An Exploratory Study of the Remixing Practices in the Scratch Programming Community: Trends, Causalities, and Influences

Prapti P. Khawas

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

Master of Science
in
Computer Science and Applications

Eli Tilevich, Chair
Clifford A. Shaffer
Na Meng

May 10, 2019
Blacksburg, Virginia

Keywords: Scratch, block-based programming, remixing, program analysis, code quality, introductory computing education.

An Exploratory Study of the Remixing Practices in the Scratch Programming Community: Trends, Causalities, and Influences

Prapti P. Khawas

(ABSTRACT)

One of the greatest achievements of Scratch as an educational tool is the eager willingness of programmers to use existing projects as the starting point for their own projects, a practice known as *remixing*. Despite the importance of remixing as a foundation of collaborative and communal learning, the practice remains poorly understood. Without a clear picture of how and why Scratch programmers remix a project as a starting point of their own projects, this programming community would remain in the dark about which programming practices encourage and facilitate remixing. The designers of programming environments for blocks lack feedback on how the remixing facility is used in the wild. To gain a deeper insight into remixing, this thesis presents the results of a comprehensive study of this practice in Scratch that investigates the following heretofore unexplored dimensions of remixing: (1) the prevailing modifications that remixes perform on existing projects, (2) the impact of the original project's code quality on the granularity, extent, and development time of the modifications in the remixes, and (3) the propensity of the dominant programming practices in the original project to remain so in the remixes. Our findings can be used to promote those programming practices in the Scratch community that encourage remixing while also improving this practice's effectiveness, thus benefiting the educational and end-user programming communities.

An Exploratory Study of the Remixing Practices in the Scratch Programming Community: Trends, Causalities, and Influences

Prapti P. Khawas

(GENERAL AUDIENCE ABSTRACT)

The Scratch programming language has become an intrinsically important tool in introductory CS education. A visual, block-based language, Scratch is web-based, featuring an enormous online programming community, through which projects are eagerly shared. One of the unique learning provisions of Scratch is the ability to easily start a project by modifying someone else's project, a practice referred to as *remixing*. Despite the central role that remixing plays in enabling the communal and collaborative learning styles in the Scratch community, the practice of remixing remains inadequately understood. This knowledge gap leaves the Scratch community in the dark about which programming practices encourage and facilitate remixing, as well as deprives Scratch environment designers from actionable feedback on how the remixing facility is used in the wild. To address this problem, this thesis reports on the results of an exploratory study of remixing in Scratch that investigates three heretofore unexplored dimensions of this practice. First, we study the general remixing trends in terms of how remixes modify the original projects. Second, we infer the impact of a project's code quality on the modifications in its remixes and the development time. Finally, we investigate whether programmers adopt the techniques and practices of the remixed projects. Computing educators can apply our findings to enhance the educational effectiveness of Scratch by encouraging the practice and magnitude of remixing.

Dedication

To my family for their unrelenting faith in me.

Acknowledgments

I would like to start by thanking my advisor, Dr. Eli Tilevich, for his invaluable support and guidance towards completing my thesis and for helping me write the paper this thesis is based on and submitting it to the conference. Thank you for all of your time, effort and humorous conversations during our writing review sessions. I would also like to thank the members of my committee, Dr. Cliff Shaffer and Dr. Na Meng, for their valuable suggestions. I would also like to thank the Department of Computer Science at Virginia Tech for everything I have learned during my master's study, and for the support of my master's education with assistantship.

I would like to express my gratefulness and gratitude to Peeratham (Karn) Techpalokul for providing the direction for my thesis and his timely suggestions and feedback. My wonderful friends who have laughed, cried and supported me throughout - Palakh, Bipasha, Niti and Supritha. Especially Tushar for being my rock.

Last, but not the least, my family without whom I would have never been able to pursue further studies.[1]

[1]This research is supported in part by the National Science Foundation through the grant DUE #1712131.

Contents

1 Introduction **1**

1.1 Remixing in Scratch 3

1.2 Research Questions 5

2 Literature Review **7**

2.1 Programmer Behavior in Block-Based Programming Environments 7

2.2 Remixing in Scratch 8

2.3 Assessing Code Readability Based on Quality Metrics 9

2.4 Code Smells in Scratch 11

2.5 Code Difference Detection 12

3 Background **14**

3.1 Scratch Environment 14

3.2 Code Quality Metrics 15

3.2.1 Program size/LOC 15

3.2.2 Halstead's Metrics 15

3.2.3 Cyclomatic complexity 16

3.2.4 ABC Metric 16

3.3 Code Smells in Scratch 17

4 Introducing Remixing **18**

4.1 Project #1: "Parachute Flight" 18

4.1.1 Remix #1: "Parachute Robbery" 19

4.1.2 Remix #2: "ola00x93" 20

4.1.3 Remix #3: "Parachute Flight Remix" 23

4.2 Project #2: "Scrolling Slopeformer" 23

5 Methodology **28**

5.1 Project Collection 28

5.2 Data Preprocessing 30

5.3 Detecting Code Changes 31

5.4 Measuring Modifications 31

5.5 General remixing trends (RQ1) 33

5.6 Impact of Code Quality on Remixing Modifications (RQ2) 35

5.7 Learning from Original Projects (RQ3) 36

6 Results **38**

6.1 General Remixing Trends (RQ1) 39

6.2 Impact of Code Quality on Remixing Modifications (RQ2) 42

6.3 Learning from Original Projects (RQ3) 45

7 Threats to validity **48**

7.1 External Validity 48

7.2 Internal Validity 49

8 Discussion **50**

8.1 General Remixing Trends (RQ1) 50

8.2 Code Quality and Remixing Modifications (RQ2) 51

8.3 Learning from Original Projects (RQ3) 52

9 Future Work **53**

10 Conclusions **55**

Bibliography **56**

Appendices **62**

Appendix A Scratch Project Structure **63**

Appendix B MongoDB Queries **64**

List of Figures

1.1 Scratch 3.0 Editor 4

1.2 Remix tree of a sample project[2] 4

4.1 Project page of “Parachute Flight” 19

4.2 Sprite1 in ‘Parachute Flight’ 20

4.3 Duplicate Sprites found in the Remix 21

4.4 Re-positioning a script in the remix 22

4.5 Modifications to costume in the remix 24

4.6 Addition of scripts in the remix 25

4.7 Project page of “Scrolling Slopeformer” 26

4.8 Renaming a sprite in the remix 26

4.9 A long script in the project 27

5.1 Analysis Infrastructure 29

5.2 Scratch Explore page 30

6.1 Proportion of block types altered 40

6.2 Change in ABC metric over time 42

6.3 Code quality metrics of original projects in three categories of remixes based on modification percentage 44

6.4 Code quality metrics of original projects in three categories of remixes based on modification time 46

A.1 Scratch Project Structure 63

List of Tables

6.1 Summary statistics of 8142 projects 39

6.2 Mean modification values in 5,384 remixes (excluding unmodified remixes) . 40

6.3 Assessing change of the code quality in the remixes with paired t-tests . . . 41

6.4 Comparing mean code quality based on modification % 43

6.5 Comparing mean code quality based on modification time 45

List of Abbreviations

ABC Assignment-Branch-Conditional

API Application Programming Interface

AST Abstract Syntax Tree

CT Computational Thinking

JSON JavaScript Object Notation

LOC Lines Of Code

Chapter 1

Introduction

Many beginner programmers find traditional coding overwhelming, as they have to contend with the syntax and semantics of text-based programming languages. With visual, block-based programming languages, including App Inventor and Scratch, novice programmers can gently ramp up to the full complexity of writing code in a text-based language [WW17]. Blocks in block-based environments are designed and developed to fit syntactically. Thus, novice programmers are encouraged to continue coding when they see their code execute rather than having to fix errors even for the trivial "Hello World" program in a text-based language.

Launched in May 2007, Scratch has become a popular block-based programming environment for young learners for many years now. Figure 1.1 shows one-window user interface of the Scratch environment. With a prominent command palette, a central scripting area, no syntax or runtime errors, and a surprisingly small number of commands along with a loosely coupled inter-object communication system, Scratch fosters programming, creativity, collaboration, and code sharing [Res+09]. Undoubtedly, it does help young programmers to lay the foundations required to write code in more advanced programming languages in the future [Das+16]. Projects on Scratch vary from games, animations, art projects to tutorials that encourage learners with different interests to participate in basic programming. Based on the constructionism theory of Seymour Papert [Pap87], Scratch encourages users to share

their projects and promotes learning through discussions in a community. Papert defined *constructionism* in a National Science Foundation proposal titled "Constructionism: A New Opportunity for Elementary Science Education" [Pap87] as follows:

"The word constructionism is a mnemonic for two aspects of the theory of science education underlying this project. From constructivist theories of psychology, we take a view of learning as a reconstruction rather than as a transmission of knowledge. Then we extend the idea of manipulative materials to the idea that learning is most effective when part of an activity the learner experiences as constructing a meaningful product."

Thus, the four aspects of constructionism—learning through activities of designing, personalizing, sharing, and reflecting —have become the foci in Scratch [Res+09]. The name Scratch is inspired by "Scratching," a DJ or turntablist method of remixing different songs to create a new one which is very similar to the environment allowing users to share their projects and create their personalized projects, called Remixes, using other's shared projects.

Intrigued by the success of Scratch, the research community has been actively exploring the codebase of the Scratch open-source repository as well as studying the programming practices of Scratch programmers. Aivaloglou and Hermans examined Scratch programming patterns in terms of program complexity, coding practices, and code smells [AH16]. Weintrop and Wilensky discovered that when it comes to learning computational concepts, students find it more expeditious and engaging to use a block-based language than a text-based one [WW17]. Another study investigated various projects to understand how programmers code and define various tinkering techniques in yet another block-based programming environment of Snap! [Don+19].

1.1 Remixing in Scratch

The success of Scratch as an educational platform is nothing short of amazing. The public repository of Scratch projects contains close to 40 million projects written by computing learners of all ages coming from all over the world[1]. Although it would be hard to pinpoint a single reason for this success, and most likely it is a combination of factors, one of the design features of the Scratch blocks editor—"Remix[this project]"—immediately comes to mind. With a single click of a button, a Scratch programmer can clone any existing project, with the clone becoming the start of the development process. The product, also called a *remix*, is a modified and shared version of an uploaded project. This feature shares similarities with the development processed fostered by open source communities like GitHub or Bitbucket, in which developers publish the source code of their projects and fork other developers' projects. This allows beginners to experience development by playing the role of a contributor to open source repositories. The ability to extend or modify someone else's work with ease must have a tremendous positive influence on the productivity and level of satisfaction of Scratch programmers [Res+09]. The number of remixes of a project also becomes a point of pride for their creators. The practice of remixing serves as an important community building activity for Scratch programmers [Res+09]. Not only beginner programmers get up to speed quicker by remixing an existing project, but they also learn programming idioms more effectively, adopting the techniques and styles of often more experienced peers who authored the remixed project.

Papert's constructionism theory, introduced above, informs and encourages a free sharing and remixing of projects. Remixing is what allows beginner programmers to quickly learn from, experiment with, and add to existing Scratch projects. Figure 1.2 shows a varied

[1] https://scratch.mit.edu/statistics/
[2] https://scratch.mit.edu/projects/273402679/remixtree/

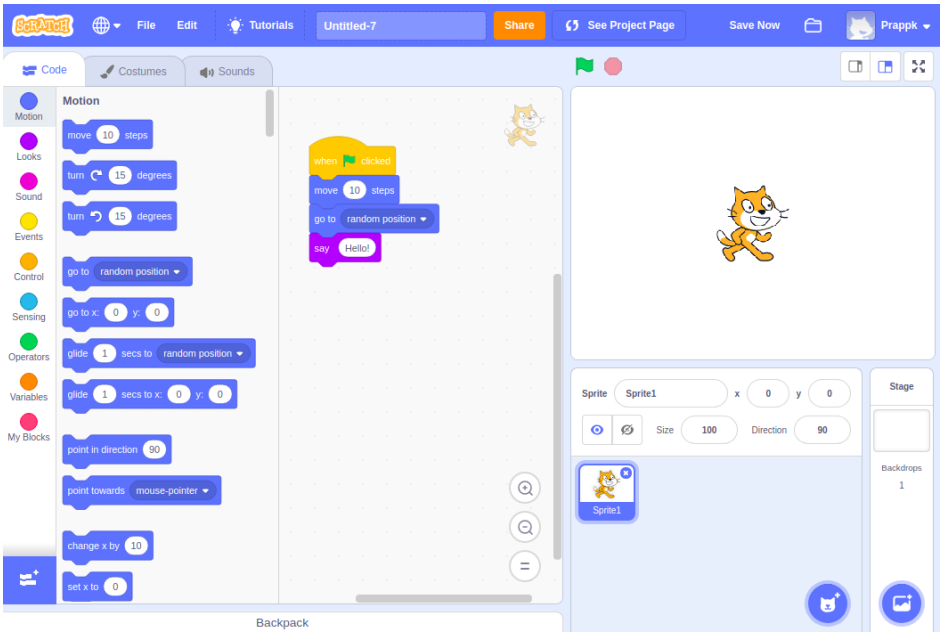


Figure 1.1: Scratch 3.0 Editor

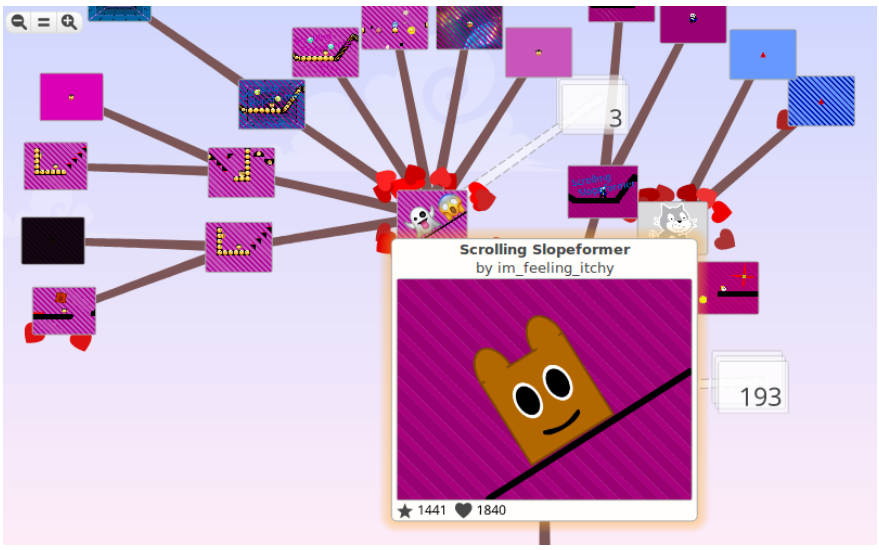


Figure 1.2: Remix tree of a sample project[2]

collection of remixes portraying how Scratch programmers tweak an original project. Not surprisingly, remixing has been a target of several prior studies. Dasgupta et al. have studied the effectiveness of Scratch as a way of building computational knowledge through remixing or appropriation of code [Das+16]. However, Hermans and Aivaloglou have established the prevalence of code smells in Scratch projects and the possibility of poor code quality to play a role in inhibiting remixing [HA16]. Yet another study by the same authors explored the relationship between the popularity as a measure of quality of remixes and their program size [HM13]. The specific modifications in remixes have been studied from different perspectives. Hill and Monroy-Hernández measured modification in remixes as edit distances [HM15], whereas Techapalokul and Tilevich computed modifications as the script addition metric [TT17]. Our work builds on these prior studies, to explore the practice of remixing from a three-dimensional perspective that combines the nature of modifications, the impact of code quality, and the rate of adoption of programming practices.

