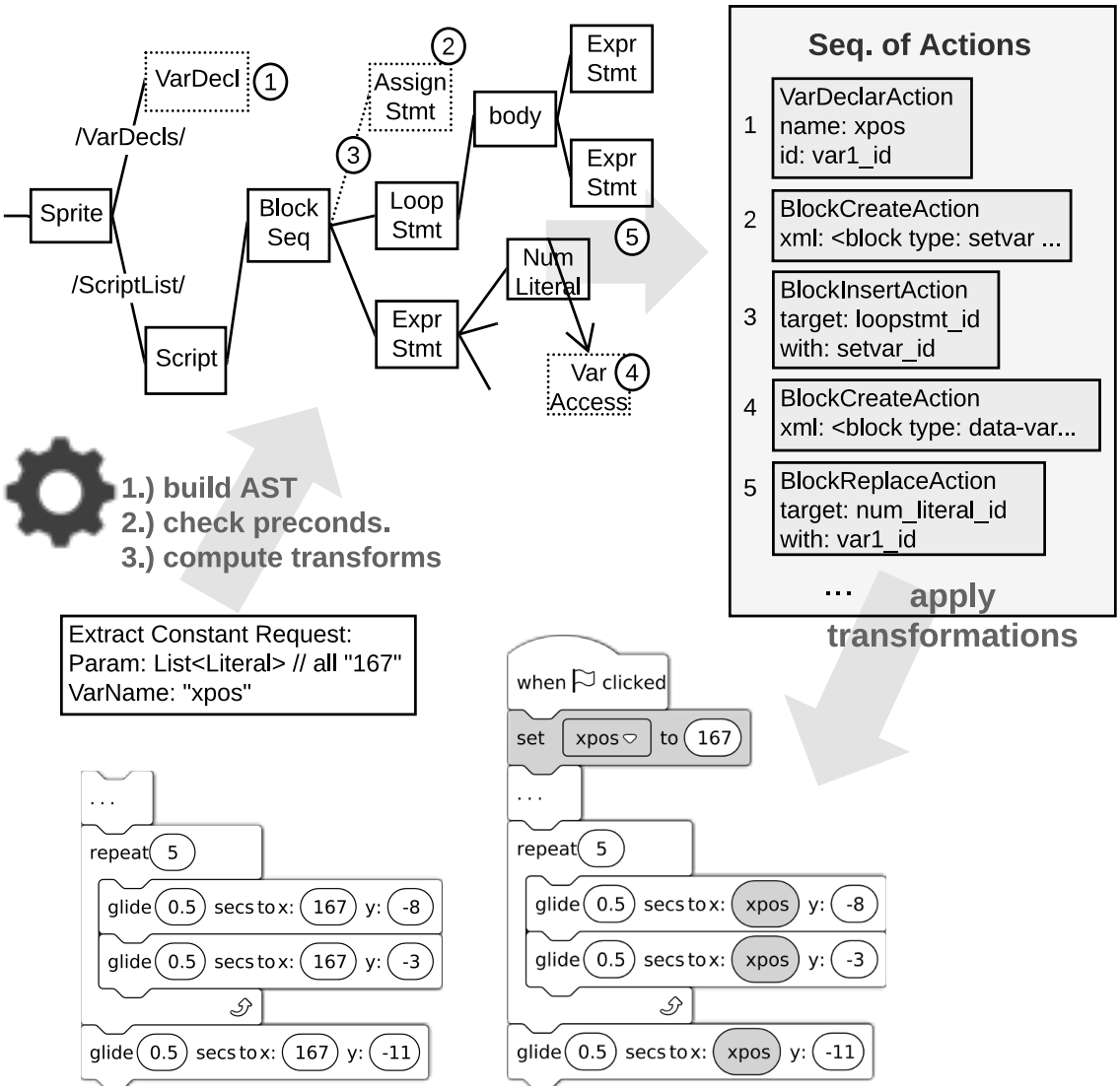


Figure 5.5: Different stages of `Extract Constant` refactoring

non-blocks) can be looked up based on their string IDs, so program changes can be mapped across representations. Additionally, the blocks editor can serialize and deserialize its internal *program model* (e.g., in XML or JSON data format).

Our program analysis and transformation operate on an AST by means of JastAdd [33], a powerful framework for language processing. We express various relationships between program elements in a *program graph* with its declarative specification language to augment

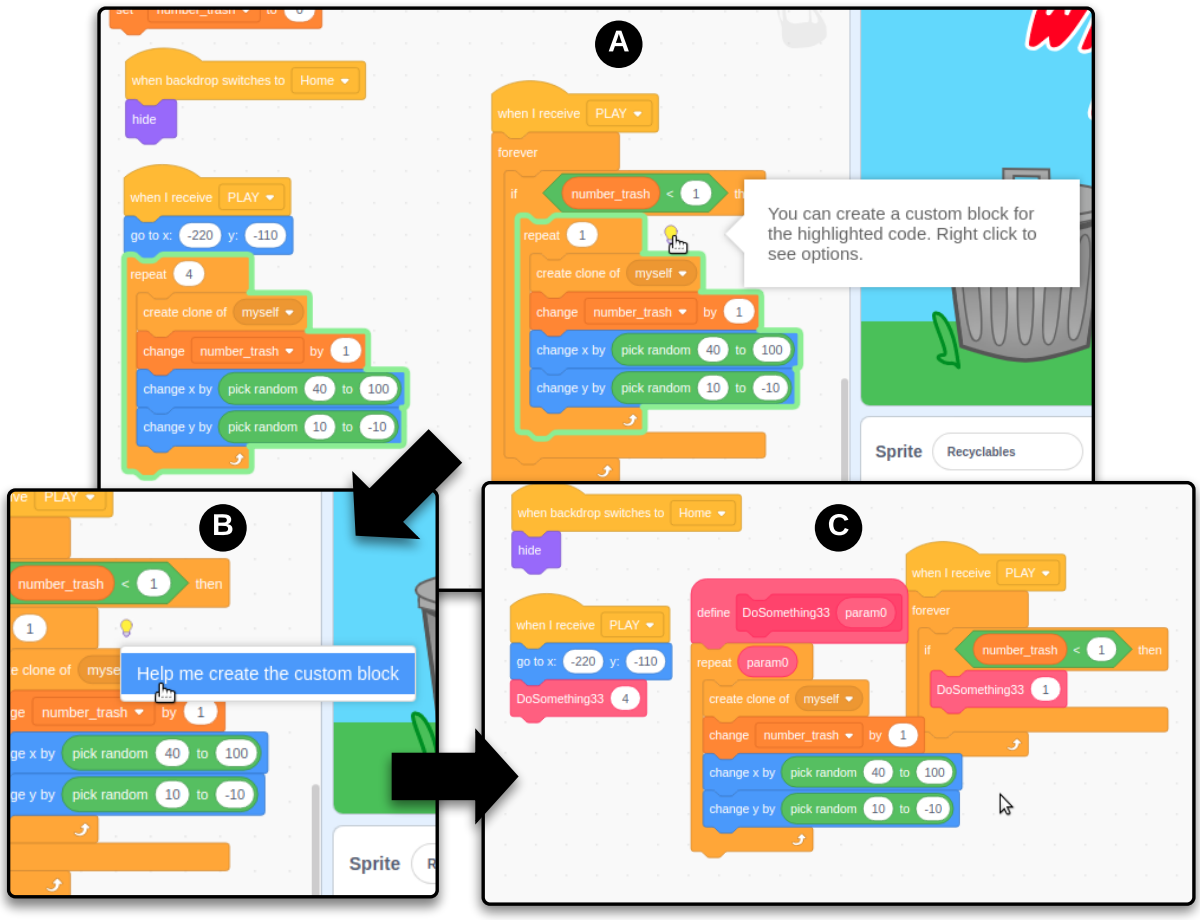


Figure 5.6: A screenshot of the EXTRACT CUSTOM BLOCK refactoring invocation interface for Scratch

the AST classes.

Fig. 5.5 illustrates the major phases of refactoring with an example of performing `Extract Constant`. The first phase starts with a refactoring request, whose parameters for `Extract Constant` comprise all the block's IDs of all duplicate literals and the edited program. To determine if all preconditions are met, the server-side refactoring engine executes various analysis routines (e.g., "check preconds") on the parsed AST. Then, the engine computes ("compute transforms") and record a sequence of transformations (i.e., "Seq. of Actions") that put the refactoring into effect. The resulting transformation actions are serialized

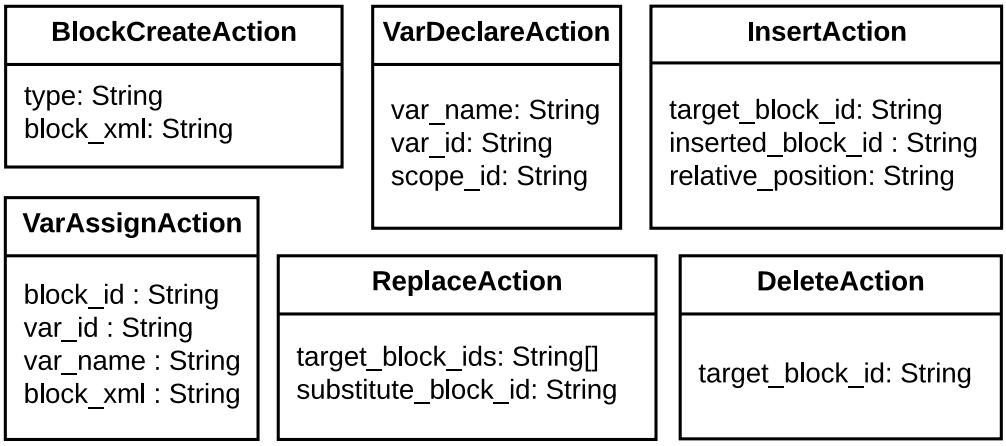


Figure 5.7: Example of transformation actions and their properties

and returned to the blocks editor, which presents the discovered smell hints along with the associated refactoring transformations ("apply transformations").

Refactoring Interface Design: While experienced programmers eagerly refactor their code, novice programmers are unfamiliar with the practice. Hence, the latter's willingness to refactor needs to be encouraged with a friendly and intuitive user interface. Refactoring starts from identifying code whose quality can be improved, a hard task that is even harder for novice programmers. To render refactoring accessible to our target audience, we follow two key design principles, also demonstrated in the screenshot in Fig.5.6, an example of applying `Extract Custom Block` refactoring to a real-world Scratch project.

1. *Code smells should be presented as improvement opportunities to the programmer.* Fig. 5.6(A) displays a code hint as a light-bulb icon, indicating an opportunity for improving code quality (`Extract Custom Block` in this case). Whenever possible a hint should be visually contextualized. For `Extract Custom Block` refactoring, our refactoring interface highlights duplicate code blocks.

2. *Refactoring should be immediately actionable.* Instead of relying on the programmer to specify the required refactoring parameters, as in traditional refactoring, the in-

frastructure should present only the actions ready for the programmer to act upon. Fig. 5.6(B) shows *"Help me create the custom block"*, the only available action for this hint in a simple terminology that can be easily understood by novice and end-user programmers.

Note that in this example, an additional refactoring hint, shown after the application of Extract Custom Block, suggests to the programmer that the just-extracted custom block should be meaningfully renamed.

5.3 Methodology

Automated refactoring can become helpful for novice and end-user programmers in improving the quality of their projects, as long as the refactorings are applicable, useful, and accessible for this programming audience. Our evaluation seeks answers to the following questions:

RQ7: How applicable is each introduced refactoring?

RQ8: How do the refactorings impact code quality?

RQ9: Does the presence of automated refactoring support affect how willing novice programmers are to refactor their code?

To answer **RQ7** and **RQ8**, we experimentally evaluate a representative sample of Scratch projects. We refactor the code smells, automatically detected by our infrastructure's smell analysis modules. To answer **RQ9**, we conduct an online between-subjects study, in which participants in the treatment group interact with our refactoring infrastructure and answer survey questions.

5.3.1 Experimental Evaluation

We limit the scope of our evaluation to highly prevalent smells and whether our refactorings can remove them. Note that some refactorings can remove more than one type of code smell (e.g., `Extract Custom Block` can remove both `Long Script` [98] and `Duplicate Code` smells). Hence, if we were to apply the available refactorings to remove all automatically detected smells, such an evaluation strategy would distort the applicability results and the refactored code quality, as some of the detected smells may not be indicative of actual quality problems (e.g.,what constitutes a `Long Script` is highly subjective). To avoid such distortions, our evaluation considers the following fixed `smell`→`refactoring` pairs: `Duplicate Code`→`Extract Custom Block`, `Duplicate Sprite`→`Extract Parent Sprite`, `Duplicate Constant`→`Extract Constant`, and `Broad Scope Variable`→`Reduce Variable Scope`.

Evaluation Dataset: To assess how viable the refactorings are, we measure the applicability and impact of applying them to third-party Scratch programs. An API request[3] to MIT's Scratch service retrieves a list of projects, divided into two categories of approximately equal size: (1) trending and (2) recent. This subject selection strategy ensures that we conduct our evaluation on a diverse sample of projects created by the Scratch community. We collected a total of 448 projects, 51% among them were viewed at most once, with the rest of projects were viewed on average 12,749 times. Among the subject projects, 88% were remixed at most once and the rest were remixed on average 93 times.

RQ7. Refactoring Applicability: For each refactoring, we assess its applicability by calculating the percent of its associated smells that are refactorable. Because code smell definitions affect the applicability of refactorings, Table 5.1 lists the considered smells and their detection criteria as the bases for interpreting our evaluation results.

[3]https://scratch.mit.edu/explore/projects/all/<recent>|<trending>

Code Smell	Definition and Detection Criteria
Duplicate Code	2 or more code fragments, containing more than one statement, are duplicate if they have identical structure except for variations in identifiers and literals (type II in clone classification [81]). If multiple duplicate fragments overlap, the largest is selected.
Duplicate Sprite	2 or more sprites are duplicate if each script within one of the sprites is duplicated in the others.
Duplicate Constant	Exact literals of at least 3 characters that are replicated at least twice (the thresholds identified experimentally to reduce false positive results)
Broad Scope Variable	A variable declared in the global scope (Stage), but only modified its value locally in a single sprite

Table 5.1: Code Smell Definitions

Metric	Definition
LOC	# statement blocks within a program
Complex Script Dens.	% of scripts (including procedure) with McCabe's cyclomatic complexity [57] value > 10 (risk threshold according to [10])
Long Script Dens.	% of scripts (including procedure) with $LOC > 11$ LOC (threshold empirically determined in previous work [98])
Procedure Dens.	# procedures within a program per 100 *LOCs*
No. Literals	# literals (numbers and strings) within a program
No. Global Var.	# global variables
No. Create Clone Of.	# `CreateCloneOf<target>` blocks

Table 5.2: Metrics Definitions

RQ8. Refactoring Impact on Quality: To assess how each refactoring impacts program quality, we apply all the evaluated refactorings in sequence on each of the detected smell type instances. We then compute the relevant software metrics of the original and the refactored versions of each subject program, so as to determine the difference or the delta in code quality, which serves as a measure of refactoring quality impact. Table 5.2 defines the software metrics used.

Size We expect duplication-eliminating refactorings to remove redundant code and decrease the code size of the afflicted projects. We observe that `Extract Custom Block` reduces a varying level of code size. A small improvement in code size is due to the small number of repetitions detected more frequently than bigger ones. Thus, most projects see a mean decrease in LOC by 3.38%. Though less applicable, `Extract Parent Sprite` refactoring removes large duplications at the sprite level. A subset of projects afflicted by `Duplicate Sprite` see a greater mean decrease in LOC by 8.49%.

Comprehension We expect `Extract Custom Block` to help shorten some long scripts and reduce the number of complex scripts due to the duplicate parts in the original scripts being extracted. We observe 4.4% decrease in the number of long scripts. On the other hand, we only observe a slight improvement in terms of the reduction in the number of complex scripts (5.77%) indicating that most refactorable `Duplicate Code` smells are *not* located within complex scripts.

Modifiability Although the software metrics literature still lacks a definitive metrics known to faithfully capture code modifiability, we can still reason about certain code modifiability improvements by measuring the number of repeated functionalities that have become localized in a single reusable program unit (i.e., procedure for `Duplicate Code`, parent sprite for `Duplicate Sprite`, and variable for `Duplicate Constant`). In Table 5.4, the *Group Size* characteristic of these refactored duplication-related smells reflects the number of locations a programmer needs to navigate to make similar changes in the duplicate parts.

Abstraction Duplication-eliminating refactorings have an obvious impact on abstraction (i.e., `Extract Custom Block` increases procedural abstraction, `Extract Parent Sprite` increases object abstraction, and `Extract Constant` increases uses of variable, a basic data

abstraction). Lastly, `Reduce Variable Scope` improves information hiding or encapsulation, which correlates with the increase in the usage of local variables. The result indicates the refactored projects (N=43) which have used local variables at least once could see an increased usage of local variables by 52% on average. A great room for improvement in the usages of local variable is expected as changing the scope of declared variable in Scratch is an expensive and tedious transformation requiring the programmer to create a new variable with the intended scope and replace each existing variable block with the newly created one manually.

5.3.2 User Study

We conducted an online user study, facilitated by Amazon's Mechanical Turk, in order to gain access to a diverse pool of novice and experienced participants. A total of 24 participants took part in the study. 7 out of 13 participants in the treatment group reported having programming experience as compared to 4 out of 11 in the control group. The participants took 30 minutes on average to complete the study (1-hour hard limit) and were compensated $3 for completing the assignment. This study investigated the impact of the availability of refactoring tools (i.e., code quality hints and automatic refactorings) on the propensity of programmers to improve the quality of their code. To that end, the participants were first primed to use custom blocks to improve program comprehensibility, and then encouraged to improve their code, amenable to the `Extract Custom Block` refactoring.

