Paper ID #29813

Teaching the Culture of Quality from the Ground Up: Novice-Tailored Quality Improvement for Scratch Programmers

Dr. Eli Tilevich, Virginia Tech

Eli Tilevich is an Associate Professor in the Dept. of Computer Science and the College of Engineering Faculty Fellow at Virginia Tech. Tilevich's research interests lie on the Systems end of Software Engineering, with a particular emphasis on distributed systems, mobile/IoT applications, middleware, software energy efficiency, software security, automated program transformation, music informatics, and CS education. He has published over 100 refereed research papers on these subjects. His research awards include a Microsoft Research Software Engineering Innovation Foundation Award and an IBM Faculty Award. Tilevich has earned a B.A. summa cum laude in Computer Science/Math from Pace University, an M.S. in Information Systems from NYU, and a Ph.D. in Computer Science from Georgia Tech. At Virginia Tech, Tilevich leads the Software Innovations lab. The lab's research projects have been supported by major US federal funding agencies (i.e., NSF, ONR, AFOSR) and private industry. Tilevich is also a professionally trained classical clarinetist, with experience in orchestral, chamber, and solo performances.

Dr. Simin Hall, Virginia Tech

Dr. Simin Hall is a research consultant. Her projects include collaborating with Dr. Eli Tilevich in the Computer Science Department at Virginia Tech (VT) on a National Science Foundation IUSE funded project to improve quality in Block Based programming. Previously, she was a Research Assistant professor in the Department of Mechanical Engineering (ME) at VT. This year she is serving as an AAAS Science & Technology Policy Fellow. Her applied research in education is in cognitive functioning using online learning technologies. She maintains research and publishing tracks in nascent interdisciplinary trust concepts, eLearning, and innovative teaching, learning in fields of statistics and research methods, engineering, medical fields, and assessment methods.

Mr. Peeratham Techapalokul, Virginia Tech

Peeratham Techapalokul is a Ph.D. candidate in the Department of Computer Science at Virginia Tech. His research interests lie on visual programming languages and computer science education.

©American Society for Engineering Education, 2020

Teaching the Culture of Quality from the Ground Up: Novice-Tailored Quality Improvement for Scratch Programmers

Abstract

As quality problems plague the modern society's software infrastructure, a fundamental learning objective of computing education has become developing students' attitudes, knowledge, and practices centered around software quality. Teaching software quality and its disciplined practices has thus far been limited to more advanced courses, due to the prevailing assumptions about the introductory learner's unpreparedness for the topic and potential negative impacts on learner motivation. In this paper, we present empirical evidence that starkly contradicts the established conventional belief. Specifically, by exploring how novice programmers learn to refactor code duplication with and without automated tools, we found strong evidence that novices grasp the importance of code quality and its improvement. This empirical evidence motivated us, in retrospect, to closely examine the design of our online interactive tutorial, a platform that drove our experimental user study. We identify and discuss the tutorial's key design principles and affirm their efficacy based on the observed learning experiences. The obtained insights can inform curricular interventions that introduce introductory students to code quality and its disciplined improvement practices.

1 Introduction

The CS Education research literature has established the importance of teaching software quality as part of the CS curriculum[1,2,3,4]. 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 CS curricula[5,6,7,8].

Our goal is to help settle this controversial debate with a systematic empirical study that explores the impact of introducing novices to code quality and its improvement. Although our study cannot fully resolve this contentious issue of introductory computing pedagogy, we gained new important insights. Specifically, we focused on evaluating the impact of automated tools on helping beginner Scratch programmers to learn how to refactor code duplication. Refactoring is a software development technique that transforms a program to improve its code quality while preserving its behavior[9]. Although industry practitioners have fully embraced this technique as part of their professional toolset, refactoring is taught only late in the traditional computing curriculum.

A valuable outcome of our study was an educational intervention that really pushes the bound-

aries of what is possible to teach to novice programmers, those who have never had any prior programming experience. The unique aspect of our study was teaching the very fundamentals of programming simultaneously with the principles and mechanics of refactoring and automated refactoring support required to remove code duplication. In particular, the study participants went through a learning experience, guided by an online interactive tutorial that taught them EXTRACT CUSTOM BLOCK[1], the refactoring transformation that replaces duplicate code snippets with calls to a custom block. In Scratch, the custom block construct reifies procedural abstraction; it mimics the functionality of a procedure in text-based languages, having a unique name, parameters, and a body of statements. We discovered that not only could absolute beginners be effectively taught a real software quality concept (i.e., code duplication), but they were quite receptive and appreciative of this knowledge. The majority of the study participants showed an inclination to keep using automated quality improvement tools in their future programming pursuits. This discovery is an important contribution to the understanding of the mindset of introductory learners, with respect to their attitude toward not only learning how to write code, but how to do it well.

