Understanding Common Scratch Programming Idioms and Their Impact on Project Remixing

Xingyu Long

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

Osman Balci

Dimitrios S. Nikolopoulos

May 10, 2021

Blacksburg, Virginia

Keywords: Programming idioms; Scratch; Block-based programming; Novice programmers; Project remixing;

Copyright 2021, Xingyu Long

Understanding Common Scratch Programming Idioms and Their Impact on Project Remixing

Xingyu Long

(ABSTRACT)

As Scratch has become one of the most popular educational programming languages, understanding its common programming idioms can benefit both computing educators and learners. This understanding can fine-tune the curricular development to help learners master the fundamentals of writing idiomatic code in their programming pursuits. Unfortunately, the research community's understanding of what constitutes idiomatic Scratch code has been limited. To help bridge this knowledge gap, we systematically identified idioms as based on canonical source code, presented in widely available educational materials. We implemented a tool that automatically detects these idioms to assess their prevalence within a large dataset of over 70K Scratch projects in different demographic and project categories. Since communal learning and the practice of remixing are one of the cornerstones of the Scratch programming community, we studied the relationship between common programming idioms and remixes. Having analyzed the original projects and their remixes, we observed that different idioms may associate with dissimilar types of code changes. Code changes in remixes are desirable, as they require a meaningful programming effort that spurs the learning process. The ability to substantially change a project in its remixes hinges on the project's code being easy to understand and modify. Our findings suggest that the presence of certain common idioms can indeed positively impact the degree of code changes in remixes. Our findings can help form a foundation of what comprises common Scratch programming idioms, thus benefiting both introductory computing education and Scratch programming tools.

Understanding Common Scratch Programming Idioms and Their Impact on Project Remixing

Xingyu Long

(GENERAL AUDIENCE ABSTRACT)

With over 68 million users and growing, Scratch has become one of the most popular programming languages for introductory computing learners. As with learning any programming language, understanding common programming idioms used in the language's application domain is important for both computing educators and learners. Educators need this understanding in order to fine-tune their curricular development, while learners can leverage this knowledge to effectively master the fundamentals by writing idiomatic code. Unfortunately, our understanding of what constitutes idiomatic Scratch code thus far has been limited. To address this knowledge gap, we systematically identified idioms based on source code with good code quality, as presented in widely available educational materials. We implemented a tool that automatically detects these idioms to assess their prevalence within a large, diverse dataset of over 70K Scratch projects. Since communal learning and the practice of remixing are one of the cornerstones of the Scratch programming community, we studied the relationship between common programming idioms and remixes. Having analyzed the original projects and their remixes, we found that different idioms may associate with dissimilar types of code changes. The ability to change a project in its remixes hinges on the project's code being easy to understand and modify. Our findings suggest that the presence of certain common idioms can positively impact the degree of code changes in remixes. Our findings can help form a foundation of what comprises common Scratch programming idioms, thus benefiting both introductory computing education and Scratch programming tools.

Dedication

Stay hungry, Stay foolish.

Acknowledgments

I would like to express my deep gratitude to my advisor Dr. Eli Tilevich for guiding me through this exciting and interesting research area and supporting me all the way[1]. I would also like to thank Dr. Balci for his valuable comments on my thesis work and thank Dr. Nikolopoulos for his helpful suggestions on academic writing. I also like to thank the Department of Computer Science at Virginia tech to give me an opportunity to pursue my Master's study and helps me through this exciting and wonderful journey.

I would like to thank Dr. Peeratham (Karn) Techapalokul for his guidance and feedback. He has consistently helped me with making steady research progress and gave me insightful suggestions during our weekly meetings.

[1]This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 1744722. Any opinions, findings, and conclusions or recommendations expressed herein are those of the author(s) and do not necessarily reflect the views of NSF.