The participants first answered background questions about their programming experience and familiarity with Scratch. Then they received a short introduction to Scratch programming and custom blocks. To prime the participants to think about code quality, they were asked to rank two program versions on their comprehensibility (both performed the same an-

imation but one was a refactored version of the other). The participants were presented with a programming task that required reusing in two places an existing block sequence in the workspace. In order to understand what the block sequence did, it was expected to be run. Manually extracting a parameterless custom block from this code sequence took 5 editing steps. The participants were asked to make sure their code was easy to understand before completing the task. In the remainder of the study, the control and treatment groups diverged. Only the treatment group was exposed to `Duplicate Code` hints and `Extract Custom Block` refactoring.

RQ9. Impacts of Automated Refactoring on Novice Programmers' Willingness to Improve Code Quality: To investigate how the treatment affected the likelihood of the participants improving code quality, we instrumented our custom Scratch editor to record the following two program versions: 1) the first submission attempt, before participants were asked to make their code easy for others to understand; and 2) the final submission. To understand how the presence of refactoring tools impacted the participants' attitude toward code quality, we asked them to complete a post-study survey.

5.4 Results

In this section, we presents the results for each of the research questions posed.

5.4.1 RQ7: How applicable is each introduced refactoring?

Table 5.3 summarizes the results of evaluating refactoring applicability. In our evaluation, as long as a project contains at least one instance of a given code smell, the project is considered *afflicted* by that smell. Different smells have been found to afflict different projects in the

evaluation dataset.

Afflicting over 30% of the subject projects, duplication-related smells are the most prevalent; afflicting around 21%, `Broad Scope Variable` is the least prevalent smell. One project may contain more than one instance of the same code smell (Total Smells > Afflicted Projects). We use all detected smells to evaluate refactoring applicability.

Smell → Refactoring	Afflicted Projects	Total Smells	Refactored Smells
Duplicate Code → Extract Custom Block	181 (41%)	290	229 (79%)
Duplicate Sprite → Extract Parent Sprite	142 (32%)	193	22 (11%)
Duplicate Constant → Extract Constant	194 (43%)	453	453 (100%)
Broad Scope Var. → Reduce Var. Scope	94 (21%)	145	118 (81%)

Table 5.3: Applicability (N=448)

The applicability of the introduced refactorings varies widely. With the success rate of over 75%, `Extract Constant`, `Reduce Variable Scope`, and `Extract Custom Block` are the most applicable refactorings. `Extract Custom Block`'s precondition failures are due to the variations in duplicate fragments failing to satisfy the preconditions (60% of the variations contain global variables and 31% contain non-constant expression blocks; 9% are located at non-parameterizable input slots). As expected, `Extract Parent Sprite` is the least applicable refactoring due to its restrictive preconditions—only 11% of the detected smell instances can be refactored. The reason for failures is that `Extract Parent Sprite` cannot handle certain duplicate sprites, out of which 63% differ slightly in terms of their contained code, 33% are multi-costumes, and 4% contain scripts starting with the "`when I start as a clone`" block. Overall, we observe the introduced refactorings to be satisfactorily applicable to the highly recurring smell types.

Metrics	N	Min	p25	Med	Mean	p75	Max
Duplicate Code							
Group Size	229	2	2	3	3.05	3	20
Fragment Size	229	3	3	4	5.05	6	27
Duplicate Sprite							
Group Size	22	2	2	2.5	22.86	4.75	238
Sprite Size	22	1	1	1.0	1.64	2.00	3
Duplicate Constant							
Group Size	453	5	5	6	8.96	10	113
Literal Length	453	2	2	3	3.14	3	58
Broad Scope Variable							
Total Uses	118	0	1	1	2.73	2.00	60
External Uses	90	0	0	0	0.47	0.75	6

Table 5.4: Characteristics of Refactorable Smells

5.4.2 RQ8: How do the refactorings impact code quality?

Table 5.4 summarizes the characteristics of the detected smells that are refactorable. Table 5.5 summarizes the percentage changes for different software metrics before and after performing each refactoring. To help the reader interpret the results, the last column translates the mean deltas into percent improvements. *Group Size* refers to the number of replications of a given program element. *Fragment Size* refers to the number of statement blocks within a replicated fragment. *Sprite Size* refers to the number of scripts within a replicated sprite. *Total Uses* refers to the number of times a given variable is read in a project. *External Uses* refers to the number of times a given variable is read from outside the sprite in which it is defined. Next, we describe the results in terms of different code quality attributes.

Metrics	% Change Statistics							% Improve
	N	Min	p25	Med	Mean	p75	Max	
Extract Custom Block								
LOC	147	-29.70	-4.16	-1.65	-3.38	-0.67	0.00	+3.38%
Complex Script Dens.	52	-100.00	-4.00	-2.47	-5.77	-1.44	-0.28	+5.77%
Long Script Dens.	137	-100.00	-6.45	0.66	-4.40	1.97	42.24	+4.4%
Procedure Dens.	46	2.72	13.61	35.48	49.98	54.73	222.15	+49.98%
Extract Parent Sprite								
LOC	20	-94.15	-12.71	-0.69	-8.49	4.45	41.18	+8.49%
No. Sprites	20	-96.39	-58.48	-12.70	-31.07	-8.90	-3.85	+31.07%
Extract Constant								
No. Literals	194	-65	-14.02	-8.33	-11.53	-4.7	-0.89	+11.53%
Reduce Variable Scope								
No. Local Variable	43	2.22	7.28	20	52.42	66.67	200	+52.42%

Table 5.5: Percentage Changes of Software Metrics Before and After Refactorings

5.4.3 RQ9: Does the presence of automated refactoring support affect how willing novice programmers are to refactor their code?

RQ9.1 Engagement with improving code quality: We tested whether the participants, who chose to engage in improving code quality, depended on receiving our improvement hints and their suggested refactorings. To that end, we performed a Chi-square independence test. The relationship between these variables turned significant, (χ^2 (1, N = 24) = 8.48, P=.004), thus implying that programmers receiving hints were likely to follow them in applying the suggested refactorings.

When asked which of the program versions they found easier to understand, 25% of the participants chose the original version, while 75% of them chose the refactored one. We looked further if the participants' preference for their choices affected their engagement in improving code quality. Among the participants who chose the "refactored version" as being

easier to understand, only 12.5% in the control group ended up improving the code quality, as compared to 80% of the participants in the treatment group.

RQ9.2 Code quality perception: When asked whether their finished programs would be easy to understand for novice programmers, 85% of the participants in the treatment group agreed as compared to 91% in the control group.

RQ9.3 Improvement hints and refactoring usefulness: The treatment group was asked how useful they found the improvement hints and the associated refactorings in making their code easy for others to understand. The vast majority of the group members found our refactoring tools useful: 54% *very useful* and 38% *extremely useful*.

5.5 Discussion

Overall, the results of our experimental evaluation and user study are quite revealing. The introduced refactorings do improve the code quality metrics by removing recurring code smells from the representative projects. We have also observed that the presence of improvement hints and their associated refactorings positively influences how programmers perceive code quality and its systematic improvement.

Suggesting Quality Improvements: Although automated hints help detect quality problems, some of the detected quality problems may not need automated support due to their triviality. A better alternative can be to provide hints in the form of before and after examples. Other detected problems are highly subjective. Fixing some of these problems require significant programmer involvement (e.g., to come up with a meaningful name). Finally, some problems are simply not amenable to automated refactoring due to the complexity of formalizing the required transformation strategies. Nevertheless, some non-trivial

refactorings would likely to present a cognitive burden and may interrupt the creative flow, without seamless and effective automated support to encourage their application.

Perceived Code Quality: Judging code quality remains somewhat subjective as our results reveal. The participants regardless of their programming experience perceive code quality differently. They rate their code positively on how easy it is to be understood by other programmers new to the language, even though their finished programs exhibit a similar quality to the ones they previously ranked as being harder to understand. Receiving no suggestions, programmers may be unaware about different alternatives they have to achieve their goal. The availability of automated refactoring influences programmers to become aware of code quality and how they can improve it.

Educational Benefits: These improvement hints and their associated refactorings can provide a timely intervention to help novice programmers become aware of alternative design and implementation options that can improve code quality. For example, certain quality hints and refactorings may alleviate the low usage of procedures (a well-documented observation in a prior work [53]), thus elevating the role of procedural abstraction—a fundamental concept in CS education and professional software development—in this domain. Our evaluation results focusing on the participants without programming experience are very encouraging. Custom blocks or procedures are considered somewhat a hard concept not introduced until later in the introductory curriculum. However, most of the non-programmer participants in the treatment group were able to take advantage of our refactoring tools and perceive the hints and their suggested code improvement actions positively.

Threats to Validity: Our study had several threats to validity. Our experimental evaluation results only reflected the partial applicability and quality impact of each refactoring to the studied code smells. As mentioned, some of these refactorings were applicable in additional code smells/scenarios, but not all of them could be properly covered in one study.

Because performing a refactoring is a subjective decision, the results did not necessarily equate with the actual applicability and quality impact. Nevertheless, our results suggest the potential usefulness of providing such support for the programmers in this domain.

In our study, the participants with some programming experience, but no familiarity with Scratch represented almost a half of the participants. Although we intended to include more results from novice participants, it would be impossible for us to make use of their incomplete task results. Although, as expected, programming experience was positively correlated with completion rates, it showed no influence on our intervention. The programming task used in the study was not representative of the real world programs in this domain. However, it was reasonably complex for a half of the participants that had no programming experience.

5.6 Conclusion and Future Work

This chapter describes our effort to introduce automated refactoring to Scratch, a widely used block-based programming language. To demonstrate the practicality of our analysis and transformation infrastructure, we implement four refactorings that remove highly recurring Scratch code smells, identified in prior works. By providing their rationales, preconditions and transformation strategies, we systematically document these Scratch refactorings for use by both programmers and language designers. To assess the potential usefulness of each introduced refactoring, we experimentally evaluate the applicability and quality impact of each refactoring on a dataset of 448 projects. Our evaluation results show that the introduced refactorings are highly applicable, while their application improves code quality.

Our refactoring infrastructure helps overcome two main hindrances: programmers being unaware of code improvement opportunities and the programming burden of the improvements. Our infrastructure provides coding hints with immediately actionable suggestions to

carry out the refactorings. Our user study reveals that the presence of improvement hints and associated automatic refactorings increases the likelihood of programmers deciding to improve code quality.

Although this work focuses on Scratch, our experiences and findings can benefit designers and developers of other block-based programming environments. Future designs of block-based environments can be improved to make program analyses and transformations easily accessible, so as to facilitate the development of semantic editing support, in addition to improving code quality. Our findings serve as a starting point in determining which refactorings are likely to be useful and worthwhile to programmers in this domain. We plan to investigate further how novice programmers interact with the refactoring tools as part of the overall programming process. The following research questions arise: *How the presence of refactoring tools affects how novice and end-user programmers code? How effectively this presence raises the code quality awareness among programmers?* The answers could inform the research community how much providing refactoring support raises the importance of code quality in the minds of programmers in this increasingly important domain.

Chapter 6

Introducing Code Quality Concepts Alongside the Fundamentals of Programming

6.1 Introduction

The CS Education research literature has established the importance of teaching software quality as part of the CS curriculum [8, 43, 44, 89]. However, it remains subject to considerable debate whether the topic of software quality is appropriate for introductory learners. Some computing educators argue that promoting disciplined programming practices is incongruent with the guiding principles of Constructivism, the educational philosophy centered around unconstrained experiential learning that guides many of today's introductory computing curricula [30, 61, 82, 103].

Although teaching refactoring can demonstrate the importance of software quality and its improvement practices to novice programmers, our understanding about the approaches and their effectiveness have been very limited. It remains unclear how teaching this advanced topic impacts novice programmers' learning experience and outcomes, as well as what strategies can be used to guide the teaching of refactoring and the related code quality concepts.

With the availability of automated refactoring, it is also unclear how this automated tool plays a role in teaching novice programmers this systematic code improvement. Although in professional software development, refactoring is highly automated, should beginner programmers learn this technique manually or using an automated tool? From the pedagogical standpoint, both approaches have their respective advantages and drawbacks. When taught how to refactor by hand, learners can see clearly how to identify code quality problems and how to remove them by modifying the code. In contrast, automated refactoring tools remove the tedium of analyzing and transforming code by hand and help ensure that the refactored code retains its original semantics.

To address this knowledge gap, we conducted an experimental case study[1] of teaching the Extract Custom Block refactoring[2], which removes code duplication. We focus on refactoring code duplication or *Duplicate Code*, a highly common code smell in Scratch not only because of its prevalence but also its important implications in introductory computing education. Similarly to text-based software, the most prevalent recurring quality problem is *code duplication*, hard-to-maintain repeated code segments that originate from the ubiquitous copy-and-paste approach to reusing existing code [98].

Rampant code duplication problems in this domain suggest that learners never become comfortable with procedural abstraction as a means of eliminating code duplication and lowering code complexity. The research literature identifies code duplication in Scratch as one of the most salient problems that stands on the way of introductory learners mastering advanced programming concepts and techniques. For example, as their programming experience and proficiency kept growing, learners were observed to continue writing code rife with duplication [78, 95]. Furthermore, recent research findings show a disappointingly low usage of procedural abstraction in a large codebase of Scratch projects[1, 45], with the same

[1] VT IRB approval was obtained for the human studies described in this study.
[2] In text-based languages, this refactoring is known as *Extract Procedure*.

issue also affecting other block-based languages such as AppInventor [53]. One explanation of this low usage is that the practice of abstraction and modularization might not come naturally and should be taught explicitly [54]. Therefore, addressing code duplication and other recurring quality problems may need educational interventions that enhance introductory computing education with practical software engineering principles and practices.

Specifically, the goal of our work is to answer the following questions:

RQ10: Do novice programmers perceive enjoyment and difficulty differently when learning how to use procedural abstraction as compared to learning how to refactor code duplication?

RQ11: Does introducing novice programmers to the Extract Custom Block refactoring help them understand the concept of custom blocks?

RQ12: Does the preference for manual or automated refactoring methods change and why, as novice programmers progress from learning refactoring to intending to refactor in the future?

To answer these questions, we conducted an online study with 24 first-time Scratch programmers. The participants were divided into two equally sized groups. Each participant was given a short tutorial on the basics of Scratch programming (Part 1) and then the Extract Custom Block refactoring (Part 2). In Part 2, the first group first learned how to perform the Extract Custom Block refactoring by hand and then how to do so by using an automated refactoring tool. The second group learned the same skills but in the reverse order. Finally, all participants were surveyed to compare and contrast the prevailing attitudes and experiences across the two groups. To help educators design similar educational interventions, we identify the tutorial's key principles and explain how they manifest themselves. We attribute the success of our tutorial to having made careful design decisions based on systematically researching the problem domain. Specifically, we followed a bottom-up experimental design approach, eventually creating a tutorial that provides a real-world context

for the introduced technical subject, while keeping the learners engaged and motivated. The retrospective insights gleaned from the tutorial and its instructional strategies can serve as a helpful guide that informs future curricular interventions for novice programmers.

The rest of the chapter is structured as follows. Section 6.2 presents the aforementioned tutorial that teaches code duplication refactoring. Section 6.3 describes our methodologies for answering the research questions posed. Section 6.4 answers each of the research questions based on the results of our controlled study. Section 6.6 discusses the significance of the results. Section 6.5 presents the underlying principles identified to guide the design of the tutorial. Section 6.7 presents concluding remarks and future work directions.

6.2 Tutorial: Code Duplication Refactoring

In this section, we begin by describing our tutorial, the platform we used in the controlled study that explores the impact of automated refactoring support on novice programmers learning how to refactor code duplication.

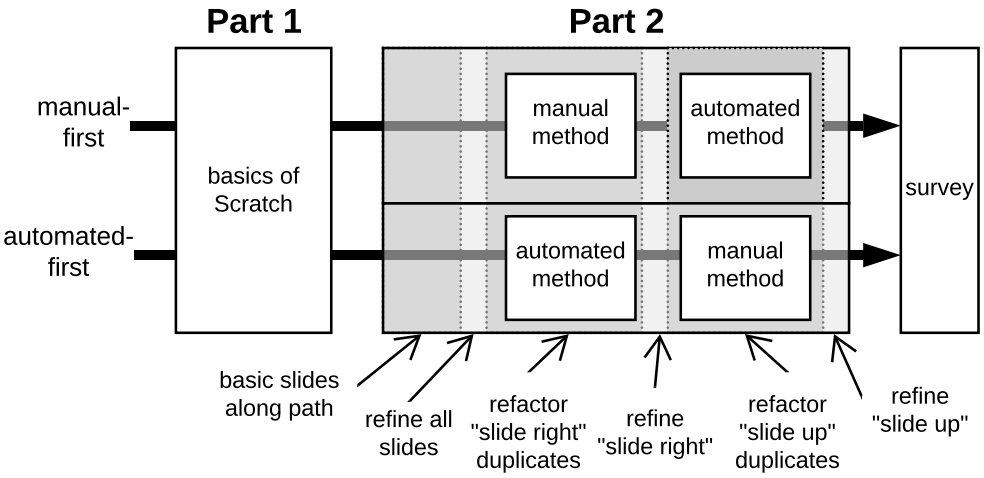


Figure 6.1: Tutorial's overall flow

6.2.1 Content Structure

The tutorial has two parts: Part 1 (Basics of Scratch) and Part 2 (Code Duplication Refactoring). Figure 6.1 outlines the tutorial's overall flow.