1.2 Research Questions

To truly understand how computing learners take advantage of remixing requires a comprehensive exploration. Our goal is not only to increase the likelihood of positive learning outcomes, but also to help improve the design of the language and its programming environment. By conducting our study, we seek answers to the following research questions:

- **RQ1:** How do programmers remix Scratch projects?

  Specifically, we investigate how programmers modify various Scratch language constructs in the original project. We also measure if the original and remixed projects differ in terms of the code quality.

- **RQ2:** How does the code quality of a project impact the granularity, volume and development time for the modifications made in its remixes?

  We assess the relationship between the original project's various code quality metrics and code smells with the type of modifications and time to modify its remixes. Our goal is to uncover the impact of a Scratch project's code quality and the nature of the modifications in its remixes. The obtained insights may be transferable from Scratch to the practices associated with professional open source software repositories, such as GitHub.

- **RQ3:** Do the prevailing coding idioms of the original project influence their use in the remixes?

  As noted by Cheliotis et al., "a remix of a single creative work cannot escape the 'aura' of the source" [Che+14]. We evaluate how a programming style, defined in terms of using certain advanced programming idioms, affect the degree of learning for beginner Scratch programmers.

Contributions and Novelty

In this study, we collect a set of 160 representative Scratch projects with 15,010 remixes. We systematically pre-process and analyze this dataset to answer the research questions above. This thesis contributes a targeted retrospective inquiry into the practice of remixing in Scratch. Among the novel aspects of our inquiry is a focus on the observed modifications in the remixes as a function of the original project's quality. Another novel aspect is an assessment of the influence of the coding practices in the original projects on those in their remixes.

Chapter 2

Literature Review

The success of Scratch has driven many researchers to study various aspects of Scratch—both qualitatively and quantitatively. However, an ever-increasing collection of research efforts studying Scratch artifacts lacks a comprehensive multidimensional study on remixing practices. In this section, we discuss related prior work that motivated various aspects of our analysis. We also describe previous studies on remixes and discuss how their work differs from ours.

2.1 Programmer Behavior in Block-Based Programming Environments

Weintrop and Wilensky studied the difference in programmer's attitudes, perceptions, and conceptual learning with text-based and block-based programming on Pencil.cc [WW17]. Based on their findings, students find learning computational concepts more expeditious and engaging with a block-based language rather than a text-based one. Another study investigated various projects to understand how programmers code in yet another block-based programming environment of Snap! [Don+19]. The authors categorized tinkering behaviour into three types: (1) Test-based, (2) Prototype-based and (3) Construction-based tinkering. We expect to observe the prototype-based tinkering behavior in our study in

the remixes, which the authors identified as expanding existing projects, that is, making modifications to the original projects by adding or removing features in the remixes; and working on side-projects, that is, making modifications in the remixes to an extent that it fails to resemble the original project.

Aivaloglou and Hermans conducted a large-scale analysis of Scratch projects in terms of program complexity, coding practices, and code smells [AH16]. Their study reported that most projects are small in size, i.e., less than 100 blocks. In addition, custom blocks ("My Blocks") are the least used blocks on Scratch. This finding lead us to examine the adoption of advanced programming concepts, like procedures, from the original projects in their remixes. Another study by Meerbaum-Salant, Armoni, and Ben-Ari, on familiarity of computer science concepts with Scratch, identified variables and broadcast-receive blocks as challenging for students [MAB13] which we consider as advanced Scratch concepts for our analysis.

2.2 Remixing in Scratch

The practice of remixing has been previously studied by Dasgupta et al. in the context of the development of computational thinking (CT) skills [Das+16]. The authors assess the relationship between programming proficiency and the presence of remixes in the projects shared by a programmer. Their findings indicate that programmers who remix more often use a larger range of Scratch blocks, thus display their superior understanding of CT concepts. Our work differs in our focus on examining the adoption of advanced programming idioms, the issues of software quality, and their impact on remixes.

Another study by Hill and Monroy-Hernández evaluated peer-ratings as a measure of quality of de novo projects and remixes [HM13] using fitted logistic regression models. They discovered that the quality of remixes is generally lower than de novo projects. In our anal-

ysis, we perform a similar comparison between quality of an original project and its remixes. However, we use software code quality metrics to assess the quality of the project.

In addition, the specific modifications in remixes have been studied from different perspectives. Hill and Monroy-Hernández measured modification in remixes as edit distances [HM15], whereas Techapalokul and Tilevich computed modifications as the script addition metric [TT17]. For our analysis, we use the number of modifications at statement level to quantify modifications made in the remix.

2.3 Assessing Code Readability Based on Quality Metrics

Before programmers can extend someone else's code, they have to be able to understand it, and code quality is known to be positively correlated with software comprehensibility. In the context of text-based languages and professional software development, prior research has applied various code quality metrics to assess the comprehensibility of a software system [LW06] [LW08]. Previously Moreno-León, Robles, and Román-González used cyclomatic complexity, and Halstead's vocabulary and length metrics to evaluate the quality of Scratch projects. Nevertheless, to the best of our knowledge, ours is the first work that systematically assesses the relationship between software quality and remixing, across the original projects and their remixes.

Some of the software quality metrics that we used in our analyses are as follows:

Program Size/LOC

One of the most widely used and easy-to-understand metrics is Lines Of Code (LOC). Based on observations from estimating a simpler model for evaluating readability, Posnett, Hindle, and Devanbu suggested that program size alone may have no impact on reported readability scores [PHD11]. Nevertheless, as the mean size of Scratch projects was found to be around 140-150 LOC [AH16] [TT17], we posit that beginner programmers are likely to find it overwhelming having to understand someone else's projects that are too large in size.

Halstead's Metrics

Prior work to study the impact of Halstead's metrics on readability have confirmed that volume metric indeed impacts readability. Alawad, Panta, and Zibran examined the correlation between two quality metrics, volume metric being one of the two, and discovered that the volume quality metric is negatively correlated with six readability metrics [APZ18]. Another study discovered that although one cannot evaluate readability by adding program difficulty or effort to the logistic regression model, volume does affect readability [PHD11]. Hence, we select the volume metric out of the six Halstead's metrics for our analysis.

Cyclomatic Complexity

Kasto and Whalley conducted a study on the responses of students in a first-year computing course examination. They examined the correlation between cyclomatic complexity and how well students comprehend code, and observe that this metric is highly correlated with the level of difficulty the code presents to students [KW13]. However, another study reports that the correlation between cyclomatic complexity and readability may not be as strong

[BW10]. Despite this finding, we use cyclomatic complexity to assess code quality of Scratch projects as it is still a reliable measure of program complexity and has been used in various research studies on Scratch.

ABC Metric

Though the ABC metric has not been studied in the context of readability and understanding, it reliably estimates code complexity [KS16]. This metric has not been used for Scratch programs, however, it is widely incorporated in professional software quality measuring tools such as Code Climate[1], GMetrics[2] etc.

2.4 Code Smells in Scratch

As defined by Kent Beck in Fowler's book on refactoring, a code smell is a surface indication that usually corresponds to a deeper problem in the system [Fow99].

Although Scratch promotes informal exploratory programming practices, the issue of code quality has been discovered to impact the educational effectiveness of this programming environment. In a prior work, Techapalokul and Tilevich cataloged 12 commonly occurring smells in Scratch projects [TT17]. Long Script, Uncommunicative Name, Duplicate Code, and Broad Variable Scope were found to be most prevalent. In addition, the presence of code smells in a project has been found to reduce the probability that other programmers would be willing to remix that project and the projects highly afflicted by smells were found to have fewer script additions than projects of higher software quality. Another study by Hermans and Aivaloglou demonstrated that presence of code smells make Scratch programs hard to

[1] https://codeclimate.com/
[2] https://dx42.github.io/gmetrics/

understand and modify for novice programmers [HA16]. They observed that *Long Script* can hinder code understanding, while *Duplicate Code* can make code difficult to maintain. Hence, we concentrate on two highly prevalent smells—*Long Script* and *Duplicate Code*—in our study.

2.5 Code Difference Detection

Various techniques have been developed to identify differences in two text-based software programs by measuring code similarity in them. These similarity measuring techniques can be categorized based on metrics-based, text-based, token-based, tree-based, and graph-based approaches. Most of the early approaches used software metrics, like Halstead's metrics, to evaluate similarity in two code snippets [Ott76] [FR87]. However, these metrics have found to be ineffective in correctly determining similarity as discovered by Kapser and Godfrey [KG03]. Text-based approaches use string based similarity evaluation which can be done in two different ways: (1) Ranking and (2) Fingerprinting. Ranking involves measuring text similarity between each pair of documents, sorting them based on the scores and selecting the pair with maximum similarity score [BTZ07]. Fingerprinting, on the other hand, uses hashing to identify each document with a collection of integers as key aspects of the document [SWA03]. Programs can be abstracted as a collection of tokens which are used in the token-based approaches [KKI02] [ĐG12]. Tree-based approach involves converting programs into Abstract Syntax Trees (AST) and comparing their sub-trees [JSC07]. Program Dependence Graphs (PDG), or Control Flow Graphs (CFG) are used in graph-based approaches [Kri01] [Cha+13]. However, both tree-based and graph-based approaches have been found to have high computation time due to comparison of highly complex structures [Bel+07]. For our study, we detect differences in projects inspired by the ranking technique for its simplicity,

suitability in the block-to-text context, and time and computation efficiency.

Chapter 3

Background

To better understand our study, we present relevant background on Scratch and techniques used in our analysis.

3.1 Scratch Environment

Developed by MIT Media Lab, Scratch is a block-based programming language and environment. We describe the relevant elements of a project in Scratch used in our study. A Scratch project comprises of a stage and sprites. **Sprite** is an object or character which users can program. **Stage** is a special sprite representing the background layer. **Costume** is an appearance of a sprite and each sprite must have at least one costume. **Sound** is an audio that can be played while the program is running. **Block** is a unit of Scratch project that programs the sprites. Scratch has a broad range of blocks which are color coded by Motion, Looks, Sound, List, Event, Control, Sensing, Operators, Variables and My Block categories. Motion and Looks blocks change the attributes of the sprite. Broadcast-receive blocks handle intercommunication between sprites. They can be used to branch out the flow of script execution. **Script** is a consecutive stack of blocks interlocked with each other. **Variable** is a unit of Scratch project that holds data values. Variables can be global, which means they can be used across sprites, or local, which means they can be used only in the sprite they are defined in.

3.2 Code Quality Metrics

Some of the software quality metrics that we used in our analyses are as follows:

3.2.1 Program size/LOC

One of the most widely used and easy-to-understand metrics is Lines Of Code (LOC). It is a software metric used to measure the size of a computer program by counting the number of lines in the text of the program's source code.

3.2.2 Halstead's Metrics

One of the most widely accepted methods for evaluation software complexity is the Halstead's metrics. Specifically, Halstead proposed a collection of metrics that are based on the presence of certain program tokens (i.e., operators and operands) [Hal77]. From this collection, we use the *volume* metric defined as:

$$\text{Volume: } V = N \times \log_2 \eta \tag{3.1}$$

$$\text{Program vocabulary: } \eta = \eta_1 + \eta_2 \tag{3.2}$$

$$\text{Program length: } N = N_1 + N_2 \tag{3.3}$$

$$\text{Calculated program length: } \hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 \tag{3.4}$$

$$\text{Difficulty: } D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2} \tag{3.5}$$

$$\text{Effort: } E = D \times V \tag{3.6}$$

where η_1 and η_2 is the number of distinct operators and operands respectively; and N_1 and

N_2 is the total number of operators and operands respectively.

3.2.3 Cyclomatic complexity

Another method for reasoning about a codebases' complexity is the cyclomatic complexity metrics. This metric is derived from the codebase's control flow graph, in which nodes represent basic blocks and a directed edge expresses that control flows from the source block to the destination block [McC76]. Mathematically, the cyclomatic complexity M is defined as

$$M = E - N + 2P \tag{3.7}$$

where, E is the number of edges in the graph, N is the number of nodes in the graph, and P is the number of connected components.