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 tutorial and its instructional strategies can serve as a helpful guide that informs future curricular interventions for novice programmers.

In this paper, we describe the aforementioned tutorial and how its design was instrumental to the success of our experimental study. We begin by discussing the most closely related examples of CS education research concerned with code quality. In the remainder of the paper, we identify the underlying design principles of our online interactive tutorial. We discuss the analysis results that affirm the tutorial's efficacy based on participant's learning experiences, extracted from the collected log data and the administered learning experience survey. Finally we highlight the key implications and recommendations that can inform curricular interventions that integrate software quality into introductory CS curricula.

2 Related Work

Recent CS education research in code quality identifies the growing need to treat the topic as an important part of the CS curriculum. Several recent studies uncover not only that student programs are rife with quality issues, but also that students remain largely unaware of the importance of code quality and its improvement. Keuning et al. identified several code quality issues in a large dataset of student-authored Java programs, as well as the undisciplined practices that led to these issues[3]. Through interviews, Börstler et al. uncovered that students possess a low degree of skills and knowledge about code quality, concluding that code quality should be discussed more in CS education programs[1]. Other works focus on developing guidelines that help educators give code

[1]In text-based programming languages, this refactoring is known as Extract Method

improvement feedback to students[4,2].

In the context of Scratch, the early works that explore code quality focus on identifying code quality problems[10,11,12,13] and applying program analysis tools to help educators better understand the quality of programs written by novice programmers[14,15,16]. These code quality tools are intended mostly for educators as an aid in grading student programs or providing informal feedback. In contrast, this work studies an intervention whose primary target are introductory students in need of effectively learning programming fundamentals while embracing disciplined software development practices.

Several recent works explore how software engineering principles and practices that promote code quality can be integrated into the introductory CS curriculum. Hermans and Aivaloglou conclude that it is feasible and useful to teach K-12 students software engineering principles and practices, including code duplication[17]. When taught 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[18]. The findings of these prior studies inspired our work to investigate into the efficacy of teaching refactoring, an advanced software development technique, to novice programmers.

A few other works focus on educational tools that help novice Scratch programmers improve code quality. 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[19]. In our prior work, we added automated refactoring to Scratch and studied whether automated support motivates novice programmers to improve code quality[20]. This work builds on these preliminary results with the goal of identifying the underlying design principles of our tutorial, an educational intervention that we empirically showed effective in teaching novice programmers the importance of code quality and its improvement practices.

3 Tutorial: Refactoring Code Duplication without Any Programming Experience

In this section, we begin by describing our tutorial, the platform we used in the experimental user study that explores the impact of automated tools on novice programmers learning how to refactor code duplication. Then, we discuss the underlying design principles that were identified as having most impacted the tutorial's learning efficacy.

3.1 Content Structure

Figure 1 depicts the tutorial's Part 1 and Part 2. Because the study participants had no prior programming experience, Part 1 introduced them to the basics of Scratch programming and procedural abstraction. In classroom use, students already familiar with Scratch and the covered content may safely skip this part. Then, Part 2 introduced the participants to code duplication and the EXTRACT CUSTOM BLOCK refactoring. To clearly specify its objectives, each part includes an animation of a correctly completed program's intended behavior. Structured as a deck of learning cards, each