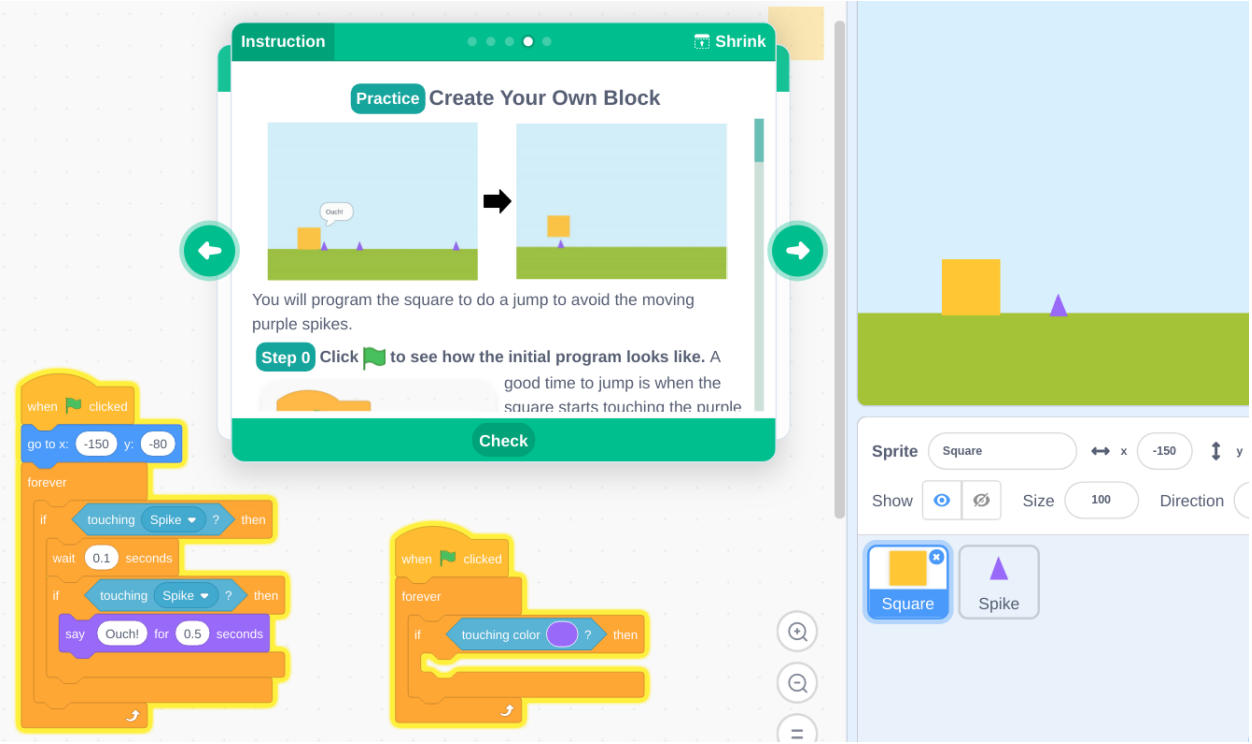


Figure 6.2: Screenshot of tutorial's Part 1

Part 1: This part presents a short hands-on practice that covers the necessary basics of Scratch programming including custom blocks, a procedural abstraction construct in Scratch. It first prepares learners to become familiarized with the basic Scratch programming interface for composing programs. It then introduces custom blocks, how to create and use them to complete the overall objective for Part 1: making a character jump to avoid touching the moving obstacles (Figure 6.2). The tutorial introduces custom blocks by following the recommendations from the Creative Computing Curriculum [3], a Scratch instructor's guide developed by the Harvard Graduate School of Education. Learners work through the following sequence: 1) think about what custom block they need to create, 2) create a custom

block, 3) define the block's body, and 4) use the created block in their program.

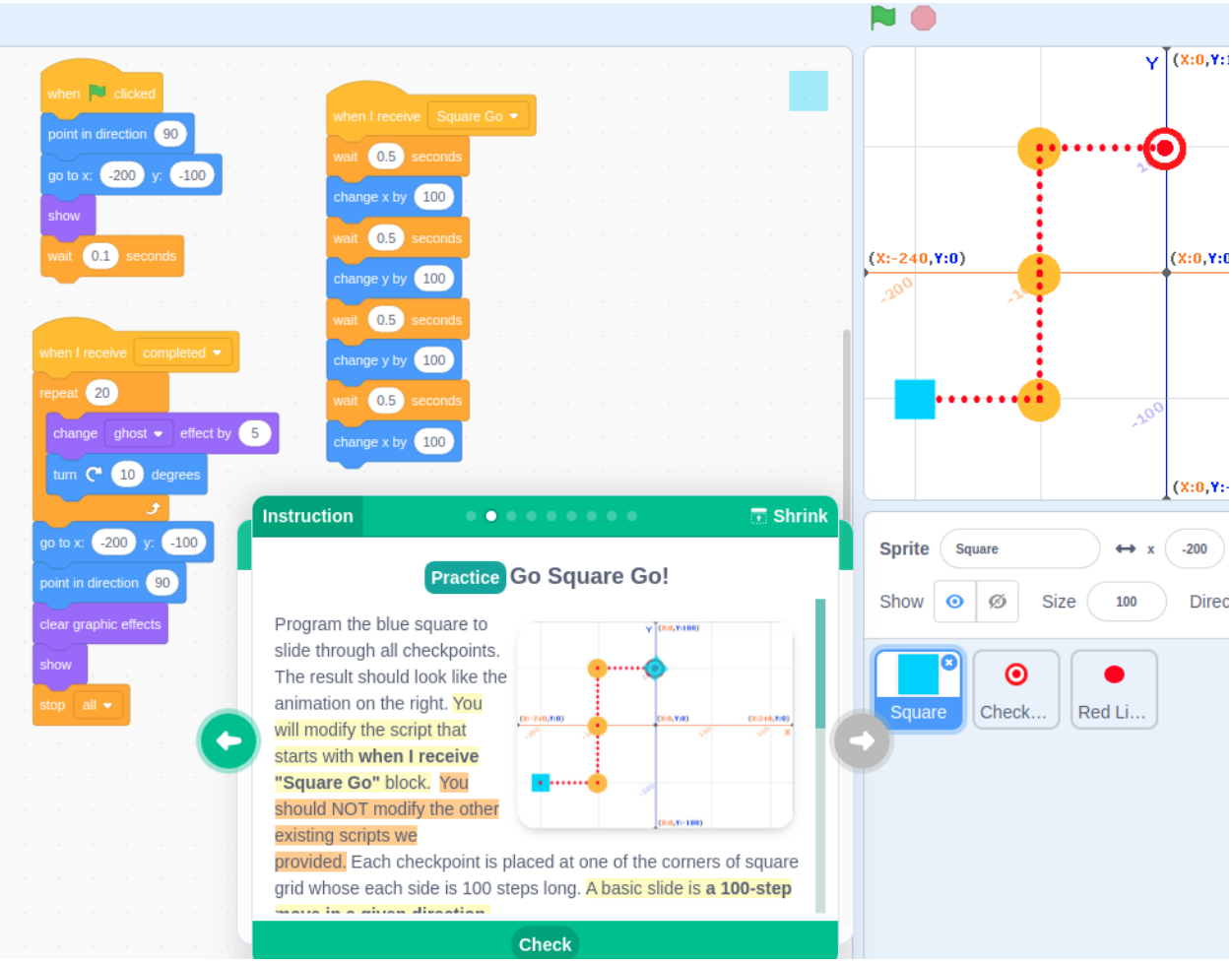


Figure 6.3: Screenshot of tutorial's Part 2; The initial project skeleton was scaffolded to help learners visualize the program behavior when working on *basic-slides* task.

Part 2: This part guides learners to complete its objective: make a character move within a grid of squares (see Figure 6.3). To motivate code duplication refactoring, it situates the learners in a scenario that demands frequent code modifications, typical for the iterative software development process [73]. The tutorial guides learners how to carry out the Extract Custom Block refactoring both manually and automatically.

The participants in both groups (manual-first and automated-first) learned how to refactor with both manual and automated methods but in different order.

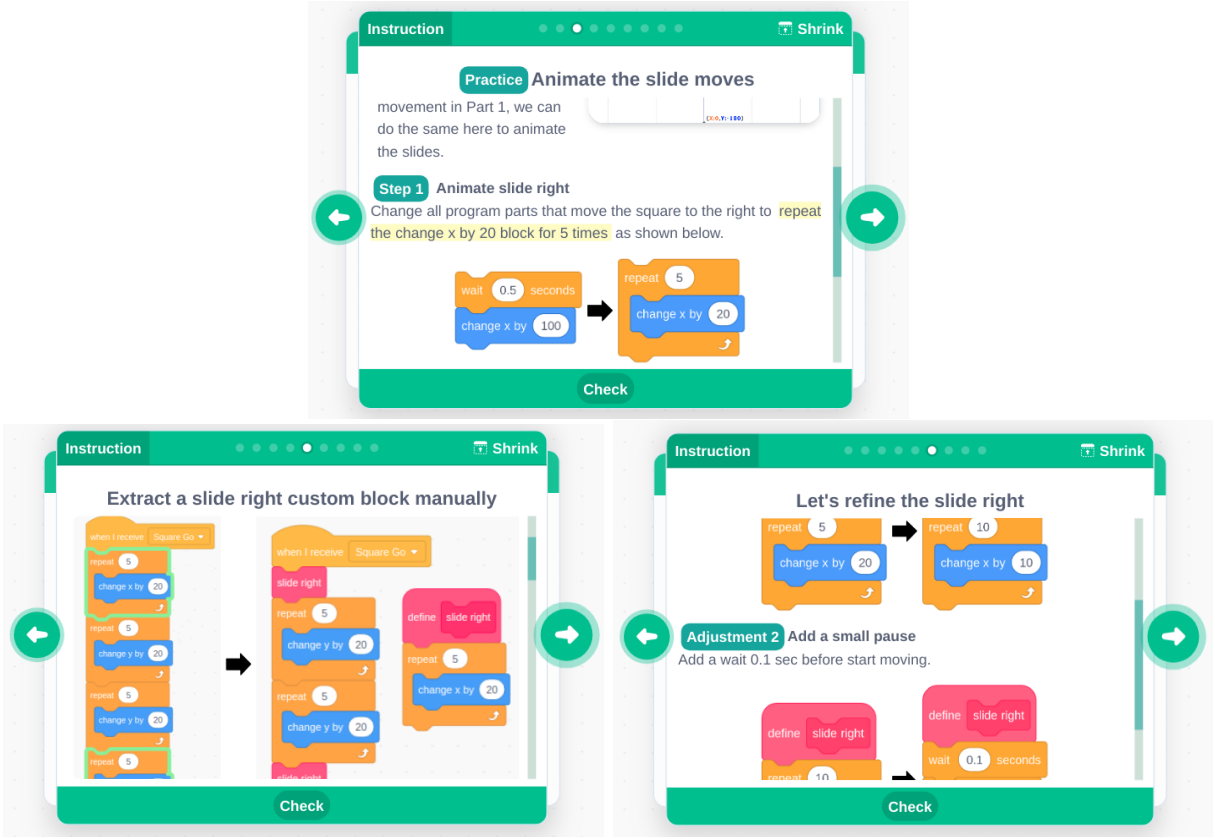


Figure 6.4: Screenshots of instruction cards for different tasks: refine all slides (top), refactor *slide right* duplicates (bottom left), and refine *slide right* (bottom right)

Learners first complete the missing program part to make a character move within a grid of squares (see Figure 6.3). The instructions specify "slide", as a special movement of 100 units that moves the character from one square to the next, alluding to the use of procedural abstraction as the recommended implementation of "slide". Given the "slide right" code as an example, learners complete the missing part to make the character slide through with the moves of "right", "up", "up", and "right". The finished code contains two duplicate parts.

At this point, the tutorial presents the first refinement task (*refine-all*) that requires changing all slide movements consistently, thus demonstrating how code duplication can make programs hard to understand and modify (see Figure 6.4, top screenshot). Then, it introduces learners how to carry out the refactoring manually (see Figure 6.4, bottom left screenshot)

and with automated tools. After being introduced to each method, learners are instructed to refine the recently refactored sliding code (see Figure 6.4, bottom right screenshot), as a way to demonstrate the beneficial impact of custom blocks on code quality. They learn each of the refactoring methods by following the steps as laid out in Section 6.2.2.

Please note the use of *change x or y* blocks nested in *repeat* blocks (e.g., repeating the command *change x by 20 units* for 5 times to move the character 100 units horizontally) is needed to create a smooth animated movement as compared to an abrupt movement with a single block approach (i.e. *change x by 100 units*). Also, the use of 100-unit square grid is an intentional design choice that demonstrates how the concept of procedural abstraction works. Command blocks in the "Motion" category are high-level abstractions for manipulating 2-D graphical objects. To make it possible for learners to intuitively learn about procedural abstraction, we introduce another level of abstraction: the square grid setting that exposes the *change x or y* blocks as low-level commands.

6.2.2 Manual and Automated Refactoring

We outline the main steps used to teach the refactoring by hand and with an automated tool as follows:

Refactoring by Hand

1. Identify the duplicate code parts
2. Create a custom block (a descriptively named procedure with an empty body)
3. Define the custom block (the procedure's body): Make a copy of one of the duplicate parts to use as the block's definition
4. Replace the duplicate parts with the calls to the newly created custom block

Refactoring using an Automated Tool

1. Identify the duplicate parts: Hint icons point to the detected code duplicates. Hovering the mouse over a hint highlights the corresponding duplicate parts in the workspace.
2. Extract Custom Block from a selected duplication: Click the hint icon to see the refactoring menu, and then click "Extract"
3. Rename the extracted custom block: Replace the custom block's placeholder name ("DoSomething") with a descriptive name that specifies what the block does.

Figure 6.5 shows the user interface of the refactoring tool.

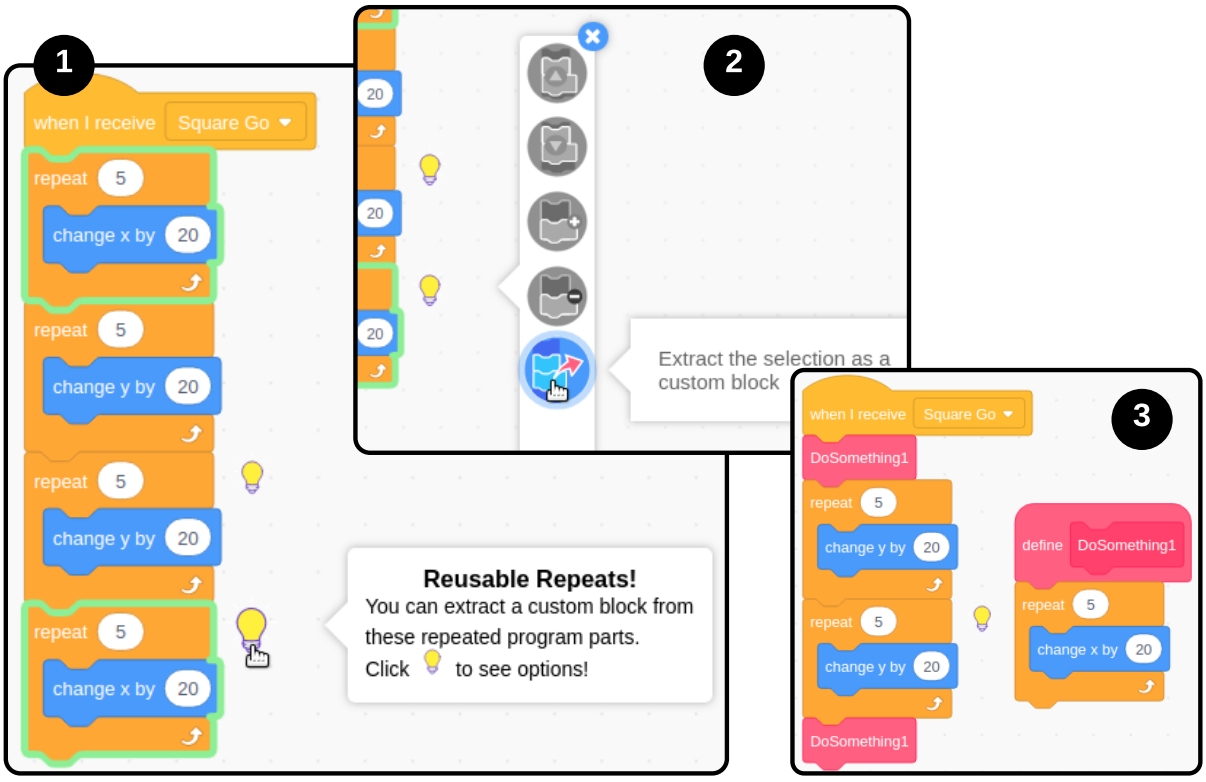


Figure 6.5: A screenshot of Extract Custom Block Refactoring tool
(1 : identify, 2 : extract, 3 : transformed code)

6.3 Methodology

We follow a mixed-method design. Specifically, we conducted a controlled study that introduces novice programmers how to refactor code duplication. This study explores the impact of the availability of automated refactoring support on learning the subject matter. The study was conducted online via Amazon Mechanical Turk, with the participants admitted on a rolling basis until reaching the target sample size of 24, spanning for three weeks in December 2019. The participants were divided into two equally sized groups (manual-first and automated-first). Only the participants with no prior programming experience (i.e., those who selected the lowest of the six levels: "I have never written any computer code.") were admitted to take the tutorial and a post-tutorial survey.