Contents

List of Figures ix

List of Tables x

1 Introduction 1

2 Background 5

2.1 Programming idioms 5

2.2 Code smells 6

2.3 Bug patterns 7

2.4 Scratch 7

2.5 Detecting code differences 8

3 Remixing in Scratch 10

3.1 Project #1: "Infinite Dash!" 10

3.1.1 Project Remix #1 11

3.1.2 Project Remix #2 12

3.2 Project #2: "Red" 12

3.2.1 Project Remix #1 13

3.2.2 Project Remix #2 14

4 Methods **16**

4.1 Identifying programming idioms 16

4.2 Studying the prevalence of Scratch programming idioms 17

4.3 Detecting programming idioms 18

4.4 Computing programming idioms metrics 18

5 Findings **20**

5.1 Common programming idioms in Scratch (RQ1) 20

5.1.1 Scratch-specific Programming Idioms 20

5.1.2 General programming idioms 22

5.2 Prevalence of programming idioms (RQ2) 24

5.3 Idioms and code changes through remixing (RQ3) 27

6 Discussion **30**

6.1 Identifying Scratch Idioms 30

6.2 Novice Programming Practices with Programming Idioms 31

6.3 Programming Idioms and Introductory Computing 32

6.4 Remixing Practices with Programming Idioms 32

7 Threats to validity **34**

8 Related work 35

9 Conclusions and future work 37

Bibliography 38

Appendices 43

Appendix A Example for obtaining Scratch projects 44

Appendix B Example for idiom analyzer 50

Appendix C Example for generating a summary for dataset 59

List of Figures

3.1 The project page for "Infinite Dash!" 11

3.2 Screenshot Sprites panel in original (left) and its remix (right) 12

3.3 Code changes in Forever IF/IF Else: insert two more blocks 13

3.4 Added more sprites in "BLUE" project 14

3.5 Code changes: delete clone-related blocks 14

5.1 Sensor Wait Until No Sensor 21

5.2 Broadcast And Stop 22

5.3 Delete List Element By Value 23

5.4 Nested IF Else 24

5.5 Repeat Ask 25

5.6 Insertion example for Forever IF/IF Else in original and its remix 28

List of Tables

4.1 Basic summary statistics of dataset of over trending. 18

5.1 Prevalence of programming idioms in trending, top Scratchers and studio datasets 25

5.2 Prevalence of programming idioms in each project category subset within the trending dataset 26

5.3 Prevalence of operations found in Nested IF Else 28

5.4 Prevalence of operations found in Forever IF/IF Else 29

List of Abbreviations

API Application Programming Interface

AST Abstract Syntax Tree

CT Computational Thinking

SCM Source Code Management

Chapter 1

Introduction

A *programming idiom* describes a recurring syntactic code pattern that implements a specific functionality. Because idiomatic code is conducive to good programming styles, programmers find programs that contain primarily idiomatic code easier to understand [30]. Similar to the effectiveness of design patterns in computing education pedagogy [15], recognizing programming idioms can be seen as an important part of introductory computing education. For more advanced learners, the awareness of common programming idioms can also help in transferring programming knowledge across languages.

Scratch has been a highly successful programming language and a learning community for novice and end-user programmers [26]. Currently, the Scratch community has over 68 million registered users with 72 million shared projects [5]. The block-based visual nature of Scratch enables introductory learning to quickly become familiar with basic programming knowledge and coding skills, so the language has become a mainstay of introductory computing classes in all levels [36]. Despite the abundance of Scratch tutorials and other learning materials, the prevalence and usage of common Scratch programming idioms remain unexplored. In this thesis, we address this knowledge gap by systematically studying this issue.

Understanding common programming idioms is important to be able to effectively use and learn a language. Some studies have indicated that certain aspects of the Scratch's open-ended learning model might not be as effective for all learners as previously thought, and why models that employ more guided learning can be beneficial [24]. Another study has

found that students superficially engage with those programming concepts that educators expected them to master [7, 18]. The highly interactive programming model of Scratch empowers students to learn through trial and error by tinkering with source code, in order to associate different code fragments with the program output. In the process, students may end up producing non-idiomatic code patterns and embracing them as part of their coding practices. In fact, writing non-idiomatic code repeatedly might have led learners to poor quality code as observed in prior studies, such as code smells [34] and bugs [20]. Steering students in the direction of using idiomatic coding patterns may help them in effectively mastering the fundamentals of programming, so they can quickly progress toward more advanced concepts.