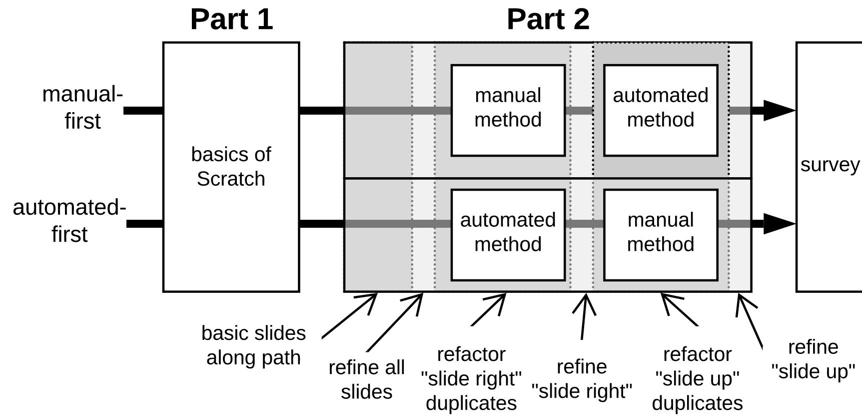

Figure 1: Tutorial's overall flow

containing step-by-step instructions, the resulting deck remained visible within our customized Scratch programming environment for users to follow as they complete tutorial tasks. Because the tutorial steps build on each other, the participants had to complete each step correctly before proceeding to the next one.

The participants in both groups (MANUAL-FIRST and AUTOMATED-FIRST) learned how to refactor by using both manual and automated methods but in different order. However, in this paper, rather than focusing on ascertaining how automated tools impact learning experiences, we instead analyze which aspects of the tutorial's design made it possible for novice programmers to learn basic programming skills alongside real-world code quality issues and improvement.

Next, we describe each part of the tutorial in detail.

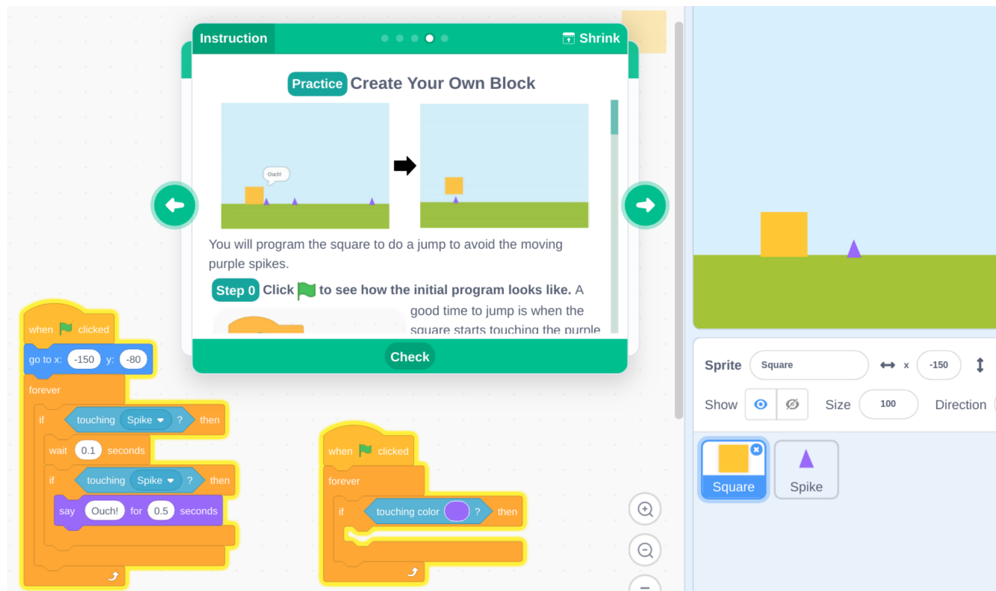

Figure 2: Screenshot of tutorial's Part 1

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 helps learners 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 2). The tutorial introduces custom blocks by following the recommendations from the Creative Computing Curriculum[21], a Scratch instructor's guide developed by the Harvard Graduate School of Education. Specifically, 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.