3.2.4 ABC Metric

Yet another method for expressing the complexity of a codebase is the ABC metric, introduced by Fitzpatrick as a triplet of values that comprehensively measure the size of a program [Fit96]. Specifically, the ABC metric combines the total number of

- Assignments (A)
- Branches (B)
- Conditionals (C)

in a program. Software tools represent this metric either as a 3-D vector <A, B, C> or as a scalar value

$$| < ABCvector > | = \sqrt{(A^2 + B^2 + C^2)} \tag{3.8}$$

3.3 Code Smells in Scratch

The code smells analyzed in our study are defined [TT17] below:

1. **Duplicate Code:**

   A segment of code is duplicated at multiple locations in the program as a way to reuse existing functionality.

2. **Long Script:**

   A script is considered unreasonably long if there are more than 11 [TT17] consecutive blocks measured vertically.

Chapter 4

Introducing Remixing

To explain how Scratch programmers remix, we next present two representative projects, selected from the 'Explore' page in the Scratch programming environment. We first present the original projects and then discuss some of their remixes. We also point out some of the discoveries that motivated the research directions of this thesis.

4.1 Project #1: "Parachute Flight"

Figure 4.1 shows the project page of the first example project, entitled "Parachute Flight"[1]. This project is a game, in which the user uses the direction keys to control a parachute to avoid randomly appearing birds and collect coins. The game's goal is to safely reach the ground and to collect as many coins as possible. This project turned out to be quite popular, as evidenced by the number of times it has been marked as 'loved' (i.e., the heart icon) and 'favorite' (i.e., the star icon). Not surprisingly, this project has been remixed heavily—as many as 492 times—at the time of this writing. We manually examine 3 random remixes, and in the following sections report what we have observed.

[1] https://scratch.mit.edu/projects/276366240/

Figure 4.1: Project page of “Parachute Flight”

4.1.1 Remix #1: “Parachute Robbery”

In the “Parachute Robbery” remix[2], we discover that the programmer has duplicated a sprite 75 times, creating that many copies of the original sprite’s code. That level of duplication is unexpected, as the original project contains a “When I start as a clone” block that makes it possible to clone the sprite in the game rather than copying it so many times. An Object-Oriented programming analogy would be instantiating a class multiple times (i.e., cloning) vs. copying the class’s code and then instantiating each copy once (i.e., copying). This finding inspires a research question of whether the presence of a “When I start as a clone” block in the original project impacts the introduction of duplicate sprites in the project’s remixes. Based on our preliminary analysis, we find that close to 28% of the remixes contain duplicate sprites.

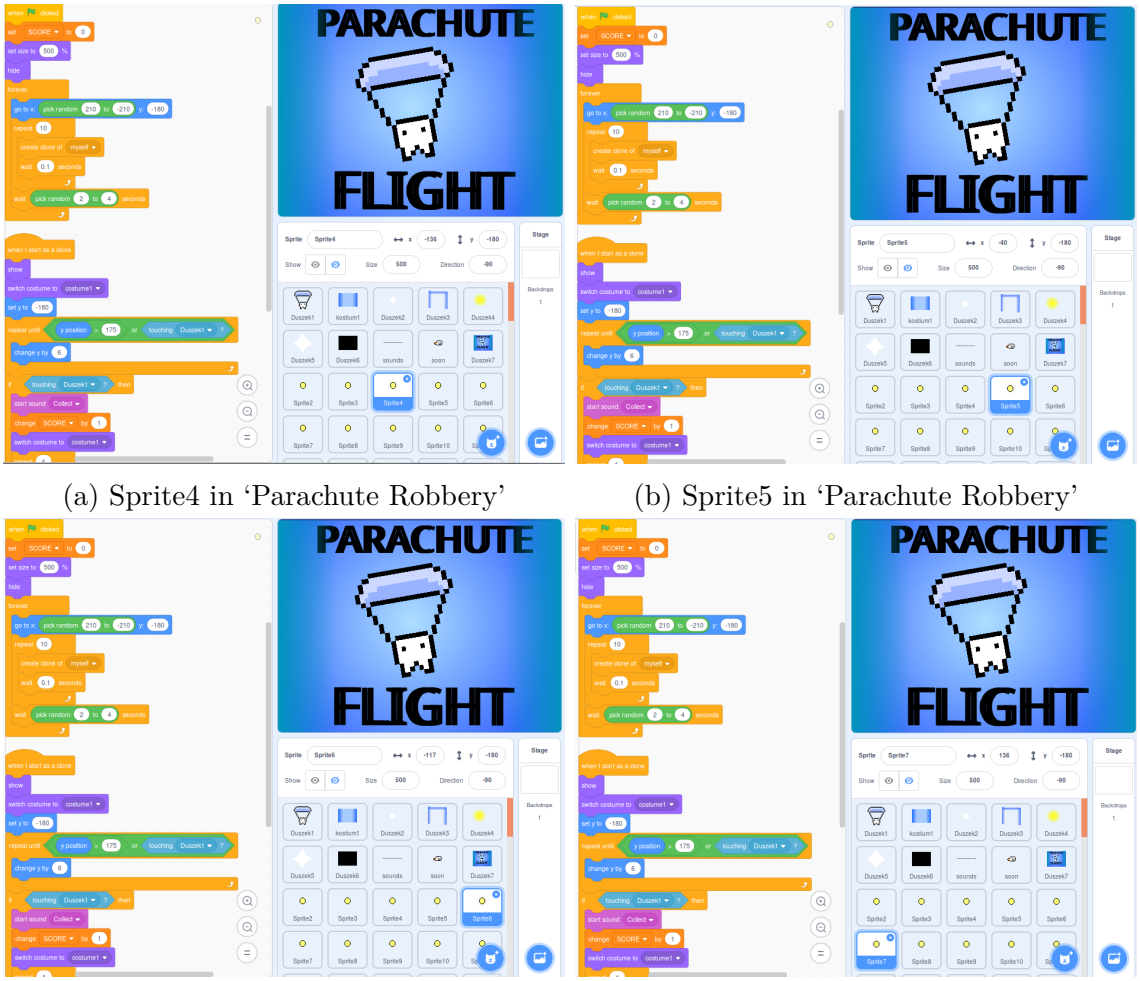
[2] https://scratch.mit.edu/projects/280764565/

Figure 4.2: Sprite1 in 'Parachute Flight'

4.1.2 Remix #2: "ola00x93"

In the "ola00x93" remix[3], the programmer has made minimal modifications to the project. What seems unusual is that in in this remix only one of the scripts has been re-positioned, as shown in Figure 4.4. That is, the remix is almost an identical copy of the original project with only one of the constituent scripts appearing on a different location in the workspace. This observation raises a research question about the number and magnitude of modifications in the remixes.

[3] https://scratch.mit.edu/projects/279813881/

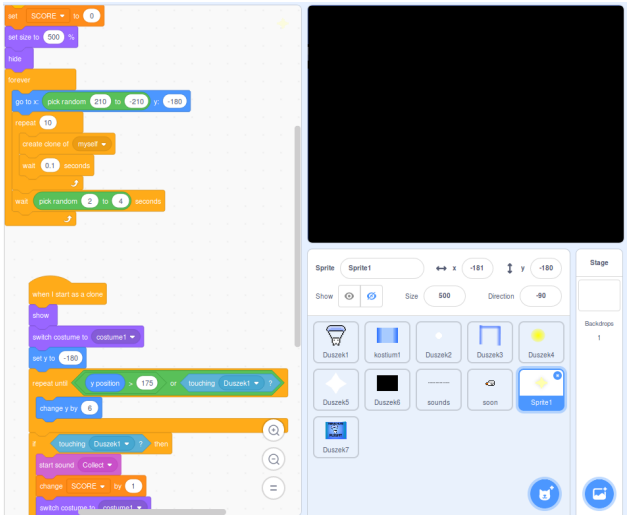


(a) Sprite4 in ‘Parachute Robbery’ (b) Sprite5 in ‘Parachute Robbery’

(c) Sprite6 in ‘Parachute Robbery’ (d) Sprite7 in ‘Parachute Robbery’

Figure 4.3: Duplicate Sprites found in the Remix

(a) ‘Duszek1’ in original project



(b) ‘Duszek1’ in remix

Figure 4.4: Re-positioning a script in the remix

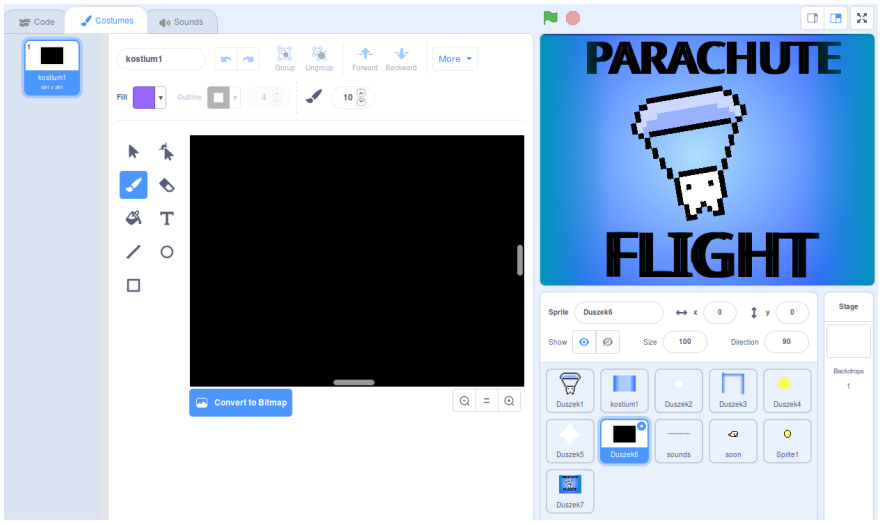
4.1.3 Remix #3: “Parachute Flight Remix”

In the “Parachute Flight Remix” remix[4], we examine the modifications that the programmer made to the costumes of the sprite as shown in Figure 4.5. What we discover is that the sprite’s source code has not been changed at all. Figure 4.6 shows how scripts are added to another sprite in this remix. The fact of scripts being freely added raises a research question of how the code quality of a project affects how much new code is added to its remixes.

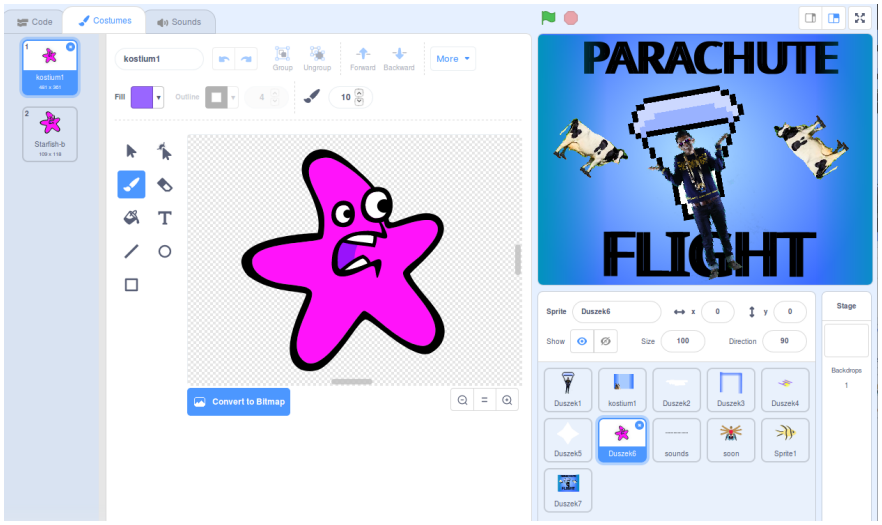
4.2 Project #2: “Scrolling Slopeformer”

Figure 4.7 shows the project page of our second example project, entitled “Scrolling Slopeformer”[5]. This project is also a game, in which the user moves a character across a slope by pressing the direction keys with the goal of avoiding randomly appearing obstacles. This project has been remixed 185 times. We find that most of modifications made in the remixes are small in size. Figure 4.8 shows an example modification in one of the remixes; the programmer only renamed a sprite, without changing any code. This observation inspires a research question about the impact of a project’s code quality and the nature of modifications in its remixes. We conjecture that the low size of modifications in this project’s remixes may be due to the presence of some long scripts in the original project, as shown in Figure 4.9.