In fact, the concept of patterns, which is embodied in programming idioms, in introductory programming is not new. A growing body of research focuses on anti-pattern (e.g., code smells, poor coding practices)—that is, patterns that should be avoided or refactored. For example, Frädrich et al. has found that a large portion of Scratch projects have been afflicted by software bugs [20]. However, the opposite of anti-patterns, as with programming idioms in Scratch, has been relatively unexplored.

The objective of this work is to better understand Scratch programming idioms by answering the following questions:

- **RQ1:** What are the common programming idioms in Scratch?

  To be comprehensive, identifying programming idioms should not only be based on a standpoint of general programming idioms but also the Scratch-specific programming domains (e.g., media computing, games, animations, etc.). We systematically identify idioms based on available educational resources, provided by educators and experienced Scratch programmers.

- **RQ2:** How common are those idioms in a large, diverse dataset of Scratch projects?

  Our goal is to understand how common these idioms are across the different categories of Scratch projects. We created a tool for detecting the identified idioms and applied it to a large number of Scratch projects to determine the prevalence of these idioms.

- **RQ3:** How does the code change within the detected common idioms when projects are remixed?

  Because certain idioms are conducive to program comprehension and modification, understanding how these idioms impact projects remixing can guide the design of educational strategies. We investigate how code changes within two types of detected idioms related to control flow (i.e., Nested IF Else and Forever IF/IF Else) between the original projects and their remixes.

We identified a total of 11 common idioms that belong to 2 main categories: 1) Scratch-specific (i.e., Change And Clone, Change And Wait, Repeat And Change, Sensor Wait Until No Sensor, Switch Backdrop And Broadcast, and Broadcast And Stop) and 2) general programming idioms (i.e., Iterate List, Delete List Element By Value, Forever IF/IF Else, Nested IF Else, and Repeat Ask). By analyzing Scratch projects for the presence of the aforementioned idioms, we discovered the high prevalence of these idioms. Additionally, by investigating how code changes between original projects and their remixes within the detected Nested IF Else and Forever IF/IF Else idioms, we discovered that the majority of these changes are code deletions. It suggests that programmers often to remove code blocks in their remixing projects to achieve their goal and they are more likely to maintain simpler projects.

This thesis makes the following contributions:

1. A catalog of 11 common Scratch programming idioms, including their definitions and

usage scenarios

2. An automated tool for automatically detecting programming idioms; we extended LitterBox [20], a state-of-the-art framework for analyzing Scratch programs

3. A large-scale study that assesses the prevalence of the studied programming idioms based on a diverse dataset of over 70K projects

4. A case study of code changes within two common control-flow idioms detected in the original projects and their remixes

The remainder of the thesis is organized as follows: Chapter 2 presents background information on programming idioms and Scratch. Chapter 3 describes remixing practice and examples of project remixes in Scratch. Chapter 8 provides a summary of prior studies on idioms and patterns in Scratch. Chapter 4 explains our approach used to answer the research questions. Chapter 5 presents the findings of our study. Chapter 6 discusses the significance and implications of our findings as well as the threats to validity of the study in Chapter 7. Finally, Chapter 9 presents future work direction and concluding remarks.

Chapter 2

Background

In this section, we provide background information on programming idioms, code smells, bug patterns and Scratch, which are needed to be able to understand the technical details of this work. The reader already familiar with these topics can safely skip this chapter.

2.1 Programming idioms

A programming idiom refers to a commonly recurring pattern of code in a given language. Since programming idioms are reflective of the prevailing programming style, they serve as a bottom-up type of programming knowledge, acquired via hands-on programming activities. Structurally, a programming idiom can comprise a single or multiple code fragments. In general, a programming idiom represents a simple functionality that the underlying programming language does not directly provide as a built-in library API. Programming idioms bear similarities with design patterns in the context of programming languages, as discussed in [21]. However, they apply at different levels of abstraction. The same programming idiom is often present in different programming languages, but manifests itself in different forms, as determined by the specific language's programming style (e.g., 'iterating over a range:' 'for (int i = 0; i < 10; i++)' in Java and 'for i in range(0, 10)' in Python).