To understand how effective our tutorial was in teaching the participants how to refactor, after being introduced to both the manual and automated methods, each participant was asked to indicate their level of agreement with a series of statements below on a five point Likert scale (Strongly disagree, Somewhat disagree, Neither agree nor disagree, Somewhat agree and Strongly agree). We also recorded the time taken by each participant to complete each tutorial task, as an additional quantitative measure of their performance.

To answer statements A1-4, the participants used a Likert response form for both the manual and automated methods side-by-side. We refer to the A1-4 statements and their responses by subscripting the method's name to the statement labels (e.g., $A1_{manual}$ corresponds to "I found learning how to Extract Custom Block with the manual method enjoyable.").

To further understand the learning experiences of the study participants, we asked them an additional open-ended question whether they found that learning the first method to refactor duplication helped them to learn the second method, and briefly explain their answer.

To compare two independent samples, we used the Wilcoxon rank sum test. The test

(A) I found learning how to extract a custom block ...

1. enjoyable
2. easy
3. hard at first but easier later
4. helpful in understanding custom blocks

(B) Overall, I found Part 2 helped me understand why ...

1. duplication can make code hard to understand
2. duplication can make code hard to modify
3. Extract Custom Block can make code easy to understand
4. Extract Custom Block can make code easy to modify

statistics for the Wilcoxon rank-sum test is denoted with W. We considered the p-values less than .05 as statistically significant.

6.4 Results

On average, it took 50 minutes per participant to complete the entire study (tutorial and survey). We provided minimal help to some participants via a live support feature of our online study website. Most of the help was given for the tutorial tasks prior to the refactoring tasks, in which participants were asked to fill in the missing parts to complete Part 2's objective (*basic-slides*). Next, we present the study results and highlight specific observations that pertain to each research question.

Figure 6.6 shows the distributions of time data of 12 participants in the manual-first group for each tutorial's task. The participants in both groups took about the same time to complete each task except the refactoring tasks. We further discuss this difference in more details when addressing *RQ11* about the content's helpfulness.

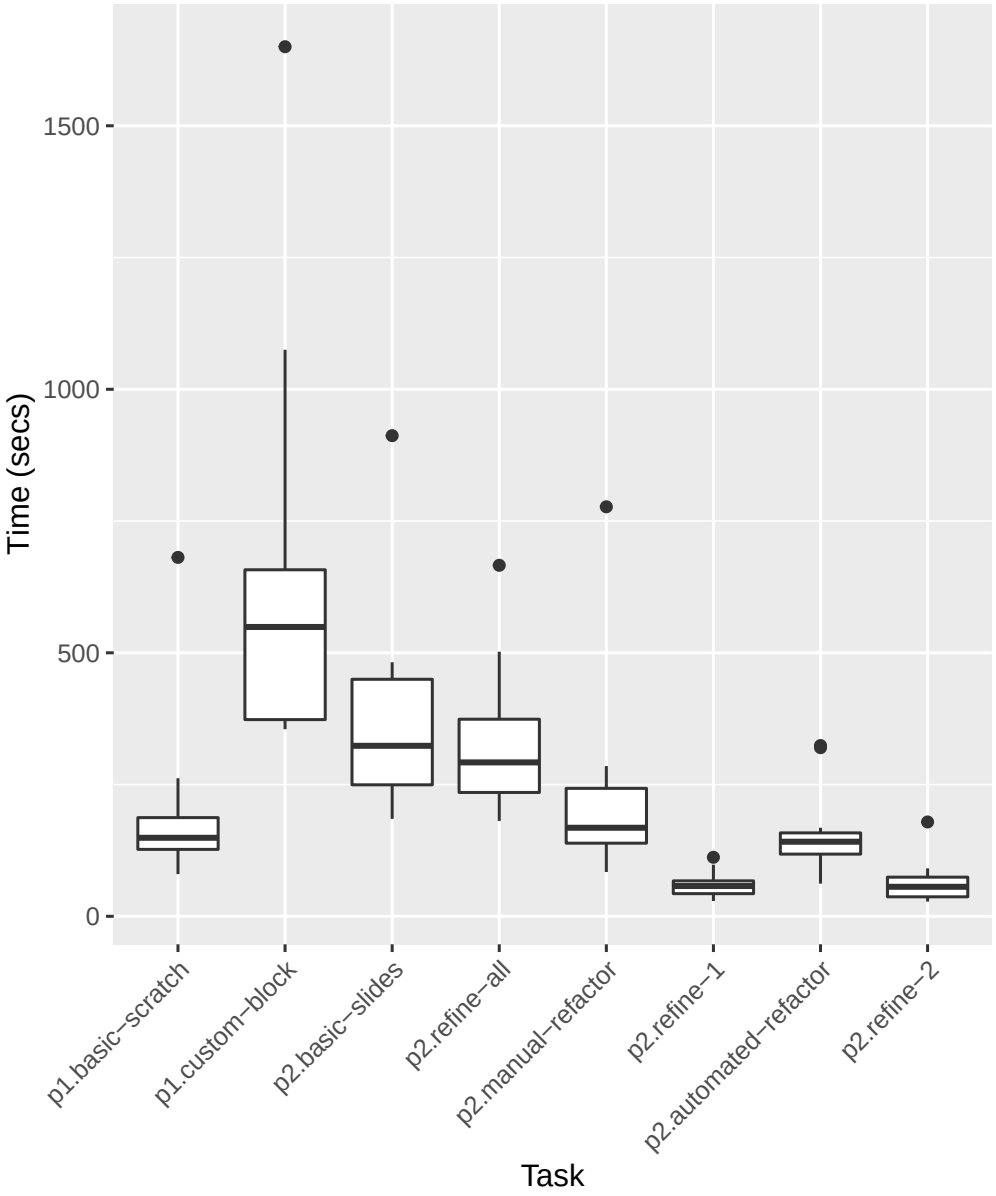


Figure 6.6: Distributions of time spent on each tutorial's task ordered chronologically

Several interesting observations can be made about how long the participants spent going through each part of the tutorial. The majority of the participants spent most of their time learning how to create and use a custom block in Part 1 (*p1.custom-block*). Novice programmers clearly went through a learning curve to grasp and use this procedural abstraction construct. Nevertheless, this time investment helped them when they learned how to carry

out a manual refactoring in Part 2 (*p2.manual-refactor*). The second longest task the majority of the participants spent on is completing the puzzle-like task, filling in the missing parts that slide the character through checkpoints (*p2.basic-slides*). They also spent as much time as in the previous task when refining their duplicate slide functionalities, possibly long enough that, upon the tutorial's guided reflection, they realized by themselves that it was inefficient to modify the duplicate code to refine the slide functionalities. Having eliminated the duplicate code, the participants took less time to refine the sliding functionalities in the two tasks (*p2.refine-1* and *p2.refine-2*) combined.

Part 1's overall learning experience: Participants expressed positive learning experiences about Part 1. They found this introductory part of the tutorial fun, interesting, and educational. Some samples of their responses are as follows:

"I had a fun time learning what I could do with the blocks"

"That was really interesting, I've always wanted to learn programming but was a bit anxious about how difficult it might be and I was worried that I might not like it but this really helped out a lot."

"I thought this was a lot of fun. After I got the instructions down it was super easy. It felt good to complete something and make a block jump!"

"The instructions were clear and the exercises were helpful. Overall, it was fun and educational."

"I think this style of programming is very interesting. I hope there's more in the rest of this study."

"I liked the visuals it helped me a lot. I also really liked how I could check each change and view the results."

Part 2's overll learning experience: Several participants reported positive learning ex-

periences. They found the learning enjoyable and interesting. One participant felt "I would have enjoyed doing more'' while another said "It was fun and the program was intuitive. I could see it being a lot of fun to work with and to learn code on.'' Feeling appreciative of the learning experience, one participant said, "Thank you for introducing me to something new!''

However, several participants found Part 2 challenging. One participant said, "I thought it went pretty well. I was scared going into it, but proud that I was able to finish it with minimal struggle.'' One participant said, "It was a little bit more challenging but I really enjoyed it.'' while another said "It was still a bit intense for a beginner but after a deeper read, I could follow along. The pictures of the blocks helped the tutorial immensely.''

Other responses suggest improvements to the tutorial. One participant said, "I liked how simple the program made the overall concept – but the instructions were a bit convoluted for a beginner.'' Few other suggestions are mostly about improving the usability and accessibility of the tutorial instructions when viewing the tutorial on a small screen.

6.4.1 RQ10: Do novice programmers perceive enjoyment and difficulty differently when learning how to use procedural abstraction as compared to learning how to refactor code duplication?

Figure 6.7 visualizes the raw results of participants' level of agreement with statements A1-4, with the responses from the manual-first and automated-first groups: the manual method at the top, and the automated one at the bottom.

Enjoyment: A majority of participants in both groups either somewhat agreed or strongly

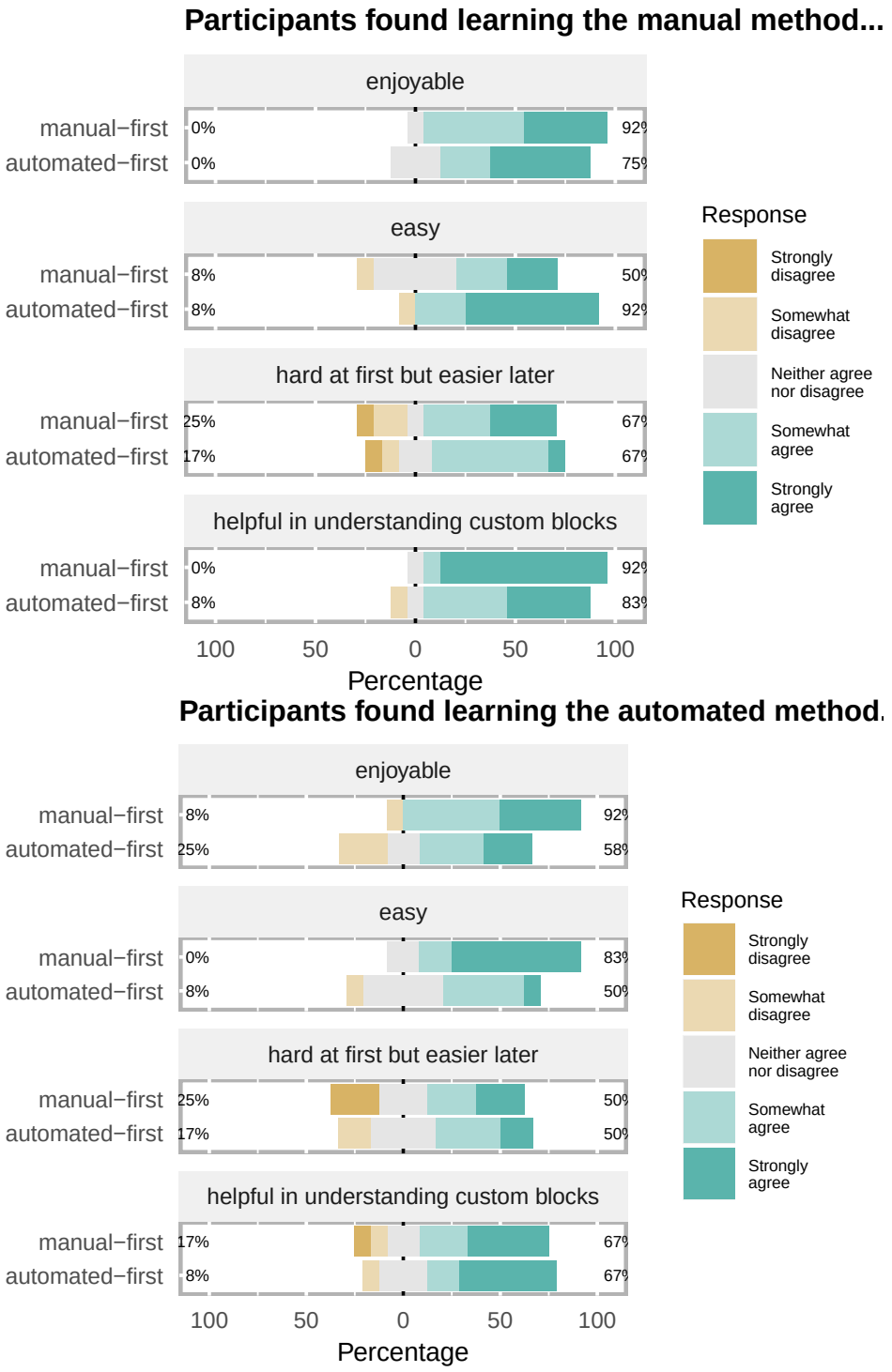


Figure 6.7: Participants' experience of learning to refactor with manual method (top) and automated method (bottom)

agreed that they found learning manual refactoring enjoyable. When asked if they found learning automated refactoring enjoyable, 92% in the manual-first group agreed, as compared to only 58% in the automated-first group.

Difficulty Level: A majority of participants agreed that learning the second method was easy, while they agreed less strongly that learning the first method was easy. For example, when comparing within the same group, 92% of the automated-first group either somewhat agreed or strongly agreed with the statement $A2_{manual}$ that they found learning manual refactoring easy, while only half of them reported feeling the same way about learning automated refactoring as their first method. When comparing between the group responses to the $A2_{manual}$ statement, only 50% of the manual-first group agreed that they found learning manual refactoring easy. A Wilcoxon rank sum test indicated that the level of agreement to the statement $A2_{manual}$ was higher for the participants in the automated-first group than those in the manual-first group ($W = 107$, $p = 0.034$). Around 67% of the participants in both groups either somewhat agreed or strongly agreed that they found learning manual refactoring method "hard at first but easier later", although the participants in the automated-first group agreed less strongly with that statement.

6.4.2 RQ11: Does introducing novice programmers to the Extract Custom Block refactoring help them understand the concept of custom blocks?

Content's Helpfulness:

When it comes to helping understand custom blocks, the majority of participants in both groups agreed that they found learning Extract Custom Block refactoring were helpful to them in understanding the concept of custom block. However, they found the *manual method*

to be more helpful than the *automated method*.

The time the participants spent in following the instructions of each refactoring method reflects how helpful is the first learning method. The boxplots in Figure 6.8 show how long participants in the two groups took to complete the tutorial tasks that guide them to refactor code duplication. Having learned to refactor with the *automated method* first, the participants in the automated-first group were only slightly faster to carry out the refactoring manually. A Wilcoxon rank sum test indicated that the performance difference between the two groups is not statistically significant ($W = 85.5$, $p = 0.452$).

In contrast, having learned how to refactor manually, the participants in the manual-first group learned to refactor by using an automated tool significantly faster. A Wilcoxon rank sum test indicated that the learning performance was higher for the participants in the manual-first group (Mdn = 141) than those in the automated-first group (Mdn = 238), ($W = 35$, $p = 0.035$).

In both plots, the outlier data points were contributed by the same early stage participant, who may have taken a break between different parts of the tutorial. The entire study's time limit was initially set to 2 hours and then to 1.5 hours, so as to encourage participants to complete the study uninterrupted.

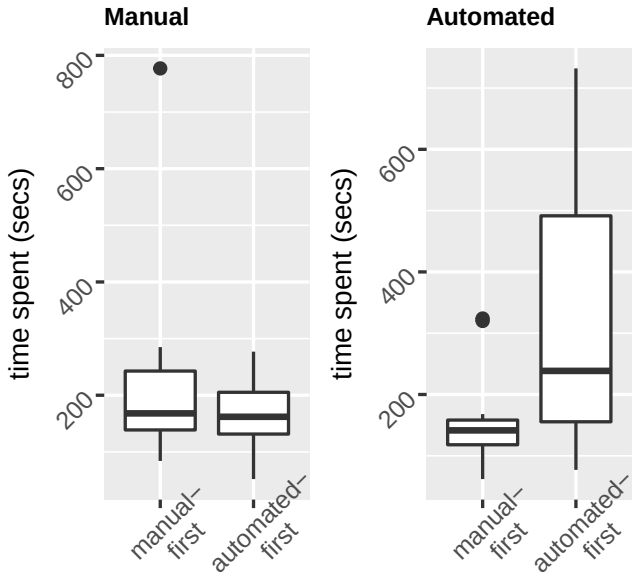


Figure 6.8: Comparing time to complete learning tasks: manual method (left) and automated method (right)

6.4.3 RQ12: Does the preference for manual or automated refactoring methods change and why, as novice programmers progress from learning refactoring to intending to refactor in the future?

The majority of participants in both groups prefer the *manual method* as the first learning method but overwhelmingly preferred *automated method* to use in the future. Around 67% members of the automated-first group reported liking being introduced to refactoring with the automated method first; many of them prefer the method because it helps with the conceptual understanding of how refactoring works. One participant pointed out that it "showed the basic process and the end result...so I started out with a basic idea of what was to be done in the manual method and how it should turn out in the end.", while another suggested that it "broke down the concept of what I was doing before doing it manually". One participant claimed that it "was not too helpful for the given example but it could be

useful in a large complicated project." Around 33% of the automated-first group wished they had learned the *manual method* first. Many of them felt that the *automated method* was simpler as compared to the *manual method* and should be learned after the *manual method*. One participant preferred to learn the *manual method* first "... so I understand the mechanics of what I'm doing, then show me the tool that makes it easier for me.", while another said it "seems to follow from Part 1 (Custom Block) better...". One participant claimed that "it would have been easy either way."

All participants in the manual-first group reported liking being introduced to the manual method first. Many of them liked the detailed step-by-step manual process as a learning aid that helped them understand what the automated tool did behind the scenes; one participant claimed that "it helped me fully understand what the tool actually does," while another one added "...in the event that the tool doesn't do what I wanted I can go then do it manually.'' One participant claimed that "I would have been confused about what the tool was doing for me if I hadn't been familiar with what was going on first.''