[4] https://scratch.mit.edu/projects/281462571/
[5] https://scratch.mit.edu/projects/273402679/

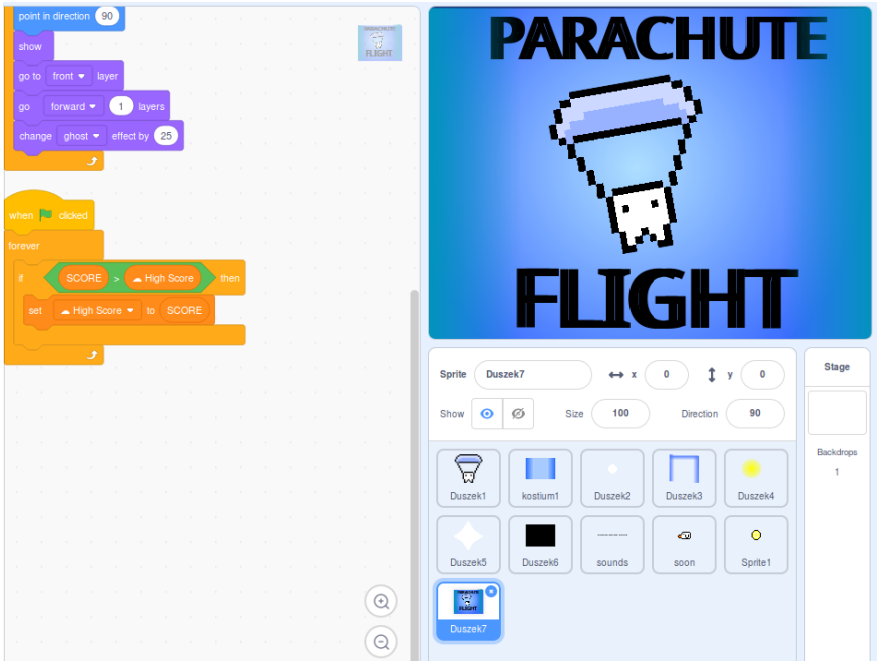


(a) Costume for ‘Duszek6’ in original project

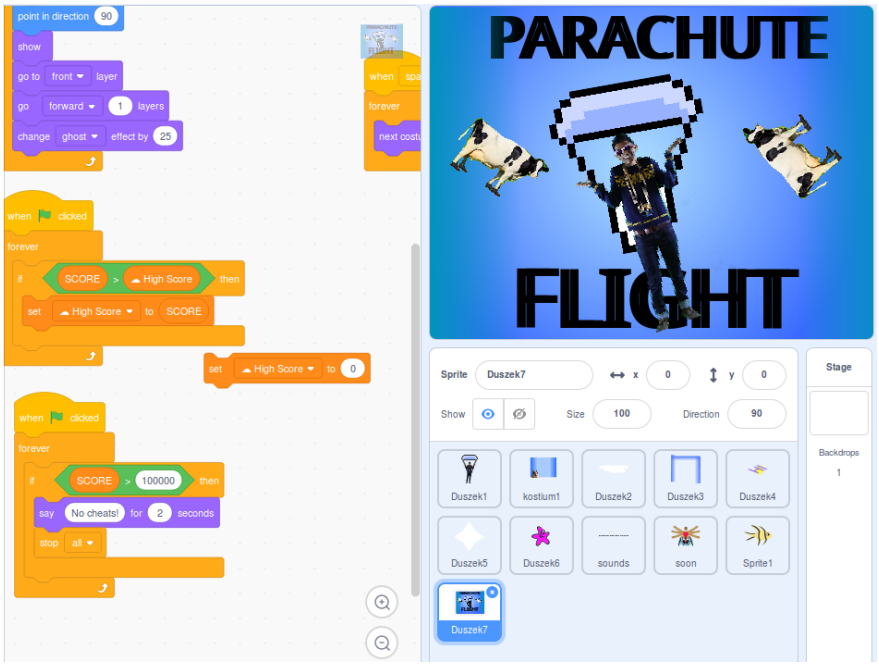


(b) Costumes for ‘Duszek6’ in remix

Figure 4.5: Modifications to costume in the remix



(a) ‘Duszek7’ in original project



(b) ‘Duszek7’ in remix

Figure 4.6: Addition of scripts in the remix

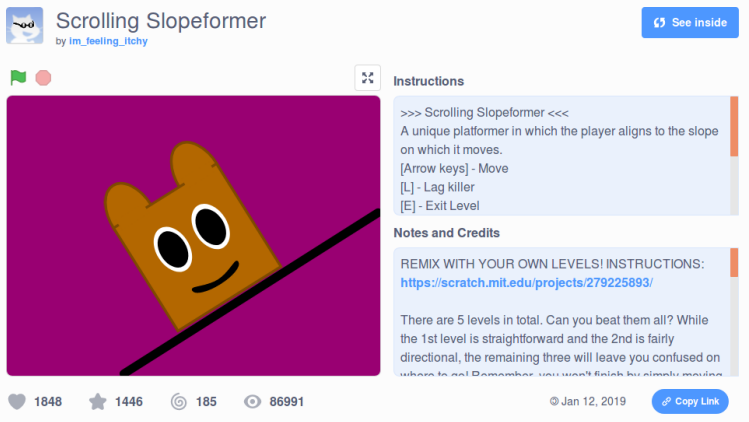
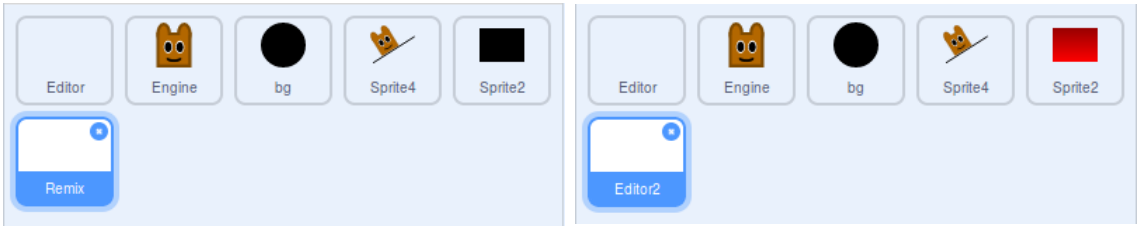


Figure 4.7: Project page of “Scrolling Slopeformer”



(a) Sprites in ‘Scrolling Slopeformer’ (b) Sprites in ‘Scrolling Slopeformer remix’

Figure 4.8: Renaming a sprite in the remix

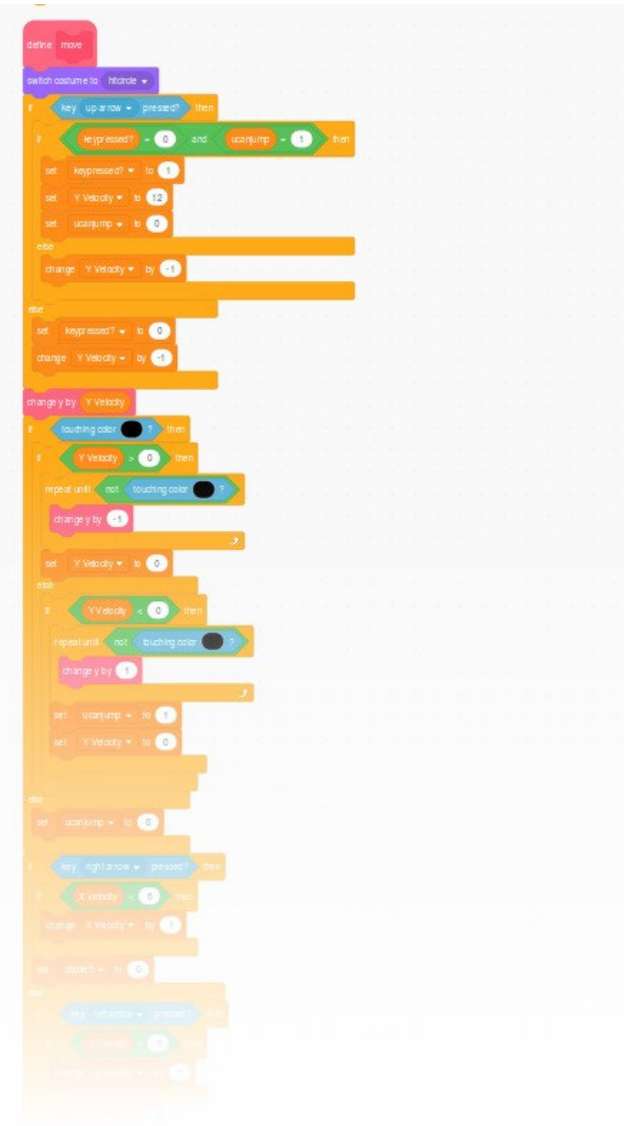


Figure 4.9: A long script in the project

Chapter 5

Methodology

In this section, we describe our data collection and pre-processing procedures as well as how we identify and record program modifications. Then, we present the research questions this work aims to answer. Finally, we explain the methods used to answer these questions. Figure 5.1 provides a high level overview of our analysis infrastructure.

5.1 Project Collection

In Scratch, the landing page contains the sections "Featured Projects," "Featured Studios," "What the Community is Remixing (Top Remixed),", and "What the Community is Loving (Top Loved)"—each of which displaying a collection of 20 projects. For our study, we chose not to use any projects from the "Top Remixed" section, as they tend to be mostly remix chains or coloring contests. To get a more extensive and varied project collection, we fetch 160 trending projects via the API[1] that is used to display them on the "Explore" page of the Scratch environment (Figure 5.2).

The Scratch projects in our experimental dataset have been remixed between 1 to 700 times. We retrieve the remix list for each project by using the API[2] that returns project remix trees. We collect only the first-level remixes; that is, the remixes that are the immediate children

[1] https://api.scratch.mit.edu/explore/projects?limit=<max_limit=40>&offset=0&language=en&mode=trending&q=*

[2] https://scratch.mit.edu/projects/<project_id>/remixtree/bare/

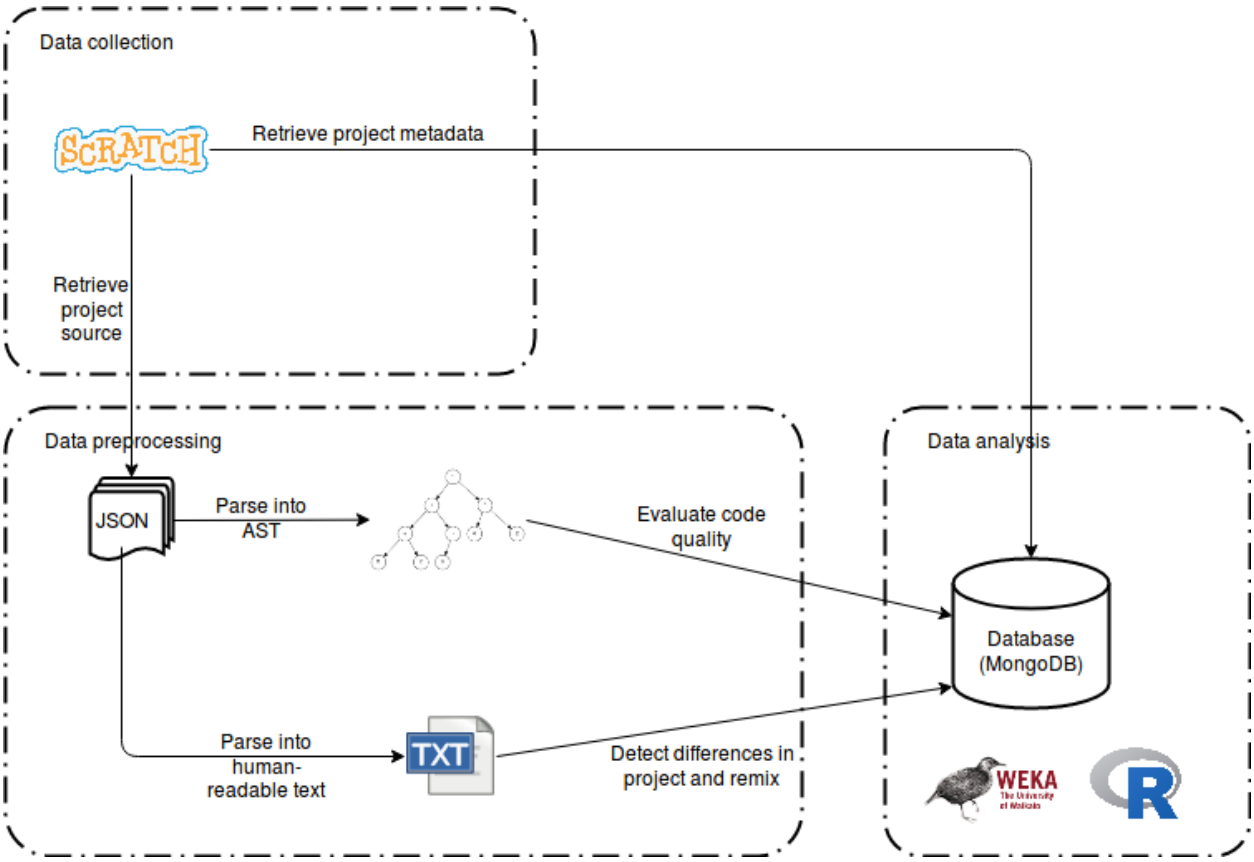


Figure 5.1: Analysis Infrastructure

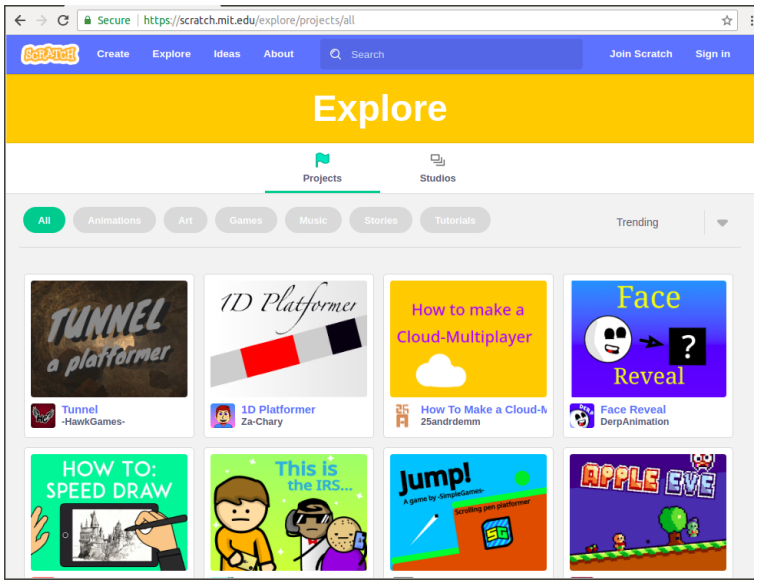


Figure 5.2: Scratch Explore page

of one of the 160 subject projects. As a result, we put together a representative collection of 15,010 Scratch projects for our study. The source code of each project was retrieved in JSON format using the following API[3].