A number of research works study programming idioms. Some early work, dating back into 1970's, suggests that programming idioms serve as a useful programming concept that can

be applied to teaching programming [31]. Smit et al. [32] examined code-conventions and their effects on maintainability. From their preliminary results, they identified several coding conventions and ran those conventions as metrics on open-source projects to determine how they impact maintainability.

As a more recent example of the state of the art, Allamanis and Sutton [9] provide a new perspective on patterns, referred to as "programming idioms" in our work. They investigated the project-specific patterns and extracted common patterns from open-source Java-based projects at Github, which is the biggest source code management (SCM) platform that provides a distributed version control for developers across the world. They developed HAGGIS, a powerful tool that can mine high-quality code idioms automatically. After a rigorous experiment, they identified a catalog of code idioms, categorized as project-specific and API-specific. Most recently, Alexandru et al. [8] built a catalog of 'Pythonic idioms' (i.e., programming idioms for Python) and explore how developers understand these Pythonic idioms. They suggested that the effects of idioms on code quality also hold for other programming languages.

2.2 Code smells

A code smell refers to a recurring problem related to program quality. What constitutes a code smell is a subjective decision, influenced by various contexts. Eliminating code smells in a program can greatly increase the program's code quality and make it less error-prone. Hermans et al. [23] explored the prevalence of code smells in block-based programming languages (i.e., Lego mindstorms EV3 and Microsoft's Kodu) and summarized a catalog of 11 code smells (i.e., dead code, deprecated interface, duplicate code, feature envy, inappropriate intimacy, lazy class, long method, many parameters, message chain, no-op, and unused

field). Techapalokul and Tilevich [34] further studied code smells in Scratch projects. They identified a catalog of 12 Scratch code smells and assessed the prevalence of each smell using a large representative dataset of over 600K projects. They also investigated how the presence of code smells in a project impacts the likelihood of that project being remixed.

2.3 Bug patterns

A bug pattern is a recurring pattern that is known to cause a program to behave incorrectly. Eliminating software bug patterns increases overall program quality, as this process is known to improve the overall program correctness. Frädrich et al. [20] explored common bug patterns in Scratch projects and presented a list of 25 common bugs (e.g., custom block with forever, unsent message, missing pen up, and etc.). They also implemented LitterBox [20], a program analysis framework and bug analyzer tool that can automatically detect bugs in Scratch programs.

2.4 Scratch

Released in 2007, Scratch has since become highly popular among novice programmers in both formal and informal educational settings [26]. Its intuitive visual programming interface allows programmers to compose a program by snapping jigsaw-like blocks together. Unlike in general text-based programming languages, such as Java and Python, learners in Scratch can quickly learn core programming concepts without the need to master the language's syntax first. Scratch makes learning programming engaging by empowering learners to create media-rich projects that include animations, games, tutorials, art, music, and stories. Scratch fosters a learning community, in which programmers learn from each other by sharing and

remixing projects. This practice can be seen as a form of code reuse, similar to forking in the context of professional software development.

A Scratch program comprises programmable objects (the Stage and its children objects known as "sprites"). The Stage and sprite objects are controlled by a set of scripts, which is a sequence of blocks. Scratch has made several design choices to keep their language features simple. For example, a Scratch procedure has no return value and cannot be shared among different sprite objects. Although minimal in its design, the Scratch language introduces novice programmers to several fundamental programming concepts, including sequences, flow controls, variables, abstractions, synchronizations, and Boolean conditions. In addition, the web-based Scratch programming interfaces comprises four major parts. The left part allows users to select blocks from different categories (i.e., motion, looks, sound, event, control, sensing, operators, variables, and customized blocks). In the middle part, programmers compose the program logic by snapping together different code blocks. The top right is the Stage, the output area that displays the program results visually. The bottom right is the sprite selection panel, where programmers create and configure new sprites.

Another advantage of Scratch is a wide availability of learning materials, including books, freely available programming tutorials, and community-curated programming tips. We used these materials as canonical examples of what constitutes a "good" Scratch programming style and our source of idioms.

2.5 Detecting code differences

Several prior research efforts focus on computing source code differences. A large number of techniques have been proposed including text-based, graph-based, and tree-based methods. The earliest work, dating back to the 70s and 80s, focus on metric-based detection of source

code similarity with the goal of preventing plagiarism [17, 29].