When asked about which method they would prefer if they were to continue refactoring code duplication, the majority of the participants in both groups preferred automated refactoring (9 out of 12 in the automated-first group and 10 out of 12 in the manual-first group). Most participants prefer the automated method because it is faster and simpler, given that they have a conceptual understanding of how refactoring works. One participant said "...using the tool is much faster. Once I understand the how and the why of the process by doing it manually, I don't need to keep doing it manually if the tool can do the same thing, but do it faster." The few participants preferred the manual method, due to the learning benefits of carrying out the refactoring by hand and the tighter control over the transformation process, with the ability to place the new code as desired and to avoid unexpected mistakes that automated code transformation might introduce.

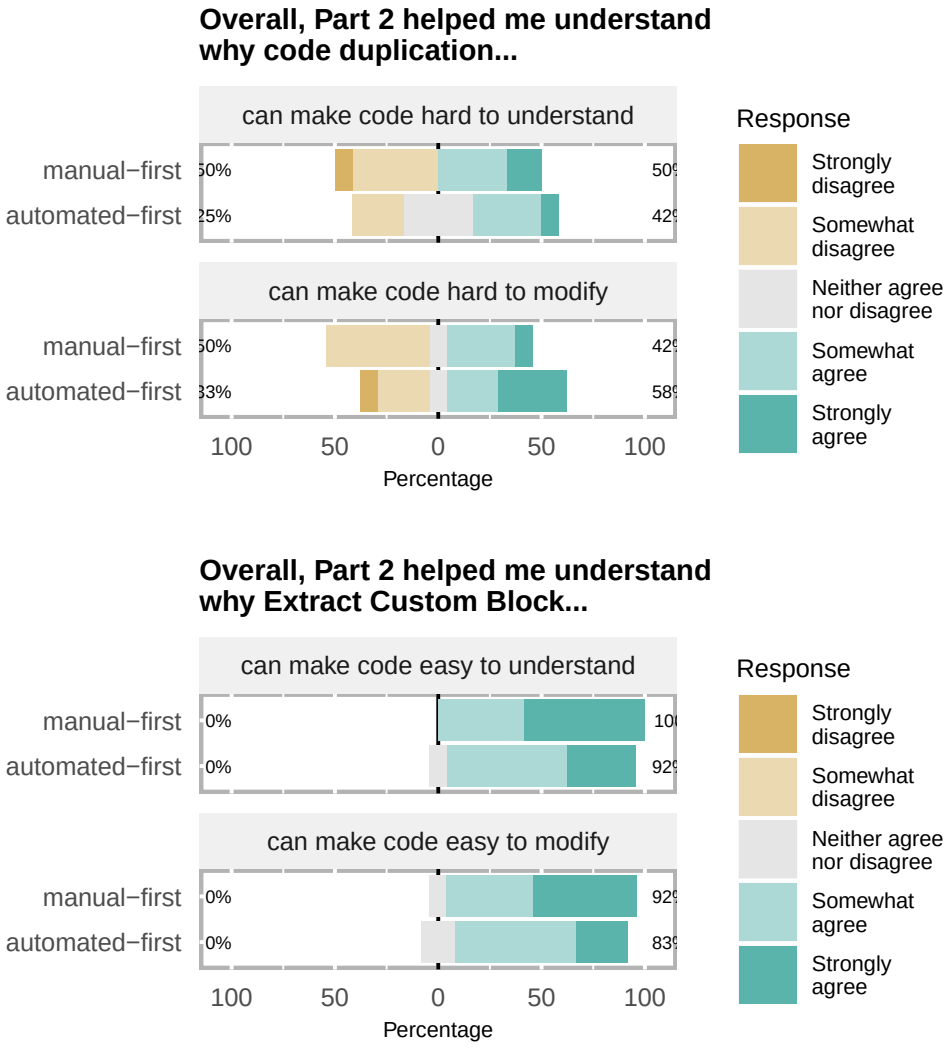


Figure 6.9: Participants' level of agreement to B1 statement (top) and B2 statement (bottom) by groups

6.4.4 RQ13: Does introducing code duplication and its refactoring to novice programmers help them understand the concepts of software understandability and modifiability?

Figure 6.9 visualizes the raw results of participants' level of agreement to the statements B1-4 about the tutorial's impact on their understanding about code quality concepts. Participants in both groups agreed less strongly that the tutorial helped them understand why code

duplication can make code hard to understand (B1), and similarly to the statement that the tutorial helped them understand why code duplication can make code hard to modify (B2). Conversely, the majority of the participants in both groups agreed that the tutorial helped them understand why Extract Custom Block refactoring can make code easy to understand (B3) and that Extract Custom Block refactoring can make code easy to modify (B4).

6.5 Design Strategies

Being the central component of our study, the aforementioned tutorial proved remarkably effective, so we take a closer look at its underlying design principles. To help with designing similar educational interventions, we identify the tutorial's key principles and explain how they manifest themselves. We attribute the success of our tutorial to having made careful design decisions based on systematically researching the problem domain. Specifically, we followed a bottom-up experimental design approach, eventually creating a tutorial that provides a real-world context for the introduced technical subject, while keeping the learners engaged and motivated. The retrospective insights gleaned from the design of this tutorial can serve as a helpful guide that informs future curricular interventions for novice programmers.

This tutorial is an educational intervention with two key learning objectives. Upon completing the tutorial, we expect learners to be able to 1) learn how to refactor code duplication in order to systematically improve code quality and 2) grasp code quality concepts, developing an appreciation for the importance of code quality and its systematic improvement practices. In retrospect, our tutorial supports these learning objectives by following five design principles in its development. We present each principle and explain how it manifests itself in the tutorial next.

1. **Simple but representative examples:** The tutorial explains how to refactor simple identical duplicate code sequences that slide a character object. The goal is to replace these duplications with a single, reusable custom block. This refactoring use case is quite rudimentary in the sense of replicating the simplest possible functionalities, introduced just before. Using such rudimentary use case can be viewed as *a worked example*, a simple example that demonstrates the mechanics of how to perform a new task, an instructional strategy based on cognitive load theory, applied in many areas including computer science [86]. This rudimentary example makes learning systematic code improvement practice accessible to novice programmers.

   Examples should match the programming knowledge already possessed by the target audience. In our case, the example code contains basic programming concepts as well as the programming constructs of *sequences* and *loops*, identified as appropriate for beginner learners in introductory CS education research [11, 115].

2. **Engaging the learner:** The design and implementation of our tutorial is tailored to encourage learners' engagement with the content materials, getting them in the loop of learning quality improvement.

   For learners to become fully engaged with the task at hand, they need to understand the task which includes the computer code accompanied the task. As opposed to displaying the example computer code up front, the tutorial asked the learners to construct it for themselves by completing a puzzle-like programming task (*basic-slides*) and refining them (*refine-all*). This puzzle-like programming task helps engage learners in a style similar to the ones commonly used in introducing learning activities to novice programmers (e.g., Code.org online learning platform). More importantly, having written the necessary code by themselves, learners are expected to understand how the code works. It is our goal to help learners become confident with the subject code

and build a mental model of how the refactored program changes structurally while retaining its behavior.

The tutorial features a simple code check that gives on-demand feedback whether a learner correctly follows the provided instructions for a given task. According to Bandura's cognitive theory of self-efficacy [5], allowing learners to check their own progress at a designated level of proficiency impacts motivation positively. Constructivist theory [79, 93] also suggests that the exchange of timely feedback can encourage learners to improve the quality of their work. Aligning with these guiding learning theory, our tutorial kept learners engaged, so as to increase their success rate of learning advanced code improvement technique.

3. **Providing a relevant real-world context:** We model our tutorial instructions as progressive refinements, which often reflect an iterative process of software developers coming up with a simple solution and iteratively improving it [105]. Situating learners in a real development context provides a unique opportunity to convey to them the importance of code quality and its improvement. It can be difficult to find simple examples that are also realistic. For code duplication, one could create several duplications but they might not be convincing when looking at the entire program. If the duplicate segments of program instructions appear artificial, it would be quite hard to convincingly select the duplicate functionality to extract and also to come up with a descriptive name for the extracted procedure.

4. **Encouraging reflective thinking:** The tutorial content includes small breaks for learners to reflect on the quality of their code they just worked on before and after they refactor their code. For example, the *refine-all* task requires learners to make consistent changes in duplicate parts that make the character slide. Having modified the duplicated parts to refined the "slide'' functionalities, the participants were pre-

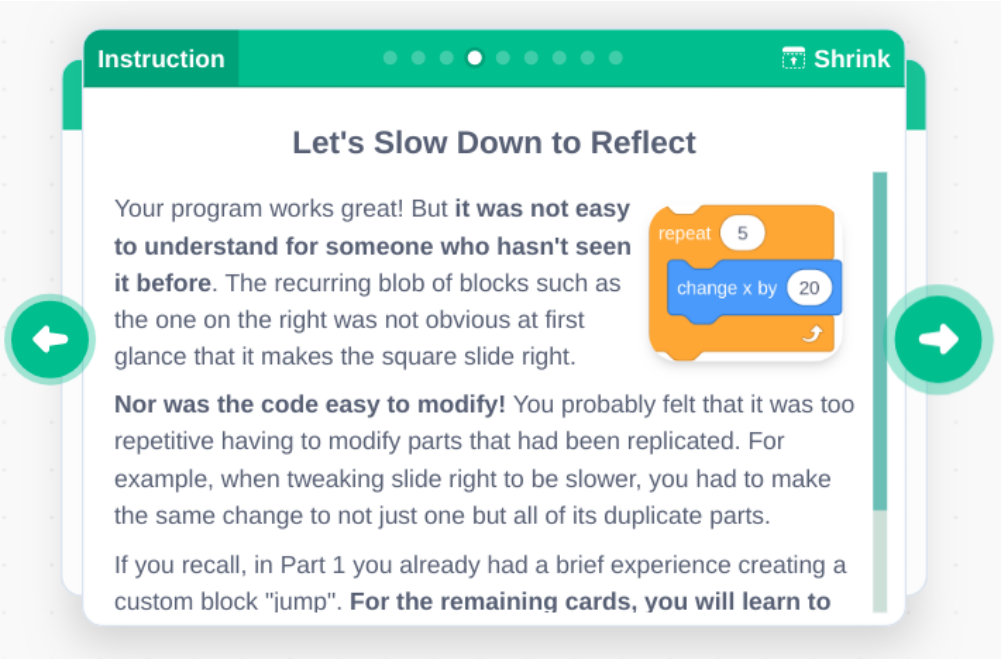


Figure 6.10: Tutorial's Part 2: A short note to help learners reflect on code quality

sented with a short note that guided them to reflect on their experience of modifying and understanding duplicate code (see Figure 6.10) before learning refactoring.

5. **Scaffolding the learning:** The tutorial loads up an initial project skeleton, scaffolded to enhance the introductory learning experience. For example, the project displays grid squares, visual checkpoints and their behavior when touching the sliding character object, and path tracing. All these additional aids provide helpful visual output that demonstrates how a program's behavior changes in response to source code modifications. Most importantly, by observing that the program's output remains the same as its structure changes, learners receive a powerful yet easy-to-understand demonstration of two non-trivial software engineering concepts: (1) the same functionality can be implemented differently; (2) refactoring is a behavior-preserving transformation.

6.6 Discussion

The results of our case study have revealed several trends that can be applied to improve various aspects of introductory computing pedagogy. We discuss these trends next.

Learner engagement: Although carrying out the Extract Custom Block refactoring can be tedious, the freshness of learning something new and the rudimentary examples used may help offset the tedium of the manual refactoring's repetitive steps. The exposure to automated refactoring seems to have helped many participants by significantly reducing their perception that manual refactoring was difficult. Nevertheless, automated refactoring can have a potential unexpected negative consequence: making learners less engaged in learning how to refactor. When simple GUI actions replace the necessity of understanding how the underlying source code should be restructured, novice programmers may miss on some important learning opportunities, when it comes to improving their skills required to understand and evolve code.

All study participants claim that both methods helped them understand the concept of custom blocks. The manual method was perceived as more helpful, possibly because custom blocks were introduced right before, thus naturally connecting these two topics. This result reaffirms that indeed there are great potential opportunities for applying solid software engineering practices (refactoring in this case) to reinforce core programming concepts as suggested by prior studies [34, 97].

Roles of automated refactoring: The information on task's completion time suggests that prior exposure to automated refactoring did not help participants significantly in learning how to refactor by hand with the examples used in this case study. Although the role of automated refactoring in the learning process may not be significant, automated support can be critical in ensuring that budding programmers continue refactoring their code, as the

overwhelming preference for automated refactoring indicates. The results for both RQ10 and RQ11 show that introducing novice programmers to refactoring effectively should not require advanced refactoring support, but it suffices to use rudimentary examples with a hands-on activity designed to help learners see for themselves the benefits of refactoring.

Code quality's learning outcomes: The results of evaluating instructional effectiveness (RQ3) suggest that the tutorial's teaching strategies are quite effective in demonstrating how the Extract Custom Block refactoring can make code easier to understand and modify, while it is less effective in conveying that code duplication hinders program comprehension and modification. Although using a non-trivial example codebase could have been more convincing, the resulting complexity and size would be inappropriate for beginners. Our results suggest that it might be more effective to demonstrate the benefits of removing duplication rather than trying to convince learners that a specific code segment has a quality problem. Indeed, as prior studies with experienced CS students [9] and beginner Scratch programmers [97] indicate, quality judgment is subjective regardless of the experience level.

Although certain aspects of code quality may be hard to convey with a small example, we have succeeded in showing learners the implementation alternatives that they can apply when they find them fitting for a specific scenario. Having been exposed to code quality concepts, novice learners can be expected to develop their own awareness of code quality and its importance as their experience grows.

Novice-level code quality principles and improvement practices: Overall, the observed learning experiences demonstrate impact of introducing novice programmers to refactor code duplication. The high completion rate among the participants indicates that the subject matter's difficulty and examples are appropriate for the target audience. Despite the challenging technical subject of Part 2, the participants' positive self-reported experience demonstrates that our design principles were effective in engaging the learners, who

overwhelmingly found the scaffolded learning content helpful. Finally, using a convincing, relevant, real-world context, as well as the reflective learning activities may explain why most participants found the tutorial helpful in understanding the benefits of code quality improvement.

One caveat of using an extremely straightforward example is that learners are unlikely to realize how error-prone refactoring can be. When manipulating source code, a common strategy to prevent introducing errors is to use automated programming tools that verify whether an attempted manipulation is safe. It would be unrealistic to expect novice programmers to learn the inner workings of refactoring (e.g., checking preconditions—what needs to hold true to ensure that the semantics of the refactored program would remain intact). Nevertheless, even introductory learners are capable of grasping the behavior-preserving nature of refactoring and can check this property by comparing the refactored program's behavior before and after the refactoring.

Reinforcing core programming concepts: The majority of the participants agreed to the statement that they found Part 1 and Part 2 being helpful in understanding custom blocks. Although we have not asked in what way did each part helped, these results show that learning about Extract Custom Block possibly enhanced their understanding about the purpose and usage of custom blocks. Despite various factors that could have negatively impacted their experience of learning to refactor (e.g., the increased difficulty, the tedious steps involved), the overall participants' learning experience turned out to be quite positive. We take this results as a promising sign that the tutorial succeeded in engaging and encouraging novice programmers to learn how to refactor code duplication while sharpening their understanding of the fundamental concept of procedural abstraction.

Integrating code quality topics in introductory curricula: Our evaluation results show that it is feasible and useful to introduce code quality and its systematic improvement

alongside the fundamental CS concepts. With the tutorial's material related to refactoring learned in a short time, a well-designed software quality intervention can naturally fit the time and content constraints of typical introductory computing in-class lessons. Our results call for computing educators to rethink what is possible when it comes to introducing novice programmers to code quality, a topic traditionally reserved for introduction only much later in the curriculum and often not raised unless students participate in real-world project-based capstone courses.

The extent to which the participants spent time in each task was in line with our expectations. By adjusting some of these tasks, one could affect certain learning experiences (e.g., adding more duplicate program segments may help further highlight the issues of code quality and its improvement, albeit at the risk of overburdening the participants).

One interesting finding is that it is ineffective trying to convince introductory students that a given code snippet's quality should be improved (small code examples are insufficient as a means of conveying the idea of poor-quality software being hard to understand and modify). Indeed, certain code quality concepts and practices cannot be effectively conveyed to introductory learners. As our results suggest, program comprehension is highly subjective and may require different learning activities to explain (e.g., peer code review). In general, it calls for further investigation which concepts and skills can be effectively introduced to novice programmers and how to design the required interventions.

Our findings demonstrate that a properly designed self-paced interactive tutorial can engage introductory learners to learn both the basics of programming and how to improve code quality. Major introductory computing education platforms, such as Scratch and AppInventor [106], can have an important role to play. They can integrate code quality topics into their catalog of programming tutorials as a means of raising the awareness among novice programmers about the importance of code quality.

Constructivist-based learning of code quality concepts: Our approach to introducing novice programmers to code quality aligns well with the Constructivist-guided process "...of learning through experience" or more specifically "learning through reflection on doing" [24], an experiential learning process. Specifically, our goal is to guide students to develop their own experiences about code quality and its importance. Instead of viewing code quality as a constraint to students' learning experiences, we can view the topic as a core competency, a solid foundation that supports their learning in understanding both CS fundamentals and software development practices. One of Scratch's design goals is to encourage learning through creative exploration [75]. In that context, code quality principles and practices play a role that is related to the design principles and guidelines in other creative activities (e.g., graphic design, music remixing, etc.)