5.2 Data Preprocessing

For each project, we parse its source in JSON format into the Abstract Syntax Tree (AST) representation that is more easily amenable for computing quality measures. To detect the differences between the original project and its remixes, we convert the AST to a human-readable string to be able to apply text-similarity techniques as described in the next section. We collect the statistics on various types of modification. 8% of the remixes ended up missing values, and as such were removed from the experimental dataset. We also disregard outliers from our dataset using interquartile range. 1% of the remaining remixes ended up being modified more than a 100%, indicating that the remix is a brand new project that shares no

[3] https://projects.scratch.mit.edu/<project_id>

similarity with the original project. We conduct our analysis with and without these projects, as they can be associated with side-projects [Don+19], in which the remixes diverge from the original project too much to skew our remixes-specific findings.

5.3 Detecting Code Changes

To identify modifications made in the remixes, we systematically compare human-readable strings of the original project and its remix as inspired by winnowing, a fingerprinting algorithm [SWA03]. We start by dissecting both projects into stage and sprite sections and comparing the stage sections of the two projects. Then, we match sprites from the original project and its remix which is done in two steps. This is done to make sure that the correct corresponding sprites are compared even if the sprites were re-arranged in the remix. First, we match sprites using their names. Next, we match the remaining sprites from both projects based on their textual similarity using the cosine metric. Unmatched sprites from the original project are counted as deleted sprites in the remix and similarly, unmatched sprites from the remix are counted as newly added sprites. We further inspect the matched sprites for any differences at statement level using java-diff-utils, a DiffUtils Java library. Algorithm 5.1 describes this process systematically.

5.4 Measuring Modifications

Scratch projects can be remixed by adding, deleting, or replacing various program elements as well as by moving scripts around within the workspace, similarly to rearranging code in text-based programs. We refer to *modification size* as the total number of blocks in a remix that are added, deleted, or replaced as compared to the original project's script. While

Algorithm 5.1 Algorithm to detect modifications in Scratch programs

```
1: global variables
2:     modificationMap, Global map to store various modifications found in remix
3: end global variables
4: originalTargets ← Divide original Scratch program into stage and sprites
5: remixTargets ← Divide remix Scratch program into stage and sprites
6: for originalTarget in originalTargets do
7:     for remixTarget in remixTargets do
8:         if originalTarget.name = remixTarget.name then      ▷ Targets with same name
9:             ComputeDiff(originalTarget, remixTarget)
10:            originalTargets ← originalTargets \ originalTarget
11:            remixTargets ← remixTargets \ remixTarget
12:        end if
13:    end for
14: end for
15: for originalTarget from originalTargets do
16:    for remixTarget from remixTargets do
17:        if CosineSimilarity(originalTarget, remixTarget) > 0.85 then    ▷ Targets with
    similar code
18:            ComputeDiff(originalTarget, remixTarget)
19:            originalTargets ← originalTargets \ originalTarget
20:            remixTargets ← remixTargets \ remixTarget
21:        end if
22:    end for
23: end for
24: function ComputeDiff(target1, target2)
25:    deltas ← Compute difference between target1 and target2 using java-diff-utils
26:    for delta in deltas do
27:        checkforRearrangement(delta, modificationMap)        ▷ For scripts re-positioned
28:        checkForBlockTypes(delta, modificationMap)        ▷ For block insertion, deletion,
    modification and categories
29:        checkForSounds(delta, modificationMap)        ▷ For sound addition and deletion
30:        checkForCostumes(delta, modificationMap)  ▷ For costume addition and deletion
31:        checkForVariables(delta, modificationMap)        ▷ For global and local variable
    addition and deletion
32:    end for
33: end function
```

calculating the modification sizes of remixes, we noticed that the metric is often influenced by the size of the remixed projects. That is, large projects would be more likely to also have remixes with large modification sizes as compared to smaller projects. This insight suggests that across projects of different sizes, the magnitude of modifications can be more uniformly captured by reporting *modification percentages* rather than plain modification sizes. The *modification percent* of a remix is the percentage ratio of blocks in a remix that is added, deleted, or replaced in the original project's script divided by the total number of blocks in the original project.

$$\text{Modification percentage} = \frac{\text{Modification size}}{\text{Original project size}} \times 100 \tag{5.1}$$

Since we study the modifications that remixes make to the original projects, we disregard those blocks that are added to new sprites in the remixes, as these modifications are unrelated to the code quality of the remixed projects.

5.5 General remixing trends (RQ1)

An analysis of a recent snapshot of the Scratch repository reveals that about a third of all recently shared projects are remixes[4]. With remixing being a prominent activity in the Scratch community, our study aims at exploring various characteristics of remixes. Our exploration is guided by the following questions:

- **RQ1.1:** How do programmers modify the project elements of remixed projects?

  To understand the practice of remixing, we count the total numbers of insertion, deletion, and change operations performed on the constituent elements of the remixed

[4] http://scratch.mit.edu/statistics

Scratch projects: Sprites, Costume, Sound, Variables (Global/Local), and Blocks. In addition, we capture the script rearrangements, that is, when programmers reposition an existing script within the Scratch workspace, and block categories that are most frequently altered.

- **RQ1.2:** Does the code quality change between the original projects and their remixes?
  To assess the change in code quality for metrics discussed in Section 3.2, we conduct a paired t-test (two tailed) of the remixes. We formulate the null and alternate hypothesis as follows:

  H_0 There is no difference in code quality between the original projects and their remixes.

  H_1 There is some difference in code quality between the original projects and their remixes.

  If null hypothesis is rejected, we conduct lower-tailed t-tests to identify the increase/decrease in code quality.

- **RQ1.3:** What is the trend in the code quality of remixes once their original project is shared?
  To provide insights into the programming practices of remixing over time, we analyze the time-series of code quality in remixes with high modification rates.

5.6 Impact of Code Quality on Remixing Modifications (RQ2)

We examine the code quality of the remixed Scratch projects via two complementary approaches: (1) detecting the incidence of code smells, and (2) calculating various code quality metrics. Our experiments aim at answering the following questions:

- **RQ2.1:** Are poor code quality projects more likely to have remixes with smaller-sized modifications?
  We investigate the code quality of remixed projects for three categories of remixes based on the modification percentages: No modification (0%), Low modification (0-50%) and High modification (50-100%).

- **RQ2.2**: Does code quality affect the time it takes to modify a project in its remixes?
  We perform a similar examination for three categories of remixes based on the time taken to modify: No modification time (0 min), Low modification time (0 min - 3.6 days) and High modification time (3.6 - 66.8 days).

To identify the presence of code smells, we compute the following metrics:

1. **Long Script Density:** We calculate this metric as the ratio of long scripts (scripts containing blocks>11) to the total number of scripts.

2. **Duplicate Groups:** We count the total number of duplicate groups within a project.

To calculate the code quality of Scratch projects, we adapt the code quality metrics, discussed in Section 3.2 for the peculiarities of block based programming constructs.

1. **Halstead's metrics:** To measure code quality, we choose the *volume* metric, as it is known to be inversely correlated to program understandability. For Scratch, we consider operands to be variable blocks or shadow blocks, and operators to be all blocks other than variable blocks.

2. **ABC Metric:** For Scratch projects, we count the set and change blocks as assignments, the procedure calls and broadcast-receive blocks as branches; and the if-else blocks or blocks containing the operator blocks as conditionals. We apply the scalar value of the ABC metric to express code quality. The higher is the ABC value, the lower is the code quality.

5.7 Learning from Original Projects (RQ3)

The Scratch community fosters an environment in which novice programmers feel comfortable learning from each other. Thus, it is likely that existing Scratch projects do influence the programming practices of beginner programmers. To understand how original projects impacts their remixes, we conduct an in-depth analysis, with the goal of answering the following questions:

- **RQ3.1:** Does the presence of sprites in a project containing the "When I start as clone" block impact the number of duplicate sprites in its remixes?

  We investigate whether projects that contain "When I start as clone" have remixes containing duplicate sprites, as based on the number of cloneable sprites (sprites containing the "When I start as clone" block) in the original project and the number of duplicate sprites in its remixes.

- **RQ3.2:** Does the presence of procedures in the original project motivate their in-

creased use in its remixes?

We study original projects that contain procedures and examine whether new procedures are added to their remixes.

Chapter 6

Results

Table 6.1 presents an overview of the projects in our experimental dataset. Out of the initial dataset of 8,142 remixes, 2,758 were left completely unmodified from the original projects and 371 contained only addition or deletion of sprites without any modification to sprites in original projects. Most of these unmodified projects were remixes of projects on the higher end of the quality metric, that is, mean value of 4.251 for cyclomatic complexity and mean value of 336.004 for ABC metric value. This observation suggests that we may have correctly assumed that project complexity is negatively correlated with the degree and number of changes in its remixes. The projects sizes of the studied remixes ranged from 2 blocks to 9616 blocks. We partition these projects into three categories based on size, with 7920 projects in the small category (between 2 and 3207 blocks), as this size seems to be the most comfortable for programmers to remix, 151 project in the medium category (3207-6412 blocks), and 60 project in the large category (6412-9616 blocks), suggesting that large projects are hard to understand and modify. In the following discussion, we present and discuss the answers to the research questions posed.

Table 6.1: Summary statistics of 8142 projects

Statistic	Min	Max	Mean
Original Projects			
Number of remixes	1	696	85
Program size (# blocks)	13	12632	498.83
Cyclomatic complexity	2	38	4.888
Program Volume	39.069	3778.843	560.477
ABC Metric	0	10983.706	247.787
Remixes			
Program size (# blocks)	3	9616	516.898
Cyclomatic complexity	2	38	4.539
Program Volume	49.05	4424.014	804.881
ABC Metric	2	10983.706	320.679
Modification percent	0	890.458	4.88

6.1 General Remixing Trends (RQ1)

RQ1.1: How do programmers modify the project elements of remixed projects?

As the numbers in Table 6.2 indicate, the programming activities in the remixes that modify and delete sprites are more prevalent than those that add new sprites. However, having inspected the sprites present in both the original projects and their remixes (i.e., disregarding the deleted and newly inserted sprites), we find that programmers more frequently insert blocks than delete them. Out of 5,384 projects, only 123 remixes added and 16 remixes deleted variables. Out of the remixes that added variables, only 9 happen to have added global variables (7%). Hence, the remixes tend to disfavor global variables as a programming construct. In addition, we discover that programmers have not modified any sound components (i.e audio related files) in the Scratch project.

From Figure 6.1, we observe that programmers often alter control blocks in the remixes. In

Table 6.2: Mean modification values in 5,384 remixes (excluding unmodified remixes)

Scratch Elements	Insertion	Deletion	Alteration
Sprite	0.253	0.484	1.365
Blocks	8.305	7.409	6.56
Costume	0.411	0.978	
Sound	0	0	
Global variable	0.003	0	
Local variable	0.167	0.004	

addition, we observe that around 35% of the altered blocks change some sprite attributes (i.e., motion and looks blocks).

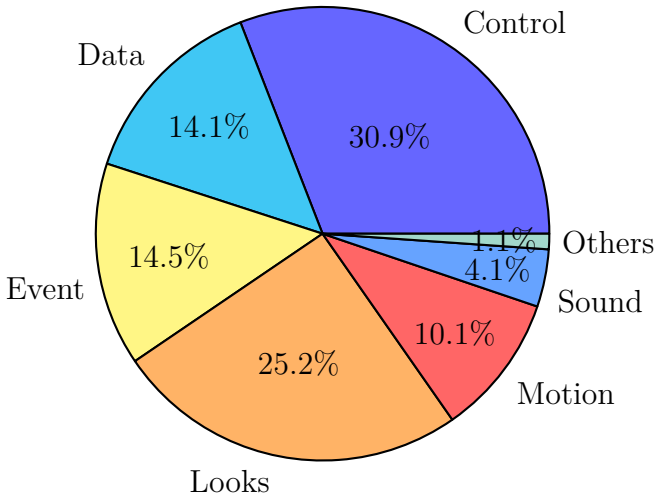


Figure 6.1: Proportion of block types altered

We find that out of 5,384 projects, 2,545 projects (47.2%) contain script re-positioning as a part of their modification. We also find that scripts have been re-arranged in 36 projects without any modifications. A deeper examination of these projects reveals that 3 of them took advantage of the the "Clean Up" feature of Scratch before having been shared. Given the automated nature of this feature, it could mean either that few Scratch programmers care about the workspace's tidiness (i.e., how blocks are arranged in the project workspace)

Table 6.3: Assessing change of the code quality in the remixes with paired t-tests

	ABC Metric	Program Volume	Cyclomatic Complexity
t Stat	3.742	9.654	5.522
P (T≤t) one-tail	$9.193\text{x}10^{-5}$	$3.013\text{x}10^{-22}$	$1.718\text{x}10^{-8}$
t Critical one-tail	1.645	1.645	1.645
P (T≤t) two-tail	$1.838\text{x}10^{-4}$	$6.026\text{x}10^{-22}$	$3.438\text{x}10^{-8}$
t Critical two-tail	1.960	1.960	1.960

or that they maintain tidiness as a regular programming practice. Another peculiar finding is that some programmers choose to move the top script within the workspace without any further changes before sharing the project. This behavior may imply some inchoate attempt to realize a programming task.

RQ1.2: Does the code quality change between the original projects and their remixes?

Based on the paired t-test (two tailed) from Table 6.3, we reject the null hypothesis of no difference in the three code quality metrics between the original projects and their remixes ($p < 0.05$). This finding means that there is significant change in code quality of projects and its remixes. On conducting the lower-tailed t-tests for the three code quality metrics, we conclude that there is a decrease in all of the code quality metrics ($p < 0.05$) in remixes.

RQ1.3: What are the trends in the code quality of remixes once their original project is shared?