In this thesis research, we applied diff-match-path [1], a third-party library, originally developed to power Google Docs, to compute the differences of two block-based code fragments in terms of their text-based format (i.e., block plugin syntax [2]).

[1] https://github.com/google/diff-match-patch, a powerful library based on Myer's diff algorithm Myers [28] for calculating textual differences between two documents

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

Chapter 3

Remixing in Scratch

"Remixing" is a popular code reuse practice in the contexts of end-user software development and introductory computing, including the Scratch community. The practice bears similarity to how professional software developers extend an existing project by forking or cloning it. Remixing has been shown to help novice programmers grow their Scratch programming repertoire as working with unfamiliar source code introduces them to new programming concepts and constructs [16].

Next, we present examples of project remixing that can be helpful for understanding the technical details of this work related to how code changes in remixed projects.

3.1 Project #1: "Infinite Dash!"

Fig. 3.1 shows the view of the "Infinite Dash!" project [1]. This project is popular as indicated by its 1,507 favorite counts by the community members. The project has been remixed 97 times. The project implements a game whose objective is to collect as many coin rewards as possible while avoiding incoming spike obstacles by clicking the mouse to make the game character jump.

[1] https://scratch.mit.edu/projects/519883034/

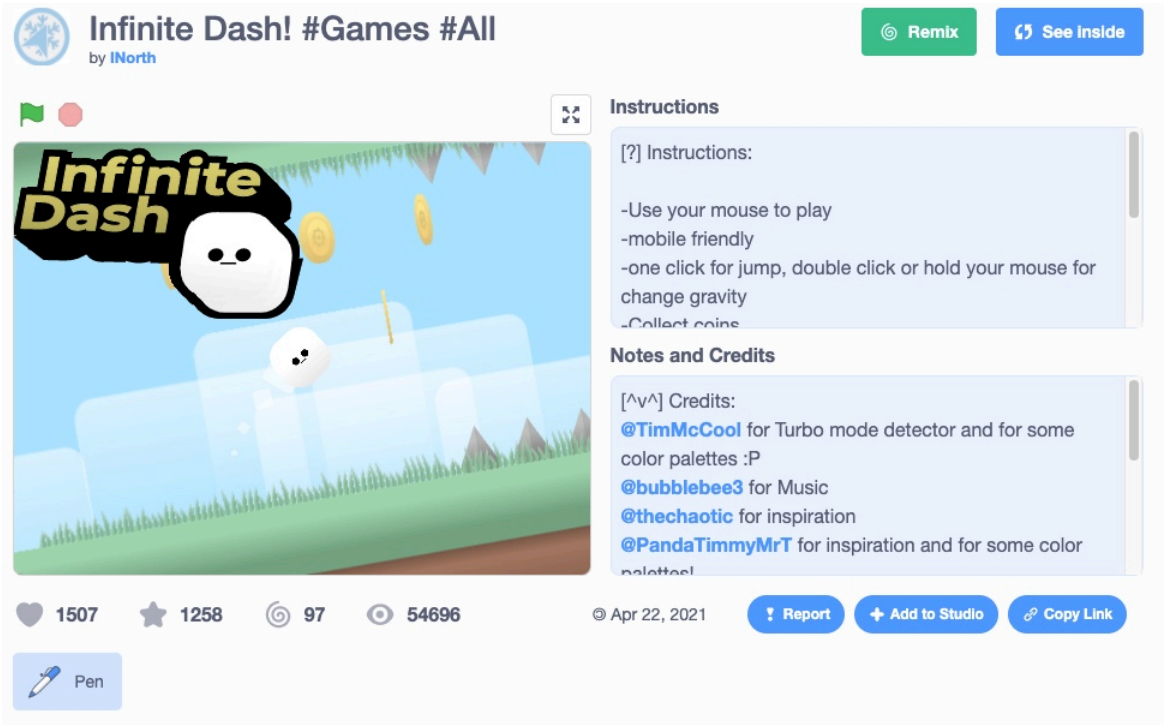

Figure 3.1: The project page for "Infinite Dash!"