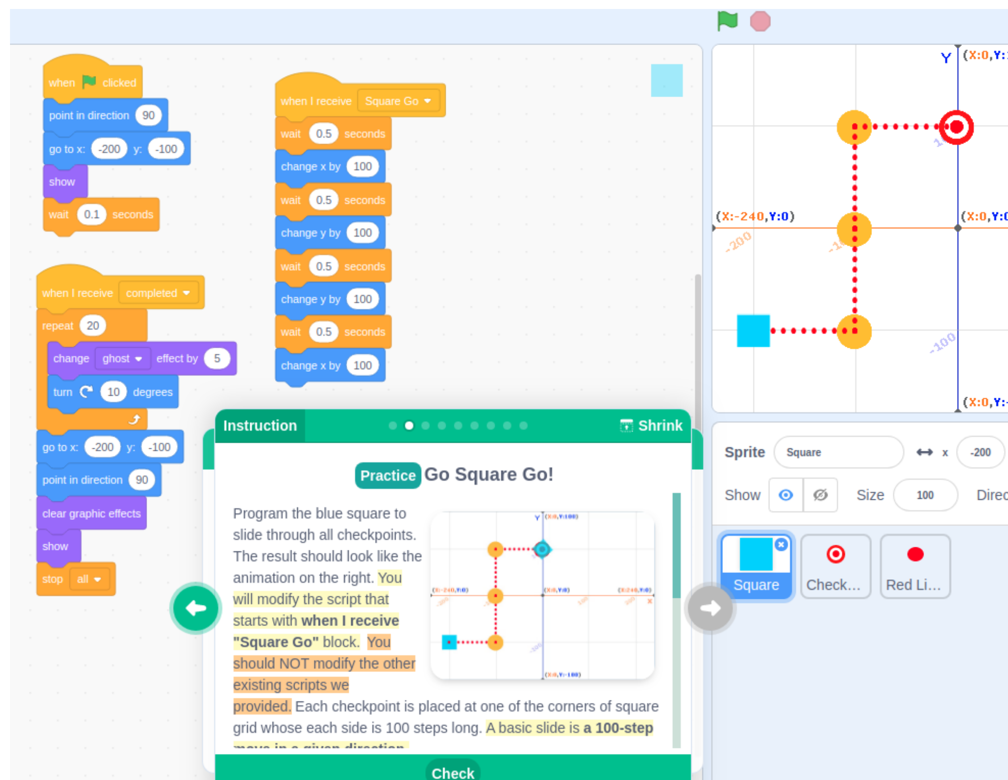

Figure 3: Screenshot of tutorial's Part 2; The skeleton project was scaffolded to help learners visualize the program behavior when working on *basic-slides* task.

Part 2: This part guides learners to complete its objective by situating them in the scenario that demands frequent code modifications, typical for the iterative software development process[22]. Learners first complete the missing program part to make a character move within a grid of squares (see Figure 3). The instructions specify "slide", as a special movement of 100 units that moves the character from one square to the next one, 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 the specified path: "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

Figure 4: Screenshots of instruction cards for different tasks: refine all slides (top), refactor *slide right* duplicates (bottom left), and refine *slide right* (bottom right)

to understand and modify (see Figure 4, top screenshot). Then, it introduces learners how to carry out the refactoring manually (see Figure 4, bottom left screenshot) and with automated tools. After being introduced to each method, learners are instructed to refine the recently refactored sliding implementation (see Figure 4, bottom right screenshot), as a way to demonstrate the beneficial impact of custom blocks on code quality.

The tutorial explains how to carry out the EXTRACT CUSTOM BLOCK refactoring by hand by following these steps: (1) identify the duplicate code parts, (2) create a custom block—a descriptively named procedure with an empty body, (3) define the custom block in the procedure's body by making a copy of one of the duplicate parts to serve as the definition, and (4) replace the duplicate parts with the calls to the newly created custom block.

When learning to refactor with the automated tool, all the participants have to do is to select an automatically detected duplicate code segments, as highlighted by the tool and click the "Extract" button, thus automatically transforming the selected segments into calls to a custom block.

3.2 Design Principles

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 motivate 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[23]. By following this rudimentary example, novice programmers acquire sufficient knowledge and skills to apply this basic refactoring as part of their programming process.

   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[24,25].

2. *Engaging the learner:* For learners to become fully engaged with the task at hand, they need to understand the task and the subject computer code, used as an example for learning how to refactor (the "sliding" code containing duplications). As opposed to displaying the subject worked example 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 sustain learner motivation 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 the details of 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 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. 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[26], allowing learners to check their own progress at a designated level of proficiency impacts motivation positively. Constructivist theory[27,28] 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[29]. Situating learners in a real development context provides a unique opportunity to convey 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.