We find that most remixes maintain the code quality levels of the original projects over time with minor fluctuations. In addition, for about a third of the original projects (31.88%), their

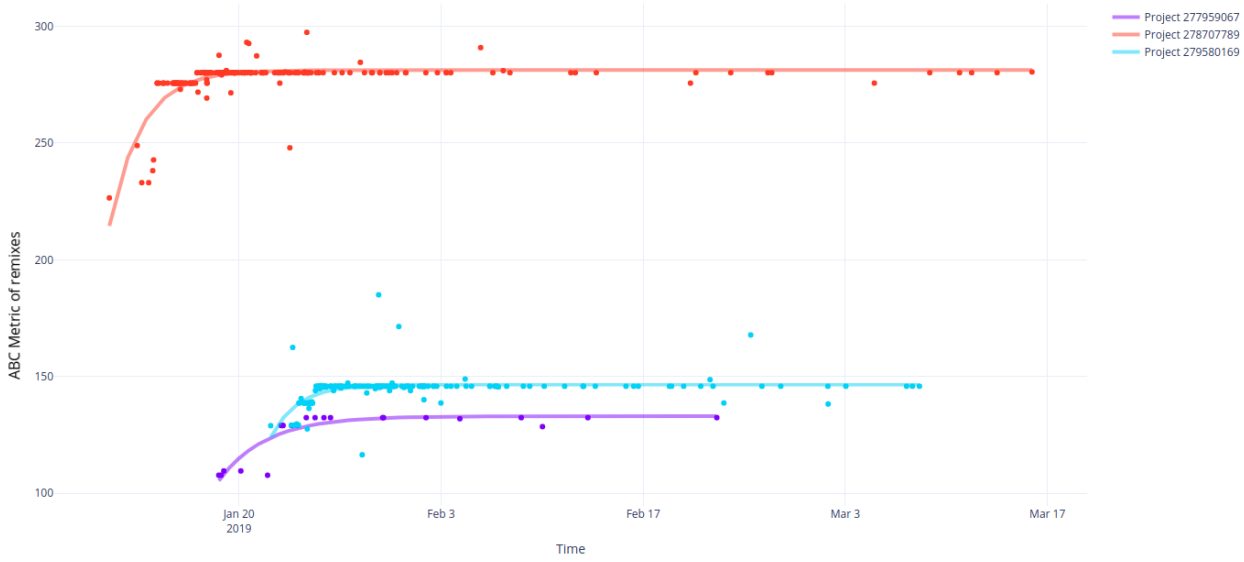


Figure 6.2: Change in ABC metric over time

remixes show a gradual decrease in code quality. This decreased quality is then maintained at a stable level over time. Figure 6.2 illustrates this observation for three representative projects. Finally, for 2 original projects, their remixes show a substantial fluctuation in code quality.

6.2 Impact of Code Quality on Remixing Modifications (RQ2)

RQ2.1: Are poor code quality projects more likely to have remixes with smaller-sized modifications?

From Figure 6.3, we observe that the program size and the ABC metric are lower for bigger modification percentages and higher for unmodified projects. We observe this trend for

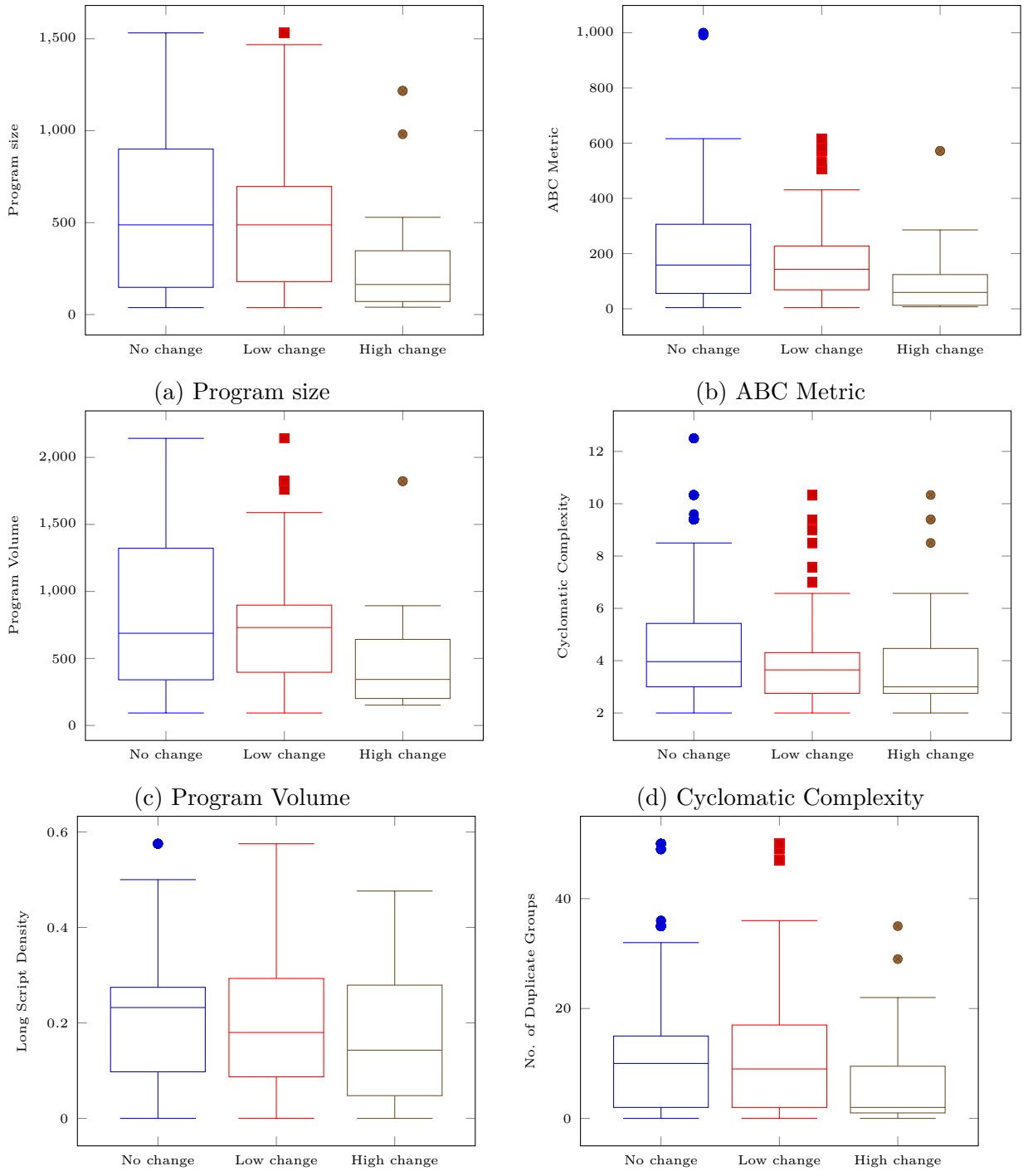
Table 6.4: Comparing mean code quality based on modification %

Code Quality Metric	No modification (0%)	Low modification (0-50%)	High modification (50-100%)
Calculated Metrics			
Program Size	551.295	520.679	256.855
ABC Metric	206.434	182.548	98.614
Program Volume	779.544	729.561	466.666
Cyclomatic Complexity	4.251	3.765	3.764
Code smells			
Long Script Density	0.204	0.203	0.169
Duplicate Groups	12.909	12.197	5.547

other quality metrics as well, as described in Table 6.4. However, cyclomatic complexity metric does not differ drastically between the low modification remixes and high modification remixes as compared to the other metrics. Based on the ANOVA test of these groups, we validate that they differ significantly in each of the cases ($p < 0.05$). This finding implies that high quality projects (i.e., projects with low metric values) are modified to a greater degree in their remixes. Or conversely, the remixes of lower quality projects tend to be modified to a lesser extent.

RQ2.2: Does code quality affect the time it takes to modify a project in its remixes?

From Figure 6.4, we observe that the program volume values are higher when the programmers have taken either no time or too long to modify. We observe this trend for other quality metrics as well, as described in Table 6.5, except for cyclomatic complexity. The cyclomatic complexity of projects that took no time to modify is less than those projects that took upto 3.6 days to modify, as opposed to the trend observed for other metrics. Based on the



(a) Program size

(b) ABC Metric

(c) Program Volume

(d) Cyclomatic Complexity

(e) Long Script

(f) Duplicate Groups

Figure 6.3: Code quality metrics of original projects in three categories of remixes based on modification percentage

Table 6.5: Comparing mean code quality based on modification time

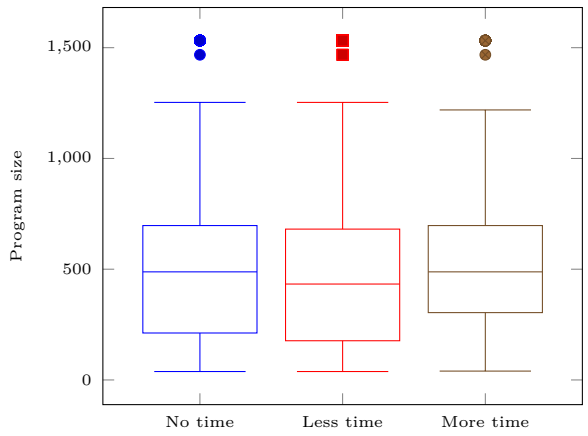
Code Quality Metric	No modification time (0min)	Low modification time (0 min - 3.6 days)	High modification time (3.6 - 66.8 days)
Calculated Metrics			
Program Size	583.806	495.239	523.242
ABC Metric	200.934	173.932	186.339
Program Volume	760.964	706.015	752.394
Cyclomatic Complexity	3.681	3.748	3.886
Code smells			
Long Script Density	0.204	0.202	0.211
Duplicate Groups	13.861	11.769	12.982

ANOVA test of these groups, we validate that they differ significantly in each of the cases ($p < 0.05$). This finding implies that programmers are more comfortable working with high quality projects (i.e., projects with low metric values). Or conversely, when remixing lower quality projects, programmers tend to take longer or avoid remixing altogether.

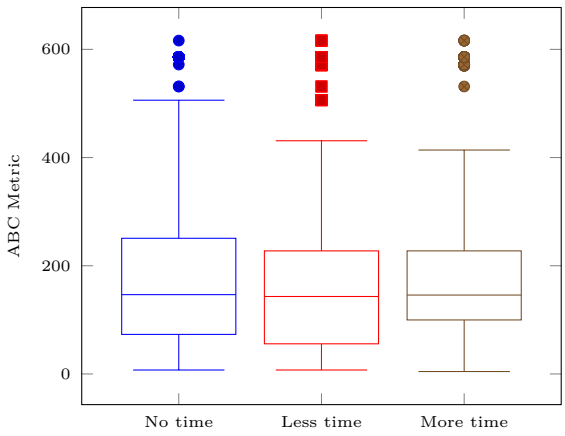
6.3 Learning from Original Projects (RQ3)

RQ3.1: Does the presence of sprites in a project containing "When I start as clone" block impact the number of duplicate sprites in its remixes?

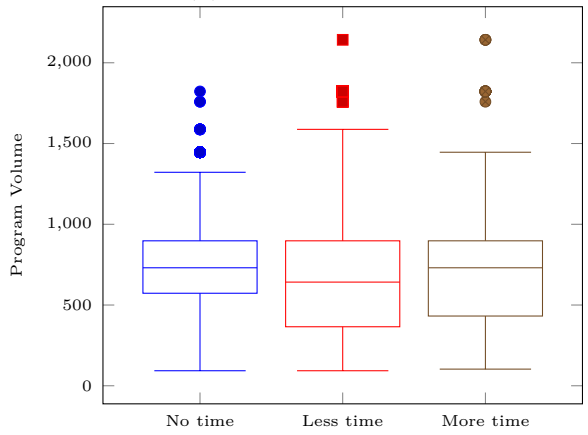
In 5,187 remixes, we find 75 remixes that introduce duplicate sprites. Out of them, the original projects of 52 (69.3%) remixes contain "When I start as a clone" block. Surprisingly, the presence of this block in a project seems to have *no* impact on preventing duplicate sprites in the project's remixes. Duplicate sprites may be needed to express certain features in the



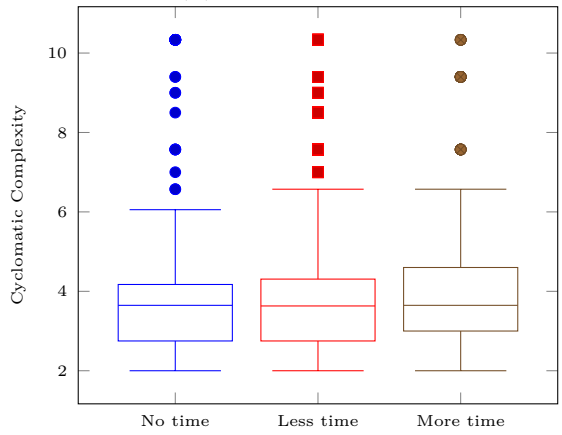
(a) Program size



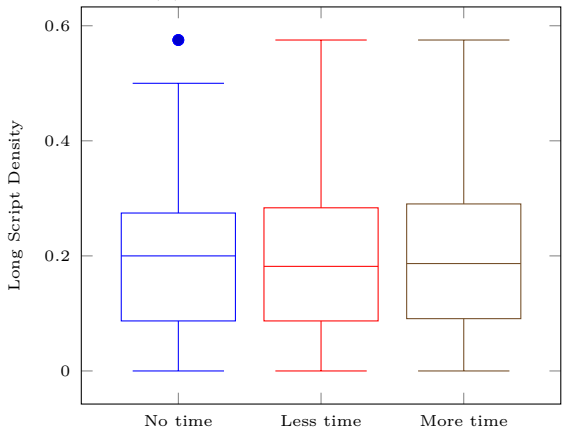
(b) ABC Metric



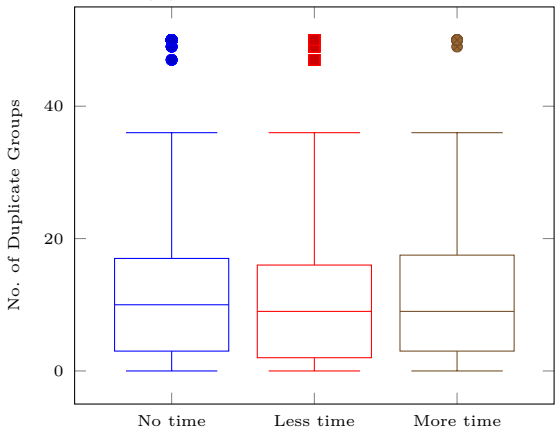
(c) Program Volume



(d) Cyclomatic Complexity



(e) Long Script



(f) Duplicate Groups

Figure 6.4: Code quality metrics of original projects in three categories of remixes based on modification time

original project that require cloning a sprite or a character. Some programmers may remain unaware of the intended capability of this special block.