3.1.1 Project Remix #1

In this project remix [2], the programmer extended the project "Infinite Dash!" by modifying the sprite appearance and adding other sprites. Fig. 3.2 shows the screenshot of the sprite panels of the original and its remix.

The added spike obstacles make the game more challenging by increasing the difficulty level of the original game. The scripts contained in the added sprites remain mostly similar to those contained in the original spike obstacle, except for a few small changes to constant values used in the game logic (e.g., how fast a spike should move, how long a spike should stay visible before it disappears, etc.).

[2] https://scratch.mit.edu/projects/524253104/

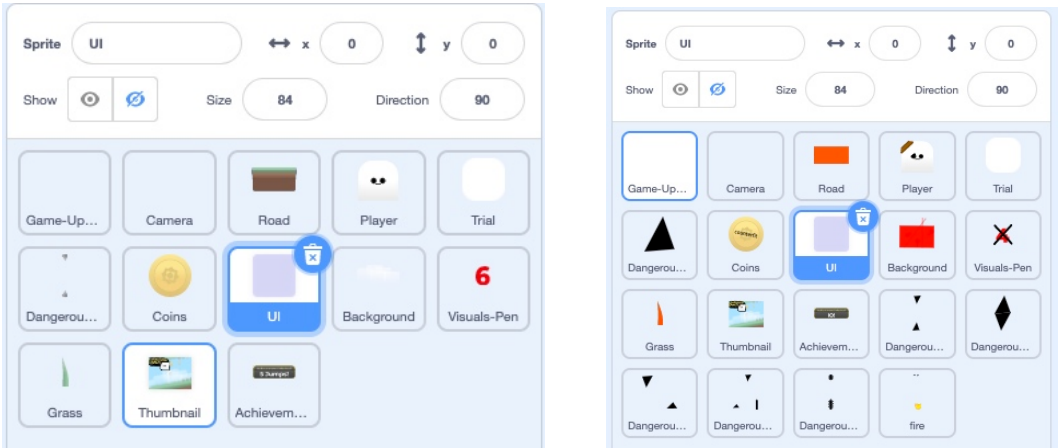

Figure 3.2: Screenshot Sprites panel in original (left) and its remix (right)

3.1.2 Project Remix #2

In this project remix [3], the programmer only changed the appearance of each sprite while keeping the original source code in each sprite unchanged. This pattern of remixing is common and worth considering when investigating how project code changes in remixes.

3.2 Project #2: "Red"

The objective of this game project [4] is to control a game character while avoiding blue obstacle objects. The game player can control the game character by pressing arrow keys. Similarly popular as Project #1, this project has been remixed 133 times.

[3] https://scratch.mit.edu/projects/525213318/
[4] https://scratch.mit.edu/projects/467335736/

3.2.1 Project Remix #1

In this project remix titled "Yellow"[5], the programmer maintains the original project structure. Although the game control's logic remains unchanged, several code changes were introduced into the original source code. This type of changes at the source code level is the focus of our work. Specifically, we investigated how a piece of code changes within the certain coding structure between the original projects and their remixes. The results help us better understand the role of programming idioms and their code changes in the remixes. In this remixed project, we found the insertion of two blocks in an instance of Forever IF/IF Else idiom. Fig. 3.3 demonstrates the changes.