Threats to Validity: The results of our study should be interpreted in light of its limitations and threats to validity that we discuss next. Different refactorings may differ in their difficulty to learn and carry out by novice programmers. The results of this study should not be generalized to other refactorings and automated refactoring tools. Constrained by time, we chose to limit the scope of the Extract Custom Block refactoring to removing code duplication without introducing parameters. It remains unknown how introducing Extract Custom Block with parameters would impact our results. In this case study, we only consider the specific design of an automated tool that emphasizes simplicity. Designed differently, the study's automated refactoring tool could have impacted novice programmers dissimilarly.

6.7 Conclusion

We presented an experimental case study of teaching 24 beginner Scratch programmers how to refactor code duplication. We discovered that novice programmers are receptive to

learning refactoring, as long as they are introduced to the subject in a novice-friendly way. Our results also shed light on the ongoing debate about the pedagogical applicability of automated programming tools. Although our study participants clearly preferred having automated refactoring support, teaching refactoring by hand first proved more efficacious from the learning effectiveness standpoint. The research literature has established the value of teaching solid software engineering principles to beginner programmers, and our findings can help computing educators put this vision into reality.

Chapter 7

Conclusions

The central role that software plays in modern society brings the issue of software quality to the forefront of the software engineering research agenda. The research community has marshalled considerable efforts into creating powerful approaches, techniques, and tools that improve the quality of software programs. This dissertation research takes an alternate approach to improving software quality: creating techniques and tools for educating a new generation of programmers who care deeply about software quality. To reach a broad target audience of introductory computing learners, we build on the success of Scratch, the most popular block-based programming language among novice and end-user programmers. Specifically, by applying the results of this research, this target audience can be introduced to code quality concepts alongside the fundamentals of programming.

The **thesis** of this dissertation research is that *it is feasible and advantageous to identify code smells in and provide automated refactoring for Scratch in order to support the introduction of software quality concepts alongside the fundamentals of programming.* To prove this thesis, we took a three-pronged approach: (1) understanding recurring code quality problems in Scratch projects; (2) creating code quality improvement techniques and tools in the form of refactoring; (3) studying the impacts of introducing novice programmers to code quality concepts alongside the fundamentals of programming. Next, we summarize the main findings and contributions of dissertation research in Section 7.1 and provide the concluding remarks in Section 7.4

7.1 Summary of Results

This section summarizes the results and findings that address the research questions of this dissertation research.

7.1.1 Understanding recurring code quality problems in Scratch

Prior works have established limited evidence of the presence of recurring quality problems in Scratch and other educational block-based programming languages. To address this knowledge gap, this dissertation's first contribution (Chapter 3) comprehensively assesses recurring quality problems and their impacts in Scratch projects.

RQ1: What known code smells described in the literature are applicable in the context of Scratch?

Our study of recurring code quality problems in Scratch (Chapter 3) has contributed a catalog of 12 code smells in Scratch (Section 3.3.1). These code smells were identified based on existing code smells in the research literature and then adapted to suit the context and language features of Scratch. These code smells can provide useful vocabulary for programmers and educators to communicate about recurring code quality problems in Scratch. Additionally, the documented rationale and formal definition of each code smell serve as a useful guide for tool designers.

RQ2: What can the analysis of popular Scratch projects teach us about the state of software quality in this programming domain?

Our statistical analysis with 512 popular Scratch projects shows that the projects afflicted by certain code smells (i.e. *Duplicate Code*, *Long Script*, *Uncommunicative Name*) at higher density are likely to have lower modifications in their remixes. This result suggests that poor

code quality can negatively impact Scratch's communal learning, as poorly written projects are unconducive to remixing, an important practice shown to help novice programmers learn new programming concepts.

RQ3: How prevalent and severe are different code smells in the general Scratch code base?

Our assessment of recurring code quality problems in a large representative sample of over one million Scratch projects shows the high prevalence (over 30% of the projects) of several code smells (i.e., *Long Script*, *Uncommunicative Name*, *Duplicate Code*, and *Broad Variable Scope*). To assess their severity, we compared how densely each smell afflicts each projects in the large dataset with the average density of such smell afflicting popular, easy-to-modify projects. We show that many of the problems (e.g., *Broad Variable Scope*, *Uncommunicative Name*, *Duplicate Code*, *Long Script* and *Duplicate String*) severely afflicted a large portion of the studied Scratch projects. These findings provide strong evidence about the issues of poor code quality problems in this domain.

We further explored how prone novice programmers are to introduce code smells into their code (Chapter 4). We collected 3,810 Scratch projects that 116 novice programmers created over time. We applied a survival analysis to study the relationship between the programmers' proneness to four code smells over time (i.e., *Broad Variable Scope*, *Long Script*, *Duplicate Code* and *Uncommunicative Name*) and programmers' characteristics extracted from the projects (e.g., their programming proficiency and prior experience with programming constructs known to reduce code smells).

RQ4: Does the code quality improve, as novice programmers gain experience?

Our longitudinal analysis results reveal that novice programmers were prone to introducing code smells to their projects despite prior exposure to programming constructs associated with avoiding the smells. For example, core programming concepts such as procedures were

not associated with helping them avoid *Duplicate Code* and *Long Script* smells. Similarly, despite multiple exposures to using variables in their code, novice programmers were not less prone to introducing *Broad Variable Scope* to their projects.

RQ5: How persistent are poor coding practices, as novice programmers gain experience?

For the four code smells focused in this study, programmers whose prior projects were more frequently afflicted by a particular type of code smell were more prone to introducing that same smell type in their subsequent projects.

RQ6: Which Scratch constructs and idioms reduce the prevalence of code smells in projects written by novice programmers?

The scenario-based programming idiom (i.e., "broadcast <message>" and "when I receive <message>") is the only one that was associated with avoiding *Long Script* code smell. The core programming constructs such as procedures and variables were not associated with avoiding smells. These result suggested a unique opportunity for educators to introduce novice programmers to code quality and its improvement that are relevant to the core fundamentals of programming being taught.

7.1.2 Code quality improvement for Scratch

Having comprehensively studied code quality problems, the next phase of this dissertation research has focused on creating techniques and tools for improving code quality for Scratch (Chapter 5). An important contribution of this study is four new refactorings (i.e., `Extract Custom Block`, `Extract Constant`, `Extract Parent Sprite`, `Reduce Variable Scope`) that remove highly recurring code smells in Scratch (Section 5.1). We implemented these refactorings by means of our refactoring infrastructure and conducted an experimental evaluation with 448 projects, a representative dataset in terms of size and complexity.

RQ7: How applicable is each introduced refactoring?

All refactorings except `Extract Parent Sprite` are highly applicable with over 75% success rate when applied to detected code smells in the experimental dataset (Section 5.4.1). The applicability of *Extract Parent Sprite* refactoring was relatively low because of its restrictive refactoring constraint.

RQ8: How do the refactorings impact code quality?

Each refactoring impacts the refactored project' code quality differently. When applied, the refactorings that eliminate duplication-related problems improve code quality, with the code size decreasing and projects becoming easier to understand and modify. Other aspects of quality improvements are the direct result of or can be inferred from applying refactorings. For example, Extract Custom Block directly increases abstraction of the refactored project and possibly increase the modifiability and modularity of the project. However, as captured by the quality metrics before and after the refactoring, the extent of improvement varied on accordance with the severity of the refactored code smells (Section 5.4.2).

We also conducted a preliminary user study with 24 novice programmers to ascertain the usability of our tool's design for novice programmers and study the impact of the availability of automated refactoring support. The majority of participants in the experimental group reported automated refactoring as being useful (54% very useful and 38% extremely useful).

RQ9: Does the presence of automated refactoring support affect how willing novice programmers are to refactor their code?

Having been primed to use custom blocks to improve program comprehensibility, participants were encouraged to improve their code which is amenable to Extract Custom Block refactoring. Given the availability of automated refactoring support, the experimental participants were more likely to refactor code duplication as compared to the control participants

who were given a manual step-by-step refactoring guide (χ^2 (1, N = 24) = 8.48, P=.004) (Section 5.4.3). The study results provide encouraging results about the potential benefits of supporting novice programmers with automated refactoring to encourage code quality improvement practices.

7.1.3 Introducing code quality concepts alongside the fundamentals of programming

Although our refactoring tools are sufficiently usable even for novice programmers, the implications of introducing refactoring to this target audience have not be systematically studied. This dissertation research has addressed this problem by experimentally studying the introduction of refactoring alongside the fundamentals of programming, as well as the impacts of automated refactoring tools.

We conducted an exploratory case study with another 24 novice Scratch programmers learning to refactor code duplication (Chapter 6). The results provided useful insights about our approach and its effectiveness. By exposing novice programmers to learning how to refactor using both the manual and automated methods, we better understood the role of automated refactoring in their learning and programming practice.

RQ10: Do novice programmers perceive enjoyment and difficulty differently when learning how to use procedural abstraction as compared to learning how to refactor code duplication?

Participants reported positive perceptions about their experiences when learning both the refactoring to remove code duplication and procedural abstraction. They found learning both topics to be enjoyable, although learning how to refactor proved more challenging. They found learning the refactoring with the *manual method* to be more enjoyable as compared to the *automated method*. Learning automated refactoring first decreased the perceived

difficulty of the *manual method*.

RQ11: Does introducing novice programmers to the Extract Custom Block refactoring help them understand the concept of custom blocks?

The majority of novice programmers in both automated-first and manual-first groups prefer to learn the code duplication refactoring manually first. The main reason is that learning the *manual method* with a rudimentary example helped them understand what the automated tool did behind the scenes.

Nevertheless, when asked about the method they would prefer to use to refactor code duplication in the future, the majority of them chose automated refactoring. One explanation is that refactoring with automated support is faster and simpler, given that they have a conceptual understanding of how the refactoring works.

RQ12: Does the preference for manual or automated refactoring methods change and why, as novice programmers progress from learning refactoring to intending to refactor in the future?

Participants reported that the tutorial helped them understand why the Extract Custom Block refactoring can make their code easier to understand and modify. However, they found the tutorial less helpful when it comes to helping them understand why code duplication makes their code harder to understand and modify.

Overall, our findings show that automated refactoring support can play an important role for novice programmers to engage in improving the quality of their code. However, introducing refactoring by hand first using rudimentary examples can be more effective from the learning effectiveness standpoint.

7.2 Implications

We highlight three key implications drawn from our research findings.

1. **Automated refactoring for block-based programming environments** When the line is blurred between learning programming and software development, block-based programming environments need to be enhanced with automated refactoring to support these two interrelated activities. Programmers in this domain often engage in creating non-trivial programming projects regardless of their learning experience. Our research findings show that supporting block-based programmers with automated refactoring is not only applicable but highly beneficial to raise the ceiling of the programming environments for blocks' affordances for systematically improving code quality.

2. **Teaching refactoring to reinforce core programming fundamentals** Although Scratch makes it easy to start programming, recurring code quality problems highlight the challenges that computing educators face when it comes to conveying core programming concepts to novice programmers. As our study shows, code smells and refactorings commonly found in this domain are associated with basic programming constructs. Our findings demonstrate how teaching refactoring not only helps learners deepen their understanding of important core concepts, but also offers a great opportunity for conveying the importance of code quality and its improvement.

3. **An experiential-based approach to the learning of code quality and its improvement** Often seen as imposing rules on how program should be written, code quality topics stand in contrast to the exploratory, active learning style of the constructivism paradigm. However, it can be slow or difficult for novice programmers to develop the knowledge and practice of code quality without a deliberate effort that

guides the learning. Our research shows how learning about code quality can be made compatible with constructivism through instructions that guide experiential learning, thereby making the learning of the subject matter engaging and effective. Our research insights call for computing educators to rethink what is possible when it comes to introducing novice programmers to code quality and its improvement.

7.3 Future Work

Our research findings open up opportunities for research necessary to reach the long-term vision of promoting the culture of quality from the ground up.

Providing automated refactoring for block-based programming environments: Future research could continue to explore various aspects of automated refactoring for block-based programming languages. Currently, the software facilities for a structured blocks programming editor expose a limited language processing infrastructure for program analysis and transformation. Future research on block-based languages and environments should consider how such facilities can be more comprehensively supported. Additionally, there are multiple research opportunities to explore the design space of the user interfaces for refactorings. We have documented a general design strategy for designing the refactoring tools that has shown to be helpful for beginner learners in two key areas: detecting refactoring opportunities and carrying out the refactorings. However, our research has not fully explored various design aspects of the refactoring tools, such as visualization techniques for communicating refactoring opportunities to beginners and the refactoring changes. Finally, more research is needed to design and evaluate automated support for other refactorings beside *Code Duplication* and *Custom Block refactoring*.

Learning outcomes of introducing novice programmers to code quality and its

improvement: It is the question of future research to investigate the full extent of the long-term learning impact of teaching code quality and its improvement to novice programmers. The findings of our controlled user studies thus far helped us understand learners' perception of code quality and their prospects of engaging in code quality improvement practice. However, there is still a lot of work required to understand how introducing code quality impacts novice programmers in how they integrate the practice into their day-to-day programming process. Investigating this long-term learning impact might prove important, as it provides strong evidence for a community buy-in, required to transform the way we teach: making code quality an integral part of introductory computing education.

Introducing novice programmers to code quality and its improvement: To promote code quality education, more research efforts should be made to explore various software quality topics and the approaches that are appropriate for novice programmers. In this dissertation, we focus on introducing code duplication refactoring, its impact and learning outcomes that help inform similar efforts aiming to help novice programmers learn about code quality and its systematic improvement effectively. The future research effort should explore how to effectively incorporate code quality learning materials into the block-based programming environments so as to introduce the subject matters to novice programming audience both in informal, open-ended learning settings and in formal, classroom settings.

7.4 Final Remarks

In this dissertation, we put forward our vision of drastically improving software quality by promoting the culture of quality from the ground up. Our empirical research findings shows how prevalent and severe code quality issues in Scratch. The findings suggest how these issues can negatively impact novice programmers and why providing them the means to

improve the quality of their code in the form of automated refactoring can be potentially beneficial. This dissertation sheds light on the feasibility and advantages of adding automated refactoring support to block-based programming environments as well as introducing novice programmers to code quality and its improvement alongside the fundamentals of programming.

This dissertation research presents evidence confirming that our long-term vision is realizable, but much additional work remains to be done. For automated refactoring to become a standard feature of the programming environments for blocks, both researchers and practitioners have to recognize that programmers in this domain need to improve code quality. More research is required to improve the usability of refactoring support for novice and end-user programmers. Finally, future studies should explore approaches that introduce code quality concepts alongside the fundamentals of programming as well as evaluating the long-term impacts of this educational intervention on novice programmers.

Bibliography

[1] Efthimia Aivaloglou and Felienne Hermans. 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, pages 53–61, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4449-4. doi: 10.1145/2960310.2960325. URL http://doi.acm.org/10.1145/2960310.2960325.

[2] Tiago L Alves, Christiaan Ypma, and Joost Visser. Deriving metric thresholds from benchmark data. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[3] Creative Computing Lab at the Harvard Graduate School of Education. Creative Computing Curriculum. https://creativecomputing.gse.harvard.edu/guide/curriculum.html, 2020. Online; accessed 24 November 2019.

[4] S. Badame and D. Dig. Refactoring meets spreadsheet formulas. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 399–409, Sept 2012. doi: 10.1109/ICSM.2012.6405299.

[5] Albert Bandura. *Social foundations of thought and action*. Englewood Cliffs, NJ: Prentice Hall, 1986.

[6] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. Learnable programming: Blocks and beyond. *Commun. ACM*, 60(6):72–80, May 2017. ISSN 0001-0782. doi: 10.1145/3015455. URL http://doi.acm.org/10.1145/3015455.

[7] Bryce Boe, Charlotte Hill, Michelle Len, Greg Dreschler, Phillip Conrad, and Diana

Franklin. Hairball: Lint-inspired static analysis of Scratch projects. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 215–220. ACM, 2013.

[8] Jürgen Börstler, Harald Störrle, Daniel Toll, Jelle van Assema, Rodrigo Duran, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, and Bonnie MacKellar. "i know it when i see it" perceptions of code quality: Iticse '17 working group report. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports*, ITiCSE-WGR '17, page 70–85, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356275. doi: 10.1145/3174781.3174785. URL https://doi.org/10.1145/3174781.3174785.

[9] Jürgen Börstler, Harald Störrle, Daniel Toll, Jelle van Assema, Rodrigo Duran, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, and Bonnie MacKellar. "i know it when i see it" perceptions of code quality: Iticse '17 working group report. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports*, ITiCSE-WGR '17, page 70–85, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356275. doi: 10.1145/3174781.3174785. URL https://doi.org/10.1145/3174781.3174785.

[10] Michael Bray, Kimberly Brune, David A Fisher, John Foreman, and Mark Gerken. C4 software technology reference guide-a prototype. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1997.

[11] Karen Brennan and Mitchel Resnick. New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 annual meeting of the American educational research association, Vancouver, Canada*, volume 1, page 25, 2012.

[12] William H Brown, Raphael C Malveau, Hays W McCormick, and Thomas J Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[13] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 156–165. IEEE, 2010.

[14] Christopher Chambers and Christopher Scaffidi. Smell-driven performance analysis for end-user programmers. In *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on*, pages 159–166. IEEE, 2013.

[15] Henrik Bundefinedrbak Christensen. Systematic testing should not be a topic in the computer science curriculum! In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '03, page 7–10, New York, NY, USA, 2003. Association for Computing Machinery. URL https://doi.org/10.1145/961511.961517.