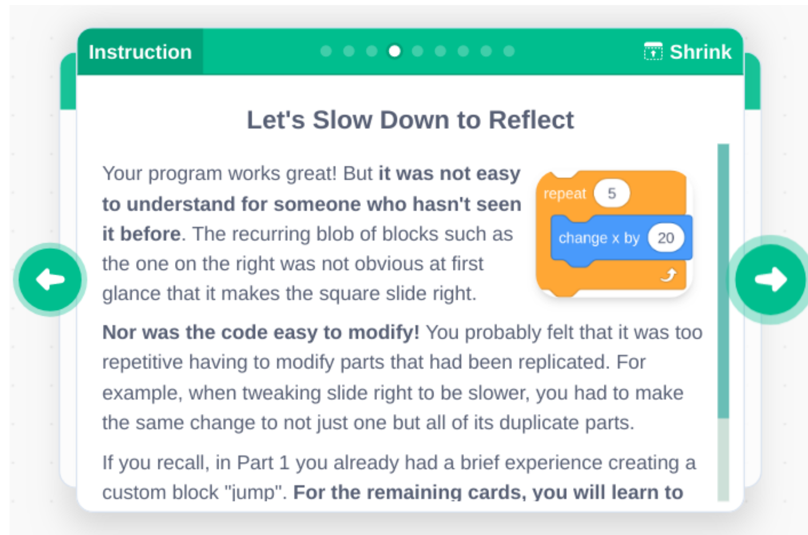

Figure 5: Tutorial's Part 2: A short note to help learners reflect on code quality

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 presented with a short note that guided them to reflect on their experience of modifying and understanding duplicate code (see Figure 5) before learning refactoring.

5. *Scaffolding the learning:* The tutorial loads up a skeletal Scratch project, 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 the program's output remaining 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.

4 Method

We follow a mixed-method design. Specifically, we conducted an experimental study that teaches novice programmers how to refactor code duplication. This study explores if the availability of automated refactoring tool increases the effectiveness of novice programmers learning the subject. 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 a programming tutorial and a subsequent survey.

The collected evaluation data comprises: (1) learning performance data, based on the time taken by each participant to complete each tutorial task and (2) survey questionnaire responses, which reflect each participant's self-reported learning experience.

In our study, both groups (MANUAL-FIRST and AUTOMATED-FIRST) experienced the same experimental conditions but in different order. Despite this difference in conditions, the tutorial impacted each group's learning experience similarly, when accounted for the automated tool's learning effect. In the following discussion, for brevity, we present the statistical results computed from the data collected from the participants in the MANUAL-FIRST group only. However, when it comes to open-ended responses, we present and carefully examine those of both groups to better understand how the participants perceived the tutorial.

Survey Data: The survey data comprises the participants' self-reported responses: the agreement rating 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). Additionally, the participants were asked a few open-ended questions about their perception of the tutorial and suggestions for improvement.

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

1. enjoyable
2. easy
3. 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

5 Results and Discussion

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 analyses and their results and discuss how they ascertain the tutorial's efficacy.

5.1 Log data

Figure 6 shows the distributions of time data of 12 participants in the MANUAL-FIRST group for each tutorial's task. Several interesting observations can be made about how long the participants

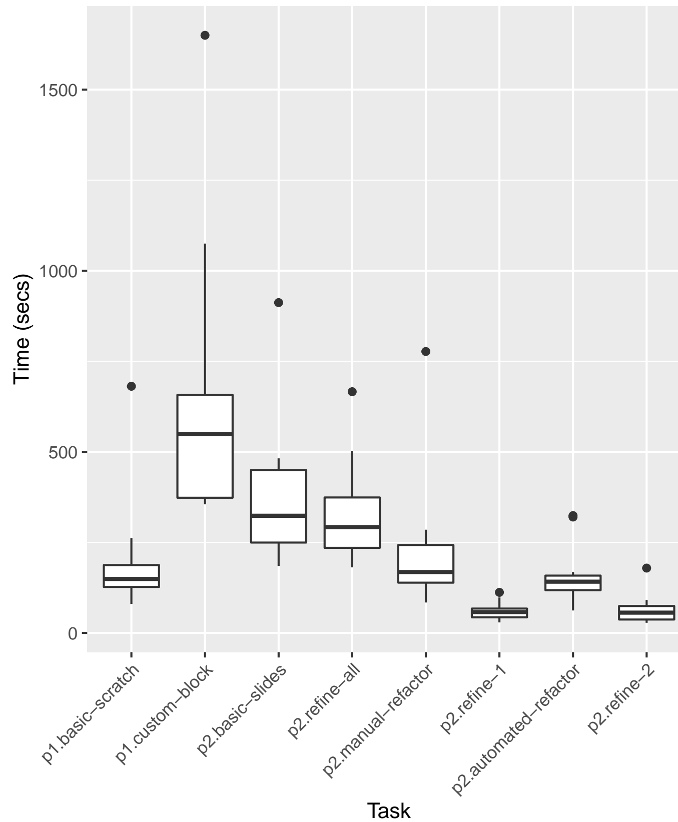

Figure 6: Distributions of time spent on each tutorial's task ordered chronologically