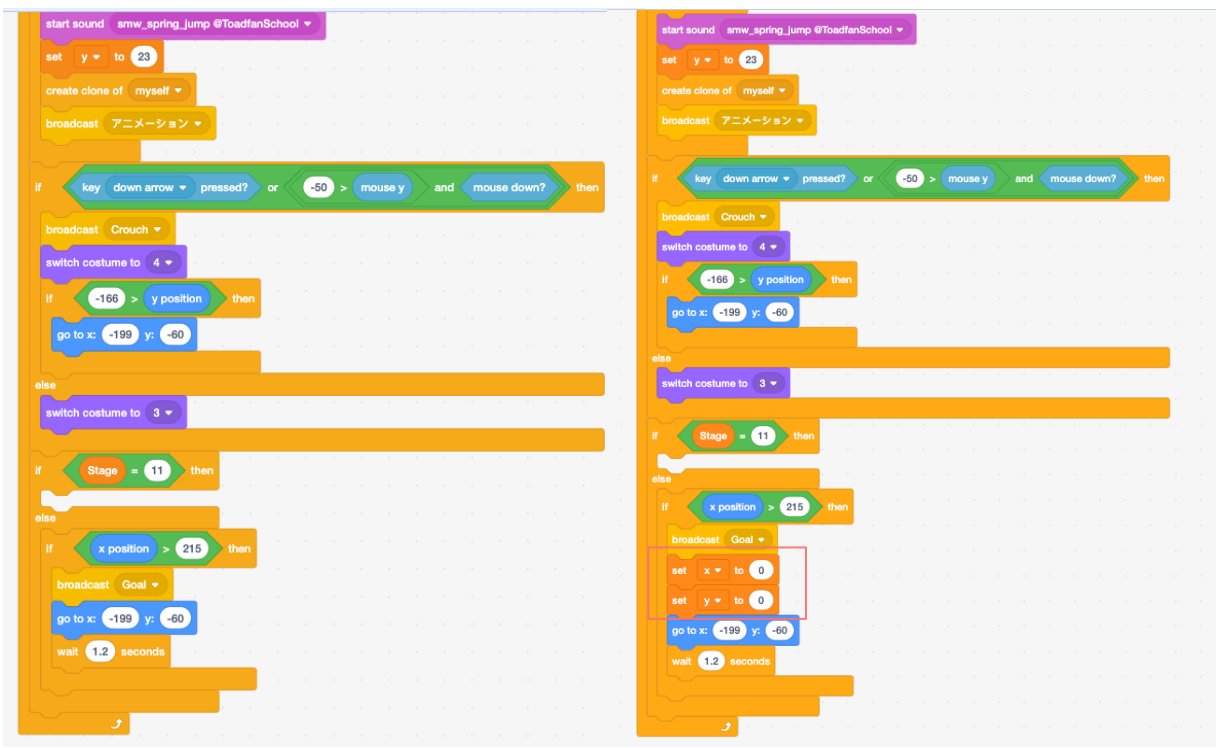

Figure 3.3: Code changes in Forever IF/IF Else: insert two more blocks

[5] https://scratch.mit.edu/projects/490627820

3.2.2 Project Remix #2

In this project remix[6] titled "Blue", we observed several sprite additions (Fig. 3.4), and deletions of programming blocks in certain idioms (Fig. 3.5). Code modifications in these idioms are mostly in the form of value updates.