[16] David Collett. *Modelling survival data in medical research*. CRC press, 2015.

[17] J. P. d. Reis, F. Brito e Abreu, and G. d. F. Carneiro. Code smells incidence: Does it depend on the application domain? In *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 172–177, Sep. 2016. doi: 10.1109/QUATIC.2016.044.

[18] Sayamindu Dasgupta, William Hale, Andrés Monroy-Hernández, and Benjamin Mako Hill. Remixing as a pathway to computational thinking. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*, pages 1438–1449. ACM, 2016.

[19] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[20] R Geoff Dromey. Cornering the chimera [software quality]. *IEEE Software*, 13(1): 33–43, 1996.

[21] Stephen H. Edwards. Rethinking computer science education from a test-first perspective. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, page 148–155, New York, NY, USA, 2003. Association for Computing Machinery. URL https://doi.org/10.1145/949344.949390.

[22] ME Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[23] Amin Milani Fard and Ali Mesbah. JSNOSE: Detecting JavaScript code smells. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 116–125. IEEE, 2013.

[24] Patrick Felicia. *Handbook of research on improving learning and motivation through educational games: Multidisciplinary approaches: Multidisciplinary approaches*. iGi Global, 2011.

[25] Ewen Ferguson, Scott Bolderson, and Belton Flournoy. *Minimising Spreadsheet Errors*. ICAEW, 2011.

[26] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[27] N Fraser et al. Blockly: A visual programming editor. *URL: https://developers.google.com/blockly/*, 2013.

[28] R Fuhrer, Markus Keller, and A Kiezun. Advanced refactoring in the eclipse jdt: Past, present, and future. In *Proc. ECOOP Workshop on Refactoring Tools (WRT)*, pages 31–32, 2007.

[29] Michal Gordon, Assaf Marron, and Orni Meerbaum-Salant. Spaghetti for the main course?: observations on the naturalness of scenario-based programming. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, pages 198–203. ACM, 2012.

[30] Shuchi Grover and Roy Pea. Using a discourse-intensive pedagogy and Android's App Inventor for introducing computational concepts to middle school students. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 723–728, 2013.

[31] Francisco J. Gutierrez, Jocelyn Simmonds, Nancy Hitschfeld, Cecilia Casanova, Cecilia Sotomayor, and Vanessa Peña Araya. Assessing software development skills among K-6 learners in a project-based workshop with Scratch. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, ICSE-SEET '18, page 98–107, New York, NY, USA, 2018. Association for Computing Machinery. URL https://doi.org/10.1145/3183377.3183396.

[32] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):33, 2014.

[33] Görel Hedin and Eva Magnusson. JastAdd—an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[34] F. Hermans and E. Aivaloglou. Do code smells hamper novice programming? a controlled experiment on Scratch programs. In *2016 IEEE 24th International Conference*

on Program Comprehension (ICPC), pages 1–10, May 2016. doi: 10.1109/ICPC.2016.7503706.

[35] Felienne Hermans and Efthimia Aivaloglou. 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, pages 13–22, Piscataway, NJ, USA, 2017. IEEE Press. URL https://doi.org/10.1109/ICSE-SEET.2017.13.

[36] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Detecting and refactoring code smells in spreadsheet formulas. *Empirical Software Engineering*, 20(2):549–575, 2015.

[37] Felienne Hermans, Kathryn T Stolee, and David Hoepelman. Smells in block-based programming languages. In *Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on*, pages 68–72. IEEE, 2016.

[38] Trudy Howles. Fostering the growth of a software quality culture. *SIGCSE Bull.*, 35 (2):45–47, June 2003. ISSN 0097-8418. doi: 10.1145/782941.782978. URL https://doi.org/10.1145/782941.782978.

[39] Dmitry Jemerov. Implementing refactorings in intellij idea. In *Proceedings of the 2nd Workshop on Refactoring Tools*, pages 1–2, 2008.

[40] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[41] Capers Jones and Olivier Bonsignour. *The economics of software quality*. Addison-Wesley Professional, 2011.

[42] Brian W Kernighan and Phillip James Plauger. The elements of programming style. *New York: McGraw-Hill, 1974, 2nd ed.*, 1974.

[43] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. Code quality issues in student programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17, pages 110–115, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4704-4. doi: 10.1145/3059009.3059061.

[44] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. How teachers would help students to improve their code. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, pages 119–125, 2019.

[45] P. Khawas, P. Techapalokul, and E. Tilevich. Unmixing remixes: The how and why of not starting projects from Scratch. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct 2019.

[46] B. Kitchenham and S. L. Pfleeger. Software quality: the elusive target [special issues section]. *IEEE Software*, 13(1):12–21, Jan 1996. ISSN 1937-4194. doi: 10.1109/52.476281.

[47] David G Kleinbaum and Mitchel Klein. *Survival analysis: a self-learning text*. Springer Science & Business Media, 2006.

[48] A Güne\cs Koru, Dongsong Zhang, Khaled El Emam, and Hongfang Liu. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Transactions on Software Engineering*, 35(2):293–304, 2009.

[49] A Gunes Koru, Dongsong Zhang, and Hongfang Liu. Modeling the effect of size on defect proneness for open-source software. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, page 10. IEEE Computer Society, 2007.

[50] Dave Krebs, Alexander Conrad, and Jingtao Wang. Combining visual block programming and graph manipulation for clinical alert rule building. In *CHI'12 Extended Abstracts on Human Factors in Computing Systems*, pages 2453–2458. 2012.

[51] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[52] Chuck Leska. Testing across the curriculum: Square one! *J. Comput. Sci. Coll.*, 19 (5):163–169, May 2004. ISSN 1937-4771.

[53] I. Li, F. Turbak, and E. Mustafaraj. Calls of the wild: Exploring procedural abstraction in app inventor. In *2017 IEEE Blocks and Beyond Workshop (B&B)*, pages 79–86, Oct 2017. doi: 10.1109/BLOCKS.2017.8120417.

[54] Sze Yee Lye and Joyce Hwee Ling Koh. Case studies of elementary children's engagement in computational thinking through scratch programming. In *Computational Thinking in the STEM Disciplines*, pages 227–251. Springer, 2018.

[55] David J. Malan and Henry H. Leitner. Scratch for budding computer scientists. *SIGCSE Bull.*, 39(1):223–227, March 2007. ISSN 0097-8418. doi: 10.1145/1227504.1227388. URL http://doi.acm.org/10.1145/1227504.1227388.

[56] Maria José Marcelino, Teresa Pessoa, Celeste Vieira, Tatiana Salvador, and An-

tónio José Mendes. Learning computational thinking and scratch at distance. *Computers in Human Behavior*, 80:470–477, 2018.

[57] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, pages 308–320, Dec 1976. ISSN 0098-5589. doi: 10.1109/TSE.1976.233837.

[58] Steve McConnell. *Code complete*. Pearson Education, 2004.

[59] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. Habits of programming in scratch. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, pages 168–172. ACM, 2011.

[60] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, Feb 2004. ISSN 0098-5589. doi: 10.1109/TSE.2004.1265817.

[61] Iwona Miliszewska and Grace Tan. Befriending computer programming: A proposed approach to teaching introductory programming. *Informing Science: International Journal of an Emerging Transdiscipline*, 4(1):277–289, 2007.

[62] Andrés Monroy-Hernández and Mitchel Resnick. Empowering kids to create and share programmable media. *interactions*, 15(2):50–53, 2008.

[63] Jesús Moreno and Gregorio Robles. Automatic detection of bad programming habits in scratch: A preliminary study. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, pages 1–4. IEEE, 2014.

[64] Jesús Moreno-León, Gregorio Robles, and Marcos Román-González. Dr. Scratch: Automatic analysis of Scratch projects to assess and foster computational thinking. *RED. Revista de Educación a Distancia*, (46):1–23, 2015.

[65] Jesús Moreno-León, Marcos Román-González, Casper Harteveld, and Gregorio Robles. 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*, pages 2788–2795. ACM, 2017.

[66] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. *2009 IEEE 31st International Conference on Software Engineering*, pages 287–297, 2009. doi: 10.1109/ICSE.2009.5070529.

[67] Kwankamol Nongpong and John Tang Boyland. Integrating "Code Smells" Detection with Refactoring Tool Support. 3523928(August):154, 2012. URL https://vpn.utm.my/docview/1074792412?accountid=41678.

[68] Yoshiki Ohshima, Jens Mönig, and John Maloney. A module system for a general-purpose blocks language. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, pages 39–44. IEEE, 2015.

[69] Steffen Olbrich, Daniela S Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement*, pages 390–400. IEEE Computer Society, 2009.

[70] William F Opdyke. *REFACTORING OBJECT-ORIENTED FRAMEWORKS*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[71] J. L. Overbey and R. E. Johnson. Differential precondition checking: A lightweight, reusable analysis for refactoring tools. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 303–312, Nov 2011. doi: 10.1109/ASE.2011.6100067.

[72] Eli Tilevich Peeratham Techapalokul, Simin Hall. Teaching the culture of quality from the ground up: Novice-tailored quality improvement for scratch programmers. In *2020 ASEE Annual Conference & Exposition*. ASEE Conferences, 2020.

[73] Roger S Pressman. *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005.

[74] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2016. URL https://www.r-project.org/.

[75] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[76] Mona Rizvi and Thorna Humphries. A scratch-based cs0 course for at-risk computer science majors. In *2012 Frontiers in Education Conference Proceedings*, pages 1–5. IEEE, 2012.

[77] Mona Rizvi, Thorna Humphries, Debra Major, Meghan Jones, and Heather Lauzun. A cs0 course using scratch. *Journal of Computing Sciences in Colleges*, 26(3):19–27, 2011.

[78] Gregorio Robles, Jesús Moreno-León, Efthimia Aivaloglou, and Felienne Hermans. Software clones in scratch projects: On the presence of copy-and-paste in computational thinking learning. In *Software Clones (IWSC), 2017 IEEE 11th International Workshop on*, pages 1–7. IEEE, 2017.

[79] Barbara Rogoff. *Social interaction as apprenticeship in thinking: Guided participation in spatial planning.* American Psychological Association, 1991.

[80] Simon P. Rose, M.P. Jacob Habgood, and Tim Jay. Using Pirate Plunder to develop children's abstraction skills in Scratch. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI EA '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450359719. doi: 10.1145/3290607.3312871. URL https://doi.org/10.1145/3290607.3312871.

[81] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470 – 495, 2009. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2009.02.007. URL http://www.sciencedirect.com/science/article/pii/S0167642309000367.

[82] José-Manuel Sáez-López, Marcos Román-González, and Esteban Vázquez-Cano. 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, 2016.

[83] Gehan MK Selim, Liliane Barbour, Weiyi Shang, Bram Adams, Ahmed E Hassan, and Ying Zou. Studying the impact of clones on software defects. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 13–21. IEEE, 2010.

[84] Mazyar Seraj, Serge Autexier, and Jan Janssen. BEESM, a block-based educational programming tool for end users. In *Proceedings of the 10th Nordic Conference on Human-Computer Interaction*, NordiCHI '18, page 886–891, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450364379. doi: 10.1145/3240167.3240239. URL https://doi.org/10.1145/3240167.3240239.

[85] Tushar Sharma and Diomidis Spinellis. A survey on software smells. *Journal of Systems and Software*, 138:158 – 173, 2018. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2017.12.034. URL http://www.sciencedirect.com/science/article/pii/S0164121217303114.

[86] Ben Skudder and Andrew Luxton-Reilly. Worked examples in computer science. In *Proceedings of the Sixteenth Australasian Computing Education Conference-Volume 148*, pages 59–64. Australian Computer Society, Inc., 2014.

[87] Megan Smith. Computer science for all, 2016. URL https://www.whitehouse.gov/blog/2016/01/30/computer-science-all.

[88] Jaime Spacco and William Pugh. Helping students appreciate test-driven development (TDD). *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA '06*, page 907, 2006. doi: 10.1145/1176617.1176743. URL http://portal.acm.org/citation.cfm?doid=1176617.1176743.

[89] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. Designing a rubric for feedback on code quality in programming courses. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, pages 160–164, 2016.

[90] Kathryn T Stolee and Sebastian Elbaum. Refactoring pipe-like mashups for end-user programmers. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 81–90. ACM, 2011.

[91] Kathryn T Stolee and Sebastian Elbaum. Identification, impact, and refactoring of smells in pipe-like web mashups. *IEEE Transactions on Software Engineering*, 39(12): 1654–1679, 2013.

[92] Herb Sutter. *C++ coding standards: 101 rules, guidelines, and best practices*. Pearson Education India, 2004.

[93] Karen Swan. A constructivist model for thinking about learning online. In *Elements of Quality Online Education: Engaging Communities, Volume 6 in the Sloan-C Series Sloan-C Foundation*, pages 13–31. 2005.

[94] Gábor Szőke, Csaba Nagy, Lajos Jenő Fülöp, Rudolf Ferenc, and Tibor Gyimóthy. Faultbuster: An automatic code smell refactoring toolset. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 253–258. IEEE, 2015.

[95] P. Techapalokul and E. Tilevich. Novice programmers and software quality: Trends and implications. In *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE&T)*, pages 246–250, Nov 2017. doi: 10.1109/CSEET.2017.47.

[96] P. Techapalokul and E. Tilevich. Quality Hound — an online code smell analyzer for Scratch programs. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 337–338, Oct 2017. doi: 10.1109/VLHCC.2017.8103498.

[97] P. Techapalokul and E. Tilevich. Code quality improvement for all: Automated refactoring for Scratch. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct 2019.

[98] Peeratham Techapalokul and Eli Tilevich. Understanding recurring quality problems and their impact on code sharing in block-based software. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, 2017.

[99] Terry Therneau, Cindy Crowson, and Elizabeth Atkinson. Using time dependent covariates and time dependent coefficients in the cox model. 2017.

[100] Terry M Therneau and Thomas Lumley. Package 'survival', 2016.

[101] Damla Topalli and Nergiz Ercil Cagiltay. Improving programming skills in engineering education through problem-based game projects with scratch. *Computers & Education*, 120:64–74, 2018.

[102] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 97–106. IEEE, 2002.

[103] Mark J Van Gorp and Scott Grissom. An empirical evaluation of using constructive classroom activities to teach introductory programming. *Computer Science Education*, 11(3):247–260, 2001.

[104] David Weintrop, Afsoon Afzal, Jean Salac, Patrick Francis, Boyang Li, David C. Shepherd, and Diana Franklin. Evaluating coblox: A comparative study of robotics programming environments for adult novices. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356206. doi: 10.1145/3173574.3173940. URL https://doi.org/10.1145/3173574.3173940.

[105] Niklaus Wirth. Program development by stepwise refinement. In *Pioneers and Their Contributions to Software Engineering*, pages 545–569. Springer, 2001.

[106] David Wolber. App Inventor and real-world motivation. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 601–606, 2011.

[107] David Wolber, Harold Abelson, and Mark Friedman. Democratizing computing with app inventor. *GetMobile: Mobile Comp. and Comm.*, 18(4):53–58, January 2015.

ISSN 2375-0529. doi: 10.1145/2721914.2721935. URL http://doi.acm.org/10.1145/2721914.2721935.

[108] Pree Wolfgang. Design patterns for object-oriented software development. *Reading Mass*, 15, 1994.

[109] Ursula Wolz, John Maloney, and Sarah Monisha Pulimood. 'Scratch' Your Way To Introductory Cs. *ACM SIGCSE Bulletin*, 40(1):298, 2008. ISSN 00978418. doi: 10.1145/1352322.1352239.

[110] Zhenchang Xing and Eleni Stroulia. Refactoring practice: How it is and how it should be supported-an eclipse case study. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 458–468. IEEE, 2006.

[111] Aiko Yamashita. Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data. *Empirical Software Engineering*, 19(4):1111–1143, 2014.

[112] Diyi Yang, Tanmay Sinha, David Adamson, and Carolyn Penstein Rosé. Turn on, tune in, drop out: Anticipating student dropouts in massive open online courses. In *Proceedings of the 2013 NIPS Data-driven education workshop*, volume 11, page 14, 2013.

[113] Diyi Yang, Miaomiao Wen, Iris Howley, Robert Kraut, and Carolyn Rose. Exploring the effect of confusion in discussion forums of massive open online courses. In *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*, pages 121–130. ACM, 2015.

[114] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets.

[115] LeChen Zhang and Jalal Nouri. A systematic review of learning computational thinking through Scratch in K-9. *Computers & Education*, 141:103607, 2019.

Appendices

Appendix A

Supplementary Materials

A.1 Quality Hound

QualityHound—improves the utility of software quality analysis for block-based software by placing at the user's fingertips a visual, browser-based code smell analyzer for Scratch projects[1]. The analysis engine of QualityHound comprises the 12 state-of-the-art quality analyzers that we originally developed to produce the experimental results for the paper, appearing in the main technical program of the VL/HCC 2017 conference. In the block-based programming community, researchers thus far have made few of their quality analysis tools available to end users for learning and experimentation. Our work is inspired by Dr. Scratch[2] [64], a browser-based tool for assessing Scratch programming proficiency that also detects three code smells (i.e., sprite naming, duplicated code and dead code) in the analyzed code. Dr. Scratch presents the detected smells textually, making it challenging for programmers to trace the identified smells back to the corresponding visual program parts.