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.

Overall, 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).

5.2 Survey Questionnaire

Learning motivation and engagement: Figure 7 visualizes the raw results of the participants' level of agreement with statements A1-3 for the tutorial's Part 1 (the basics of Scratch and custom blocks,

top figure) and Part 2 (code duplication and EXTRACT CUSTOM BLOCK refactoring, bottom figure). Their perception about the difficulty level differs noticeably: 83% of the participants found Part 1 easy, while only 50% found Part 2 to be so as well. All participants either somewhat agreed or strongly agreed that they found Part 1 enjoyable. Similarly, a high percentage (92%) of the participants either somewhat agreed or strongly agreed when asked to rate the same agreement for Part 2. 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 motivating novice programmers to learn how to refactor code duplication.

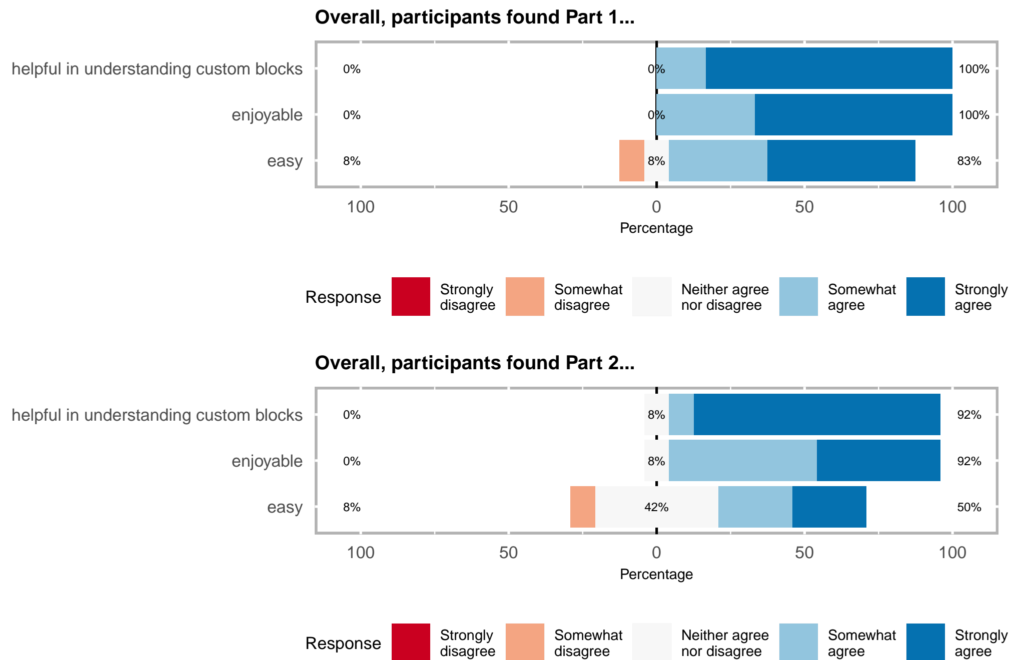

Figure 7: Participants' self-reported learning experience for Part 1 and Part 2 of the tutorial

Code quality perception: Figure 8 visualizes the raw results of the participants' level of agreement about how useful they found the tutorial in helping them understand the importance of code quality and its improvement practices. The majority of the participants agreed to the statement 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). However, they 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). The results show that the tutorial was more effective in conveying the benefits of code improvement practices but less so in the more subjective matters of the quality problems making code hard to understand and modify.