RQ3.2: Does the presence of procedures in a project motivate its increased use in the remixes?

From 5187 remixes, 2529 are derived from the original projects that contain procedures. Out of the 2529 remixes, 36 projects (1.4%) add new procedures as part of their modifications of the original projects. Whereas, out of the remaining 2658 remixes derived from the original projects that contain no procedures, only 7 (0.2%) of them add new procedures. This finding indicates that when it comes to organizing code as procedures, programmers tend to follow the procedure usage trends established in an original project in its remixes.

Chapter 7

Threats to validity

The results reported above are subject to both external and internal threats to validity that we outline next.

7.1 External Validity

The main external threat to validity is that the sampling of Scratch projects we picked for our experiments may not have been fully representative of the whole collection of shared Scratch projects. Rather than collecting a random sample, we instead selected the most recent projects marked as *trending*. This risk is mitigated, however, as our subjects are among the most popular as identified by the Scratch community, so our findings certainly pertain to a highly relevant subset of projects.

Our study excludes projects that contain the Scratch 3.0's new extension blocks[1]. This external validity threat is mitigated by a very limited adoption rate of this feature.

[1] https://en.scratch-wiki.info/wiki/Scratch_Extension

7.2 Internal Validity

The internal validity can be threatened by our methodology for counting the number of modified blocks in a project. We count modifications at the statement level, so all affected blocks in a statement are counted as a single modification to the parent block. Even though the total number of affected blocks can be higher that we report, this methodology is congruent with the Scratch execution model and the way it is perceived by programmers.

For our experiments, we make use of the software quality metrics originally introduced in the context of text-based languages. Their applicability to Scratch may present another internal validity threat. For example, to compute the number of lines of code, we count the number of visual program statements. The imperative nature of Scratch mitigates this threat, with almost all visual constructs having direct counterparts in text-based languages.

Chapter 8

Discussion

In this section, we interpret the results of our experiments presented in Section 6 and discuss their implications for computing educators and language designers.

8.1 General Remixing Trends (RQ1)

The analysis driven by our first research question reveals that most remixes change sprite attributes and add sound related functionality. Indeed, the majority of remixed projects include some animation and gaming functionality, so it is warranted for block editors to provide a comprehensive palette of animation and media blocks. Despite the availability of the default option "For all sprites" for inserting new variables, programmers end up inserting sprite-specific variables, indicating their awareness of the scoping issues and preventing the introduction of the Broad Variable Scope smell. However, this disciplined programming practice observed in the remixes was found lacking in the prior study that assessed the software quality across all projects, finding Broad Variable Scope to be highly prevalent [Tec17]. This discrepancy may be explained by a higher level of programming experience typically required to remix a project. By investigating the experience level of the authors of the studied remixes, we determined that 60% of those who inserted local variables in their remixes are ranked as "Scratchers" (indicating a non-newbie status), thus confirming that experienced programmers are more aware of variable scoping issues.

Programmers of the studied remixes either maintain a tidy workspace or are unconcerned about workspace tidiness, as evidenced by the small percentage of programmers sharing projects via the "Clean up" facility. Remixes also exhibit lower ABC, program difficulty, and cyclomatic complexity metrics as compared to their original projects. This change is explained by the prevalence of deleting sprites as a part of remixing, thus lowering program complexity. The observed high number of unmodified remixes motivates the need for warning programmers about sharing duplicated projects, thus encouraging originality and creativity.

From Figure 6.2, we observe that the ABC metric of remixes increases gradually between releases and eventually becomes steady, matching the ABC metric of their original projects. One explanation is that the quality of the original projects keeps deteriorating over time, a trend reflected in their remixes. These findings are consistent with the established body of knowledge in software maintenance and evolution, as codified by the "Declining Quality" law of Software Evolution [Leh80]. Even though this law was defined based on the analysis of software systems implemented in text-based languages, it is interesting to see it manifesting itself in block-based software as well.

8.2 Code Quality and Remixing Modifications (RQ2)

We discovered that projects whose code is unnecessarily complex and low quality result in remixes with limited modifications. If a project is hard to read and understand, it tends to be only modestly modified in its remixes. In addition, we find that poor quality projects either require the most time due to difficulty in comprehending the project or the least time as programmers are discouraged to make modifications due to its complexity. Encouraging code comments to aid readability and quality improvement practices can invigorate the range and scale of remixing modifications.

8.3 Learning from Original Projects (RQ3)

Our conjecture that the presence of the "When I start as clone" block prevents sprite duplication turned to be false. The programming practices of a project tend to be followed in its remixes, at least as far as the issues of code quality are concerned. Programmers may find it hard to understand the concept of *cloneable sprites* or encounter scenarios in which duplicating sprites is unavoidable. Perhaps some programming abstraction that expresses similar behaviors across sprites without code duplication could reduce the presence of duplicated sprites.

When it comes to the use of procedures, the trends set in a project tend to be followed in its remixes. If a project uses procedures, its remixes are 7 times more likely to introduce new procedures than the remixes of a project without any procedures. For computing educators, this insight highlights the importance of teaching procedures, confirming the findings of a prior study of App Inventor applications [LTM17]. For language designers, it suggests the need for useful feedback and recommendation features in the programming environments to educate beginner programmers.

Chapter 9

Future Work

Although our results provide evidence on the impact of code quality on remixing and the adoption of programming practices while remixing in Scratch, to fully ascertain the practice of remixing practice would require a substantial follow-up effort. In a way, our study identified more research questions than it definitively answered. The future work ideas inspired by this study can be pursued by several major research directions that we outline below:

- To further delve into the patterns and coding practices that encourage remixing, we plan to investigate whether remixes modify the same code parts, that is, discover interesting scripts in Scratch projects that are modified by most programmers. The most modified interesting scripts can be then analyzed for their features like the presence of good programming practices to facilitate code comprehension or presence of smells which are then eliminated in the remixes.

- Our study focused on first-level of remixes to understand the remixing practice. The next step to this study would be to explore the remixing practices across all levels of the remix tree of the most popular projects to shed light on how code quality and prevalence of smells change across the levels. Examining remixing practices along the breadth and depth can provide further insights for computing educators and programming tool designers.

- An additional dimension of our study can categorize developers based on the types of

modifications they make in project remixes. In fact, our analysis infrastructure supports inquiries required to ascertain this information by storing developer ids alongside their projects. Identifying distinct developer profiles can be helpful for Scratch designers to be able to better accommodate the most common developer types and their programming practices.

- With the prevalence of copy-paste practices on Scratch, it is necessary to make sure that the output is useful to the community. In Scratch, every project has instructions for end-users and notes for programmers who wish to remix it. By examining how often programmers edit project instructions and notes when they introduce substantial modifications while remixing, we can gain insights into the willingness of making the code useful for the community.

- Given the observed strong impact of a project's code quality on the types and magnitude of the modifications in its remixes, one should consider how Scratch programmers can be encouraged to improve the code quality of their completed projects. To reach that objective would require both raising an awareness of code quality via new pedagogical approaches and supporting code quality improvement with new types of automated programming tools.

Chapter 10

Conclusions

We conducted an empirical study of 8,142 projects with the goal of understanding and improving the remixing culture in Scratch. Our exploration centers on three basic themes of the remixing behavior: general remixing trends, the impacts of code quality on the nature of remixes, and the influence of a project's programming practices on that in its remixes. We discovered some interesting practices that programmers exhibit when remixing projects. One such practice is the prevalent use of local variables, as an indication of a programmer trying to avoid introducing what is known as *the Broad Variable Scope* smell. We found that very few programmers use the "Clean Up" functionality of Scratch, an observation that Scratch designers may want to act upon. We also confirmed that a project's code quality and coding practices do affect how programmers modify code in its remixes and how long it takes to make the modifications. Finally, we assessed how programmers learn advanced programming constructs from an existing project through the process of remixing. Our findings can be applied to encourage and expand remixing, as a highly effective communal learning technique that can also be better supported by programming environments. Our study motivates further research to fully comprehend the multifaceted nature of the practice of remixing.

Bibliography

[AH16] 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. Melbourne, VIC, Australia: ACM, 2016, pp. 53–61. isbn: 978-1-4503-4449-4. doi: 10.1145/2960310.2960325. url: http://doi.acm.org/10.1145/2960310.2960325.

[APZ18] Duaa Alawad, Manisha Panta, and Minhaz Fahim Zibran. "An Empirical Study of the Relationships between Code Readability and Software Complexity". In: 2018.

[Bel+07] Stefan Bellon et al. "Comparison and evaluation of clone detection tools". In: *IEEE Transactions on software engineering* 33.9 (2007), pp. 577–591.

[BTZ07] Steven Burrows, Seyed MM Tahaghoghi, and Justin Zobel. "Efficient plagiarism detection for large code repositories". In: *Software: Practice and Experience* 37.2 (2007), pp. 151–175.

[BW10] R. P. L. Buse and W. R. Weimer. "Learning a Metric for Code Readability". In: *IEEE Transactions on Software Engineering* 36.4 (July 2010), pp. 546–558. issn: 0098-5589. doi: 10.1109/TSE.2009.70.

[Cha+13] Dong-Kyu Chae et al. "Software plagiarism detection: a graph-based approach". In: *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. ACM. 2013, pp. 1577–1580.

[Che+14] Giorgos Cheliotis et al. “The Antecedents of Remix”. In: *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*. CSCW ’14. Baltimore, Maryland, USA, 2014, pp. 1011–1022. isbn: 978-1-4503-2540-0. doi: 10.1145/2531602.2531730. url: http://doi.acm.org/10.1145/2531602.2531730.

[Das+16] Sayamindu Dasgupta et al. “Remixing as a Pathway to Computational Thinking”. In: *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing - CSCW 16* (2016). doi: 10.1145/2818048.2819984.

[ĐG12] Zoran Đurić and Dragan Gašević. “A source code similarity system for plagiarism detection”. In: *The Computer Journal* 56.1 (2012), pp. 70–86.

[Don+19] Yihuan Dong et al. “Defining Tinkering Behavior in Open-ended Block-based Programming Assignments”. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education - SIGCSE 19* (2019). doi: 10.1145/3287324.3287437.

[Fit96] Ronan Fitzpatrick. “Software quality: Definitions and strategic issues”. In: (Apr. 1996).

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999. isbn: 0-201-48567-2.

[FR87] Jinan AW Faidhi and Stuart K Robinson. “An empirical approach for detecting program similarity and plagiarism within a university programming environment”. In: *Computers & Education* 11.1 (1987), pp. 11–19.

[HA16] Felienne Hermans and Efthimia Aivaloglou. “Do code smells hamper novice programming? A controlled experiment on Scratch programs”. In: *2016 IEEE*

24th International Conference on Program Comprehension (ICPC) (2016). doi: 10.1109/icpc.2016.7503706.

[Hal77] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977. isbn: 0444002057.

[HM13] Benjamin Mako Hill and Andrés Monroy-Hernández. "The cost of collaboration for code and art". In: *Proceedings of the 2013 conference on Computer supported cooperative work - CSCW 13* (2013). doi: 10.1145/2441776.2441893.

[HM15] Benjamin Mako Hill and Andrés Monroy-Hernández. "The Remixing Dilemma: The Trade-off Between Generativity and Originality". In: *CoRR* abs/1507.01295 (2015). arXiv: 1507.01295. url: http://arxiv.org/abs/1507.01295.

[JSC07] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. "Context-based detection of clone-related bugs". In: *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM. 2007, pp. 55–64.

[KG03] Cory Kapser and Michael W Godfrey. "Toward a taxonomy of clones in source code: A case study". In: *Evolution of large scale industrial software architectures* 16 (2003), pp. 107–113.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. "CCFinder: a multilinguistic token-based code clone detection system for large scale source code". In: *IEEE Transactions on Software Engineering* 28.7 (2002), pp. 654–670.

[Kri01] Jens Krinke. "Identifying similar code with program dependence graphs". In: *Proceedings Eighth Working Conference on Reverse Engineering*. IEEE. 2001, pp. 301–309.

[KS16] M A Kuznetsov and V O Surkov. “Analysis of complexity metrics of a software code for obfuscating transformations of an executable code”. In: *IOP Conference Series: Materials Science and Engineering* 155 (Nov. 2016), p. 012008. doi: 10.1088/1757-899x/155/1/012008. url: https://doi.org/10.1088/1757-899X/155/1/012008.

[KW13] Nadia Kasto and Jacqueline Whalley. “Measuring the Difficulty of Code Comprehension Tasks Using Software Metrics”. In: *Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136*. ACE ’13. Adelaide, Australia: Australian Computer Society, Inc., 2013, pp. 59–65. isbn: 978-1-921770-21-0. url: http://dl.acm.org/citation.cfm?id=2667199.2667206.

[Leh80] M. M. Lehman. “Programs, life cycles, and laws of software evolution”. In: *Proceedings of the IEEE* 68.9 (Sept. 1980), pp. 1060–1076. issn: 0018-9219. doi: 10.1109/PROC.1980.11805.