QualityHound parses JSON-based ASTs of Scratch programs into an internal representation used by the analysis routines. The analyzers are implemented using JastAdd [33], a Java-based language processing framework. The detected code smells are expressed as block regions, so the smelly blocks or scripts can then be reported to the user. Finally, the

[1]The passages in this section are based on our VL/HCC showpiece publication [96]
[2]http://www.drscratch.org/

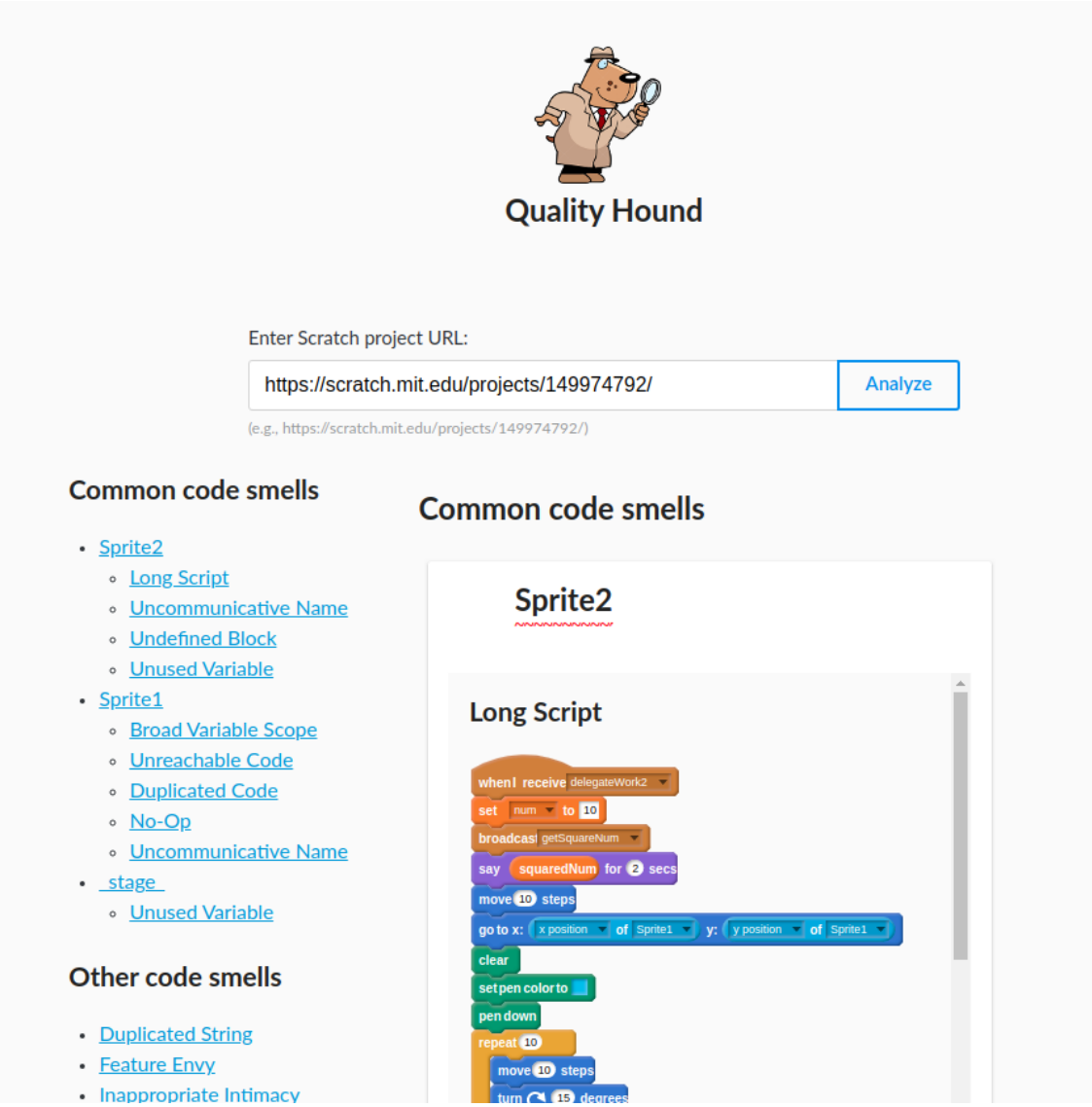


Figure A.1: Screenshot of QualityHound interface

presentation layer formats the detected blocks and scripts, to be rendered in the browser by a third-party JavaScript library[3].

QualityHound renders our analysis infrastructure available to end users, who are expected to possess little to no expertise in software quality analysis. The browser-based user interface lowers the learning curve and increases usability. The tool takes the URL of a Scratch project

[3] https://github.com/scratchblocks/scratchblocks

as input and then presents the detected smells visually to the user in a web browser window. In Figure A.1, one can see QualityHound in action and a screenshot of the tool showing the detected smells. Users with some familiarity of the analyzed project can leverage the tool's visual output to navigate to the exact locations of the detected smells. By better understanding why the tool signaled the presence of code smells, programmers can determine if they should engage in quality improvement.

A.2 Survival Data Analysis

A.2.1 Data format for Recurrent Event Modeling

Table A.1 shows a simplified input data used to study the impact of various predictors on the programmers introducing BV smell. Each row represents a programmer with the baseline explanatory variables and the $etime$'s represent the first five events of code smell afflicted projects. For example, programmer id=2, introduced BV code smell in 3 of the first 15 baseline projects. In the observation period of 8 projects (indicated by $futime$), the programmer introduced BV-smell afflicted project again in project no.7 (shown in $etime1$ column).

Since projects containing code smell are recurring events, we use the *Counting Process* (CP) format[47, 99] to pass multiple lines of data representing the recurring events for the same individual as input to the Cox model. We transform the time dependent variables of recurring smell events into a new format in Table A.2. The new format has two time points ($tstart$, and $tstop$) and the smell status. The programmer id=2 now is represented in two rows, one with time (0-7] with the smell status =1 and the other (7-8] with the status of 0 indicating the censored status (once the subject is censored, no further information is

id	BVS	views	exp	CC	futime	etime1	etime2	etime3	etime4
1	0	191	2	1	1	NA	NA	NA	NA
2	3	331	2	1	8	7	NA	NA	NA
3	2	309	3	2	5	4	NA	NA	NA
4	0	117	2	1	7	NA	NA	NA	NA

Table A.1: Simplified input data (The first four rows with a reduced set of explanatory variables)

id	BVS	views	exp	CC	futime	tstart	tstop	status
1	0	191	2	1	1	0	1	0
2	3	331	2	1	8	0	7	1
2	3	331	2	1	8	7	8	0
3	2	309	3	2	5	0	4	1

Table A.2: Simplified data layout used in the study (The first four rows with a reduced set of explanatory variables)

available). Since the same values of explanatory variables apply over the two time intervals, they are the same in the two rows. Note that the analysis time is the project sequence number rather than the physical time, similar to the approach used in [18].

A.2.2 Model Development Approach

Two important considerations for model development are the selection of explanatory variables, and the assessment to decide if the model fits well. According to Collett [16], in practice, one selects a set of significant explanatory variables through a combination of knowledge of the science, trial and errors and automatic variable selection procedures. We build separate models for each analysis of code smell by following Collett's approach to model development.

A.2.3 Checking proportional hazards assumption

The final model needs to be checked if there exists any interaction between any of the explanatory variables with time. We test the correlation between the *Schoenfeld* residuals and survival time with the null hypothesis being the correlation of zero which supports the proportional hazards assumption[99]. That is, if the test result is highly insignificant, we accept the null hypothesis. We also visually inspect this assumption by plotting a graph of the scaled Schoenfeld residuals, along with a smooth curve that represents (t). If the PH assumption is met, the fitted curve should look horizontal, as the Schoenfeld residuals would be independent of survival time.

A.2.4 Using extended Cox model

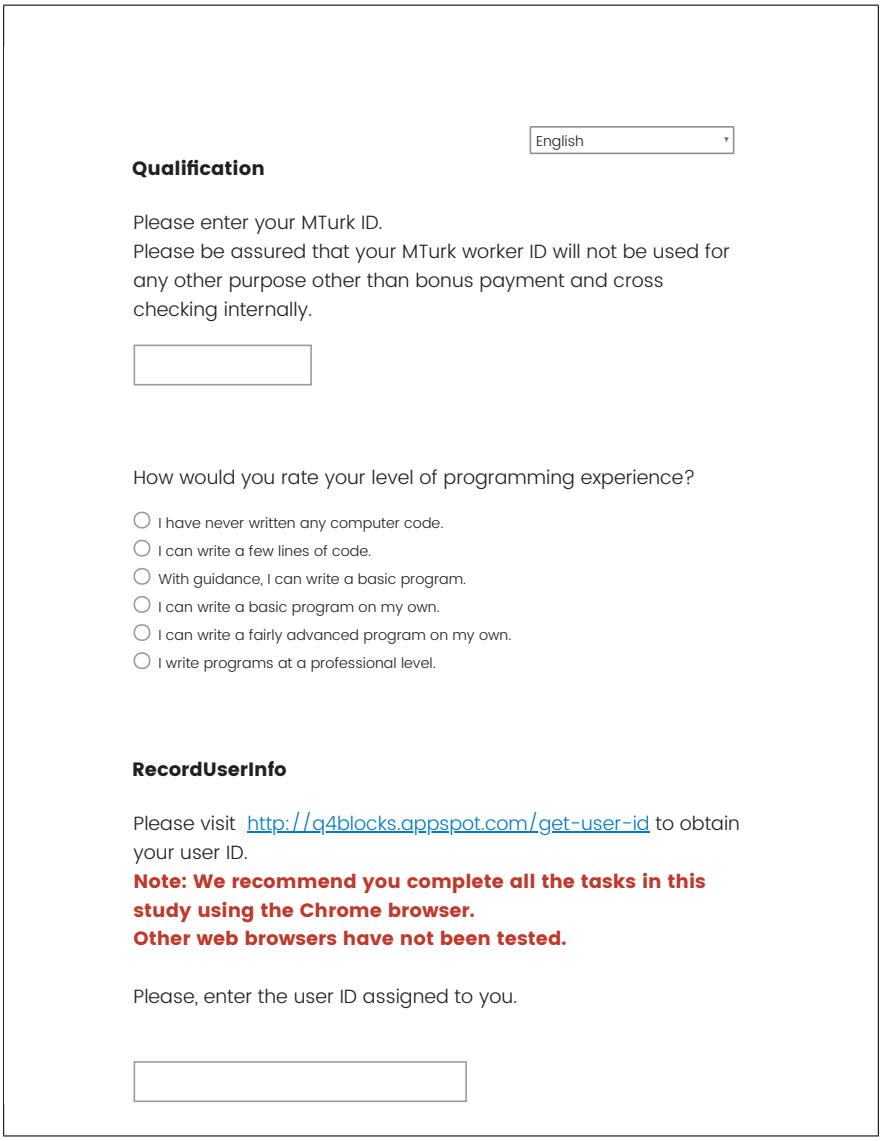
When the Cox proportional hazard model assumption is violated for a variable, we use the extended Cox model where the variable is stratified. This simply means the effect of such variable is not constant over time. The *stratified Cox model* (SC) model is a modification of the Cox proportional hazard (PH) model to allow for control by "stratification" of the explanatory variable not satisfying the PH assumption. Only variables that satisfy the PH assumption are included in the model as predictors; the stratified variables are not included in the model, though they can still be visualized. The extended model allows the dataset to be divided as strata and create a different baseline hazard for each strata.

A.3 Learning Experience Survey 2

Learning to refactor duplication with manual-first vs. automated-first method

The online survey is dynamically displayed based on the following variables:

- taskid=jumping-square (part 1) or sliding-square (part 2)
- Automated-first: first=Code Wizard (automated), second=Manual
- Manual-first: first=Manual, second=Code Wizard (automated)



English

Qualification

Please enter your MTurk ID.
Please be assured that your MTurk worker ID will not be used for any other purpose other than bonus payment and cross checking internally.

How would you rate your level of programming experience?

- I have never written any computer code.
- I can write a few lines of code.
- With guidance, I can write a basic program.
- I can write a basic program on my own.
- I can write a fairly advanced program on my own.
- I write programs at a professional level.

RecordUserInfo

Please visit http://q4blocks.appspot.com/get-user-id to obtain your user ID.
Note: We recommend you complete all the tasks in this study using the Chrome browser.
Other web browsers have not been tested.

Please, enter the user ID assigned to you.

VerifyQualification

...loading...

Informed Consent

Welcome to the research study!

Researchers at Virginia Tech are interested in understanding how novice programmers write code using Scratch. You will be presented with information relevant to the basics of programming with Scratch and asked to do a few short programming exercises and answer some survey questions. Please be assured that your responses will be kept completely confidential.

The study should take you around 30-45 minutes to complete, and you may receive up to $6 for your participation. Your participation in this research is voluntary. You have the right to withdraw at any point during the study, for any reason. There are 2 parts of the study with each part takes around 15-20 minutes to complete. The programming task of the survey is designed to be taken in sequence, meaning that unless you complete the previous part, you will not be able to proceed to the next part. You will receive $2 in compensation for Part 1, $4 bonus for Part 2.

Please note that participating in this study requires a laptop or desktop computer. Chrome browser is recommended.

This study is IRB approved through Virginia Tech Institutional Review Board (Research Protocol # 18-639)

The detailed consent form of this survey study can be found here.

- ○ I consent, begin the study
- ○ I do not consent, I do not wish to participate

Non-Consent

Thank you for your interest to participate in this study. Nothing has been recorded. You may close this window.

Participant Info

What is your age?

What is your gender?

- ○ Male
- ○ Female
- ○ Others (specify)

What is the highest degree or level of school you have completed? If currently enrolled, highest degree received.

Less than high school	High school graduate	Some college	2 year degree	4 year degree	Professional degree	Doctorate
○	○	○	○	○	○	○

Part1-Programming

Visit https://q4blocks.appspot.com/?taskid=${e://Field/taskId} to start **Part 1.**
Upon completion, please enter Part 1's completion code for the programming task in the text box below. You will upload your completed program next.

Note: If you were not able to complete this programming part, please enter **TERMINATE** (all capital letters) for the programming part's completion code to end this survey.
You will be asked to provide your MTurk worker ID in the next page so that we can compensate your time and effort via a bonus payment.

TERMINATE1

Is there anything else that you would like to share with us about this research study? Please let us know how we can improve our study.

Since you did not complete Part 1, you will NOT receive a survey completion code to enter into the Mturk HIT assignment.

In the next few days, we will create a special HIT with $0.01 reward to pay you.
You will then receive the additional bonus amount from that HIT

based on how much you have completed the study.

Do not forget to click "Submit" or your response may not be recorded.

Part1-Programming File

Please upload your completed Part 1 program. From the Scratch editor, click File > Save to your computer.

Part1 Post Survey

Overall, I found Part 1...

	Strongly disagree	Somewhat disagree	Neither agree nor disagree	Somewhat agree	Strongly agree
enjoyable	○	○	○	○	○
easy	○	○	○	○	○
helpful in understanding custom blocks	○	○	○	○	○

Is there anything else that you would like to share with us about your experience with Part 1 ?

Part2-Programming

Visit https://q4blocks.appspot.com/?taskid=${e://Field/taskId} to start **Part 2.**
Upon completion, please enter Part 2's completion code for the programming task in the text box below. You will upload your completed program next.

Note: If you were not able to complete this programming part, please enter **TERMINATE** (all capital letters) for the programming part's completion code to end this survey.

TERMINATE2

Is there anything else that you would like to share with us about this research study? Please let us know how we can improve our study.

You will receive a survey completion code to enter into the MTurk HIT assignment in the next page. Since you did not fully complete Part 2, you will receive a bonus amount between $0-$4 based on how much you have completed.

SurveyCode

Your completion code is **${e://Field/randomNum}**

Please use this code to put in the MTurk HIT assignment.

DO NOT forget to click Submit or your response may not be recorded.

Part2-Programming

Please upload your completed Part 2 program. From the Scratch editor, click File > Save to your computer.

Part2-Post Survey1

The following questions are about your experiences with **learning to extract custom block** using the ${e://Field/first} and ${e://Field/second} methods.

For each method, please indicate how you feel about the following statements.
(1=Strongly Disagree, 2=Somewhat Disagree, 3=Neither Agree Nor Disagree, 4=Somewhat Agree, 5=Strongly Agree)

	${e://Field/first}					${e://Field/second}				
	1	2	3	4	5	1	2	3	4	5
enjoyable	○	○	○	○	○	○	○	○	○	○
easy	○	○	○	○	○	○	○	○	○	○
hard at first, but easier later	○	○	○	○	○	○	○	○	○	○
helpful in understanding custom blocks	○	○	○	○	○	○	○	○	○	○

Did you find it helpful to learn Extract Custom Block using the ${e://Field/first} method before learning the ${e://Field/second} method?

○ Yes ○ No

Please briefly explain your choice above (required).

Which method do you prefer when you want to extract custom block to improve your code?

○ Manual
○ Code Wizard

Please briefly explain your choice above (required).

I found code duplication...

	Strongly disagree	Somewhat disagree	Neither agree nor disagree	Somewhat agree	Strongly agree
easy to identify on my own	○	○	○	○	○

	Strongly disagree	Somewhat disagree	Neither agree nor disagree	Somewhat agree	Strongly agree
something I was aware of while coding	○	○	○	○	○

Part 2 Post Survey2

Overall, Part 2 helped me understand why...

	Strongly disagree	Somewhat disagree	Neither agree nor disagree	Somewhat agree	Strongly agree
duplication can make code hard to understand	○	○	○	○	○
duplication can make code hard to modify	○	○	○	○	○
extract custom block can make code easy to modify	○	○	○	○	○
extract custom block can make code easy to understand	○	○	○	○	○

Is there anything else that you would like to share with us about your experience with Part 2 ?
(e.g, suggestions to improve the tutorial, the code wizard tool and its hints, etc.)

Powered by Qualtrics