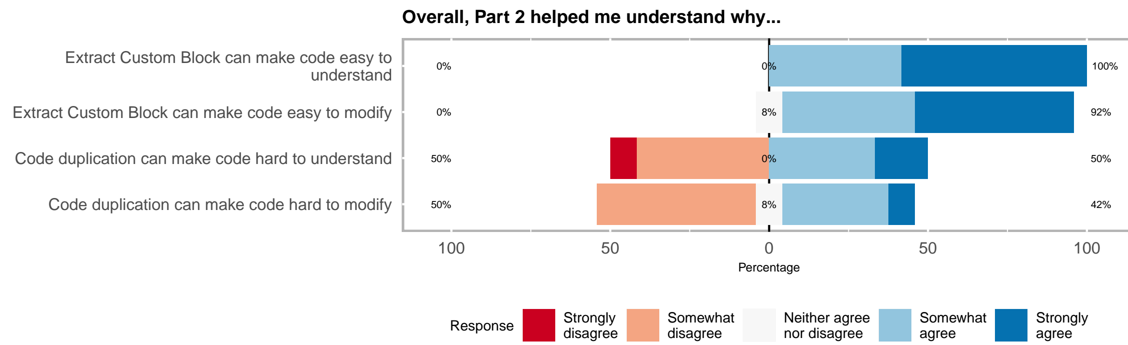

Figure 8: Participants' perception of the tutorial's efficacy in demonstrating the importance of code quality and its improvement practices

Part 1's 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 learning experience: Several participants reported positive learning experiences. 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.

Overall, the observed learning experiences demonstrate the tutorial's learning efficacy for teaching 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 are effective in engaging and motivating 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 recognized the importance of code quality and its improvement.

6 Implications and Recommendations

Novice-level code quality principles and improvement practices: 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.

Integrating code quality topics in introductory curricula: Our evaluation results clearly show that it is feasible and useful to teach code quality 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.

An 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 taught to novice programmers and how to design the required interventions.

Our findings demonstrate that a properly designed self-paced interactive tutorial can motivate and 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[30], 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.

Placing the teaching of code quality in the context of Constructivist-based learning: 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"[31], 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[32]. 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.)

7 Conclusion

The motivation of this work stems from a phenomenon we observed investigating how automated tools impact the learning effectiveness in the context of teaching refactoring to novice programmers. Specifically, we observed that novice programmers were quite receptive to the importance of code quality and its improvement, contradicting the conventional belief that the topic is inappropriate for introductory learners. This positive learning outcome led us to revisit a more fundamental question of what the contributing factors to our tutorial's effectiveness are, as an educational intervention intended for novice learners.

In this paper, we have identified several key design principles that contribute to the tutorial's efficacy. Altogether, these design factors situate students in a simple but relevant code quality problem and an improvement scenario, with a learning scaffolding enhancing their engagement and motivation. By analyzing the results of this learning experience on the sample of 24 novice programmers, we further ascertain and explain the efficacy of our approach in reaching the following objectives: motivate and engage novice programmers to learn how to refactor code duplication, while helping them develop an appreciation for code quality and its systematic improvement. Our findings can help inform not only similar curricular interventions, but also the design of introductory computing curriculum that integrates the topics of software quality and other fundamental principles and practices in software engineering.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable feedback that helped improve this manuscript. This research is supported by the National Science Foundation through the Grant DUE-1712131.

References

[1] 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*, pages 70–85, 2018.

[2] 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.

[3] 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*, pages 110–115, 2017.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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*, ITiCSE '11, pages 168–172, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0697-3. doi: 10.1145/1999747.1999796. URL http://doi.acm.org/10.1145/1999747.1999796.

[11] 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. doi: 10.1109/fie.2014.7044055.

[12] 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.

[13] 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.

[14] 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.

[15] Jesús Moreno-León and Gregorio Robles. Dr. Scratch: A web tool to automatically evaluate Scratch projects. In *Proceedings of the Workshop in Primary and Secondary Computing Education*, WiPSCE '15, pages 132–133, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3753-3. doi: 10.1145/2818314.2818338. URL http://doi.acm.org/10.1145/2818314.2818338.

[16] 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.

[17] 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. ISBN 978-1-5386-2671-9. doi: 10.1109/ICSE-SEET.2017.13.

[18] 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. ISBN 9781450356602. doi: 10.1145/3183377.3183396. URL https://doi.org/10.1145/3183377.3183396.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

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

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

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

[28] 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.

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

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

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

[32] 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.