[LTM17] Isabelle Li, Franklyn Turbak, and Eni Mustafaraj. “Calls of the wild: Exploring procedural abstraction in app inventor”. In: *2017 IEEE Blocks and Beyond Workshop (B&B)*. IEEE. 2017, pp. 79–86.

[LW06] Jin-Cherng Lin and Kuo-Chiang Wu. “A Model for Measuring Software Understandability”. In: *The Sixth IEEE International Conference on Computer and Information Technology (CIT’06)*. Sept. 2006, pp. 192–192. doi: 10.1109/CIT.2006.13.

[LW08] Jin-Cherng Lin and Kuo-Chiang Wu. “Evaluation of software understandability based on fuzzy matrix”. In: *2008 IEEE International Conference on Fuzzy Systems (IEEE World Congress on Computational Intelligence)*. June 2008, pp. 887–892.

[MAB13] Orni Meerbaum-Salant, Michal Armoni, and Mordechai (Moti) Ben-Ari. "Learning computer science concepts with Scratch". In: *Computer Science Education* 23.3 (2013), pp. 239–264. doi: 10.1080/08993408.2013.832022.

[McC76] T. J. McCabe. "A Complexity Measure". In: *IEEE Trans. Softw. Eng.* 2.4 (July 1976), pp. 308–320. issn: 0098-5589. doi: 10.1109/TSE.1976.233837. url: https://doi.org/10.1109/TSE.1976.233837.

[MRR16] J. Moreno-León, G. Robles, and M. Román-González. "Comparing computational thinking development assessment scores with software complexity metrics". In: *2016 IEEE Global Engineering Education Conference (EDUCON)*. Apr. 2016, pp. 1040–1045. doi: 10.1109/EDUCON.2016.7474681.

[Ott76] Karl J Ottenstein. "An algorithmic approach to the detection and prevention of plagiarism". In: (1976).

[Pap87] Seymour Papert. *Constructionism: A New Opportunity for Elementary Science Education*. 1987. url: https://nsf.gov/awardsearch/showAward?AWD_ID=8751190.

[PHD11] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. "A Simpler Model of Software Readability". In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. MSR '11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 73–82. isbn: 978-1-4503-0574-7. doi: 10.1145/1985441.1985454. url: http://doi.acm.org/10.1145/1985441.1985454.

[Res+09] Mitchel Resnick et al. "Scratch: Programming for All". In: *Commun. ACM* 52.11 (Nov. 2009), pp. 60–67. issn: 0001-0782. doi: 10.1145/1592761.1592779. url: http://doi.acm.org/10.1145/1592761.1592779.

[SWA03] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. "Winnowing: Local Algorithms for Document Fingerprinting". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD '03. San Diego, California: ACM, 2003, pp. 76–85. isbn: 1-58113-634-X. doi: 10.1145/872757.872770. url: http://doi.acm.org/10.1145/872757.872770.

[Tec17] Peeratham Techapalokul. "Sniffing Through Millions of Blocks for Bad Smells". In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '17. Seattle, Washington, USA: ACM, 2017, pp. 781–782. isbn: 978-1-4503-4698-6. doi: 10.1145/3017680.3022450. url: http://doi.acm.org/10.1145/3017680.3022450.

[TT17] Peeratham Techapalokul and Eli Tilevich. "Understanding recurring quality problems and their impact on code sharing in block-based software". In: *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2017). doi: 10.1109/vlhcc.2017.8103449.

[WW17] David Weintrop and Uri Wilensky. "Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms". In: *ACM Transactions on Computing Education* 18.1 (2017), pp. 1–25. doi: 10.1145/3089799.

Appendices

Appendix A

Scratch Project Structure

Scratch projects are downloaded as .sb3 extension files by clicking File > Save to your computer. The project file contains project.json file and other media asset files used in the project. All project elements are stored in the project.json file. The JSON file structure is described in the Figure A.1.

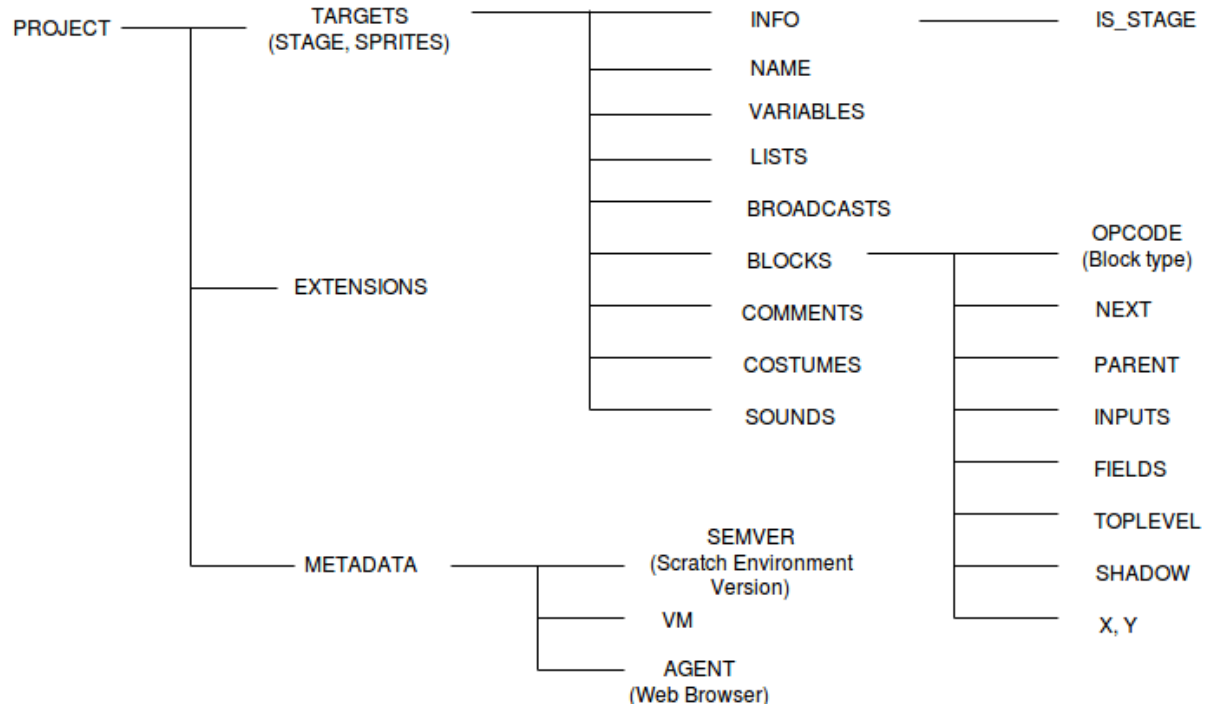


Figure A.1: Scratch Project Structure

Appendix B

MongoDB Queries

Here, we present the queries used to obtain relevant information from data collected in MongoDB.

To get modifications made to all Scratch elements - blocks, sprites, costumes, sounds, global variables, local variables:

```
db.getCollection('remix_projects').aggregate([
    {   $lookup:{
            from: "remix_stats",       // other table name
            localField: "_id",   // name of users table field
            foreignField: "_id", // name of userinfo table field
            as: "remix_st"         // alias for new table
        }
    },
    {   $unwind:"$remix_st" },
    {   $lookup:{
            from: "general_metrics_o",       // other table name
            localField: "original_project",   // name of users table field
            foreignField: "_id", // name of userinfo table field
            as: "general_st"         // alias for new table
        }
    },
    {   $unwind:"$general_st" },
    {   $match:{
         "remix_st.modification_percent" : { $exists : true },
         "remix_st.script_cyclomatic_complexity" : { $exists : true },
         "remix_st.program_volume" : { $exists : true },
         "remix_st.abc_score" : { $exists : true },
         "remix_st.long_script_dens" : { $exists : true }
     }
    },
    {   $project:{
```

```
            _id : 1,
            "original_project" : 1,
            "locs_original" : "$general_st.locs",
            "abc_original" : "$general_st.abc_score",
            "pv_original" : "$general_st.program_volume",
            "cc_original" : "$general_st.script_cyclomatic_complexity",
            "ls_original" : "$general_st.long_script_dens",
            "modification_percent" : "$remix_st.modification_percent",
            "blocksInserted" : { $ifNull : ["$remix_st.blocksInserted", 0]},
            "blocksDeleted" : { $ifNull : ["$remix_st.blocksDeleted", 0]},
            "blocksModified" : { $ifNull : ["$remix_st.blocksModified", 0]},
            "spritesInserted" : { $ifNull : ["$remix_st.spritesInserted", 0]},
            "spritesDeleted" : { $ifNull : ["$remix_st.spritesDeleted", 0]},
            "spritesModified" : { $ifNull : ["$remix_st.spritesModified", 0]},
            "costumeInserted" : { $ifNull : ["$remix_st.costumeInserted", 0]},
            "costumeDeleted" : { $ifNull : ["$remix_st.costumeDeleted", 0]},
            "soundInserted" : { $ifNull : ["$remix_st.soundInserted", 0]},
            "soundDeleted" : { $ifNull : ["$remix_st.soundDeleted", 0]},
            "globalVarInserted" : { $ifNull : ["$remix_st.globalVarInserted", 0]},
            "globalVarDeleted" : { $ifNull : ["$remix_st.globalVarDeleted", 0]},
            "localVarInserted" : { $ifNull : ["$remix_st.localVarInserted", 0]},
            "localVarDeleted" : { $ifNull : ["$remix_st.localVarDeleted", 0]}
        }
    }])
```

To get the distribution of block categories modified:

```
db.getCollection("remix_stats").aggregate([
    {   $match:{
            "modification_percent" : { $gt : 0 },
        }
    },
    {   $project:{
            "motion" : { $ifNull : ["$motion", 0]},
            "looks" : { $ifNull : ["$looks", 0]},
            "event" : { $ifNull : ["$event", 0]},
            "sound" : { $ifNull : ["$sound", 0]},
            "control" : { $ifNull : ["$control", 0]},
            "sensing" : { $ifNull : ["$sensing", 0]},
            "data" : { $ifNull : ["$data", 0]},
            "pen" : { $ifNull : ["$pen", 0]},
        }
    }])
```

To get remix code quality time-series:

```
db.getCollection('remix_projects').aggregate([
    {   $lookup:{
            from: "remix_stats",
            localField: "_id",
            foreignField: "_id",
            as: "remix_st"
        }
    },
    {   $unwind:"$remix_st" },
    {   $lookup:{
            from: "original_stats",
            localField: "original_project",
            foreignField: "_id",
            as: "general_st"
        }
    },
    {   $unwind:"$general_st" },
    {   $match:{
         "remix_st.modification_percent" : { $exists : true },
     }
    },
    {   $project:{
            _id : 1,
            "shared" : 1,
            "modified" : 1,
            "original_project" : 1,
            "abc_score" : "$remix_st.abc_score",
            "abc_score_original" : "$general_st.abc_score",
            "program_difficulty" : "$remix_st.program_difficulty",
            "program_difficulty_original" : "$general_st.program_difficulty",
        "script_cyclomatic_complexity" : "$remix_st.script_cyclomatic_complexity",
        "script_cyclomatic_complexity_original" : "$general_st.script_cyclomatic_complexity"
        }
    }])
```

To get code quality vs. modification size and modification percent:

```
db.getCollection('remix_projects').aggregate([
    {   $lookup:{
            from: "remix_stats",
            localField: "_id",
            foreignField: "_id",
```

```
            as: "remix_st"
        }
    },
    {   $unwind:"$remix_st" },
    {   $lookup:{
            from: "original_stats",
            localField: "original_project",
            foreignField: "_id",
            as: "general_st"
        }
    },
    {   $unwind:"$general_st" },
    {   $match:{
         "remix_st.modification_percent" : { $exists : true },
         "remix_st.script_cyclomatic_complexity" : { $exists : true },
         "remix_st.program_volume" : { $exists : true },
         "remix_st.abc_score" : { $exists : true }
     }
    },
    {   $project:{
            _id : 1,
            "original_project" : 1,
            "locs_original" : "$general_st.locs",
            "abc_original" : "$general_st.abc_score",
            "pv_original" : "$general_st.program_volume",
            "cc_original" : "$general_st.script_cyclomatic_complexity",
            "ls_original" : "$general_st.long_script_dens",
            "ds_original" : "$general_st.duplicate_groups",
            "modification_percent" : "$remix_st.modification_percent",
            "time_to_modify_min" : "$remix_st.timeToModifyInMin"
        }
    }])
```

To get the number of projects whose original project contains “When I start as clone” block and those which do not:

```
db.getCollection('remix_stats').find(
{
    modification_percent : { $exists: true},
    blocksTotal : { $lt : 3207 },
   startCloneSpritesInOriginal: { $exists: true, $gt : 0 }, //use $eq : 0 for projects not
                                                            containing cloneable sprites
    duplicateSpritesInOriginal: { $exists: true, $eq : 0 },
   duplicateSprites: { $exists: true, $eq : 0 } //to find remixes with no duplicate sprites
```

```
}, {
    startCloneSpritesInOriginal: 1,
    duplicateSprites: 1
}).itcount()
```

To get projects containing an addition of new procedures when its original project contains a procedure:

```
db.getCollection('remix_stats').aggregate([
    {   $lookup:{
            from: "original_stats",
            localField: "original_project",
            foreignField: "_id",
            as: "gen_metrics"
        }
    },
    {   $unwind:"$gen_metrics" },
    {   $match:{
            "proceduresInserted" : { $gt : 0 }
        }
    },
    {   $project:{
            _id : 1,
            "proceduresInserted" : 1,
            "delta_duplicate_groups" : { $subtract: [ "$duplicate_groups",
                                        "$gen_metrics.duplicate_groups" ] }
        }
    }])
```