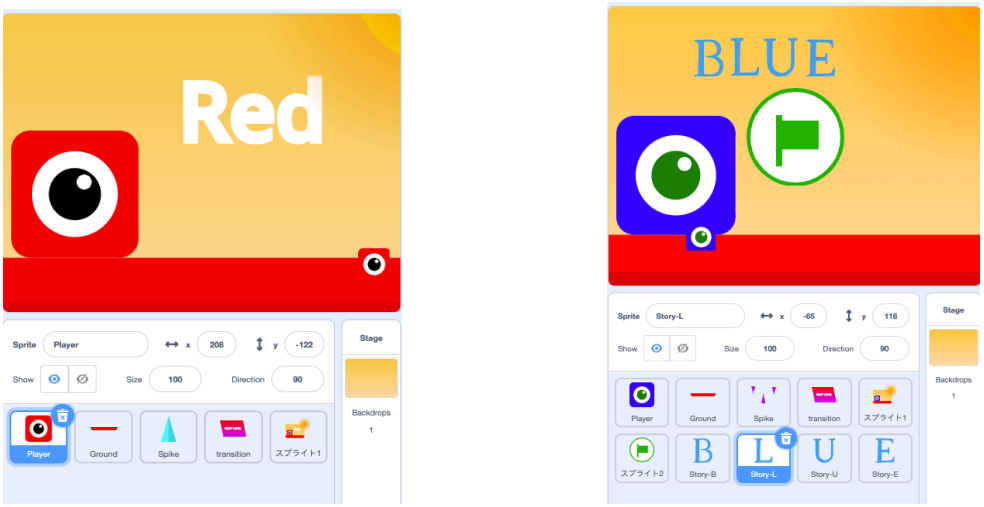

Figure 3.4: Added more sprites in "BLUE" project

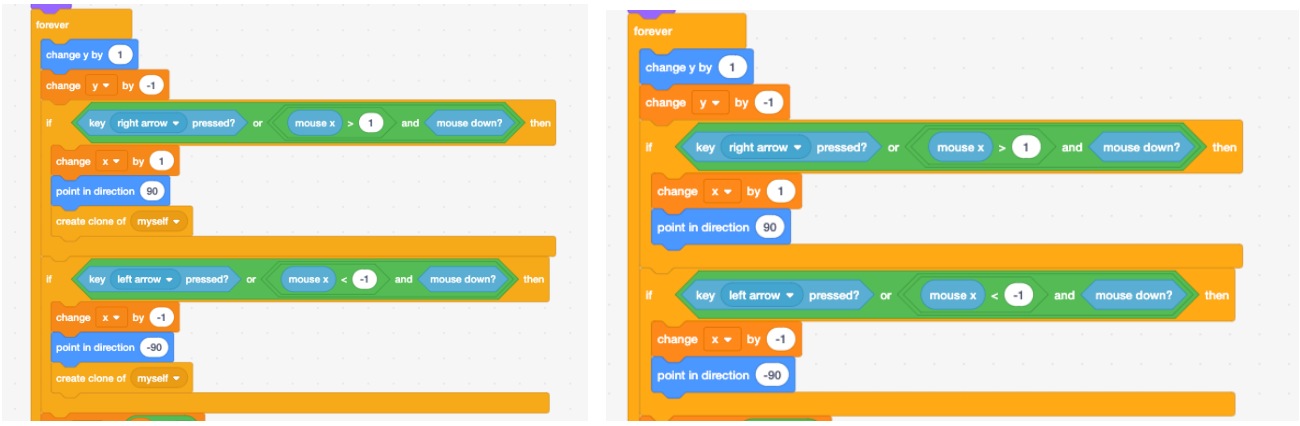

Figure 3.5: Code changes: delete clone-related blocks

The quality of code changes in remixes serves as a good indicator of how easy the original source code is to understand and modify. Therefore, a better understanding of how

[6] https://scratch.mit.edu/projects/491539511

programming idioms affect the code changes in the remixes can inform the design of educational interventions that seek to better support the introductory programming curriculum in this domain.

Chapter 4

Methods

In this section, we describe the approaches we used to answer our research questions.

4.1 Identifying programming idioms

To ensure that our idioms reflect the commonly used idioms in Scratch, we systematically documented a catalog of programming idioms by analyzing the technical learning materials provided for both the latest version of Scratch (3.0) and the previous version (2.0). We assume that learning materials are written by language experts, so the provided code samples should be idiomatic and of code quality. Specifically, in our analysis we used the following learning materials as our primary sources: "Scratch 3 Programming Playground [33]" , "Learn to Program with Scratch [27]" , "Make Your Own Scratch Games! [14]" , "Hello Scratch!: Learn to program by making arcade games [19]" , and "Scratch by Example: Programming for All Ages [35]".

The first and the second authors systematically studied the learning resources above. Each identified recurring programming idiom was evaluated as a candidate to be included into our list of idioms. The included idioms were divided into two categories: *Scratch-specific* and *general programming* idioms. *Scratch-specific* idioms come from usages that perform specific media-computing functionalities, such as animation, art, games, music, stories. For each programming idiom identified, we consulted a crowd-sourced, online catalog of programming

idioms curated for major text-based languages [3]. We classified any idioms that also appear in other programming languages as *general programming* idioms.

4.2 Studying the prevalence of Scratch programming idioms

To assess the prevalence of programming idioms, we collected a large representative sampling of Scratch projects from the "trending" category. We obtained our list of trending projects via the project fetching API that allows specifying different query parameters (e.g., https://api.scratch.mit.edu/explore/projects?limit=16&offset=0&language=en&mode=trending&q=animation). We chose to focus on trending projects, as they tend to be more mature in comparison to the projects in the "recent" category. Trending projects are also likely to be non-trivial, as indicated by their high visibility among community members, thus making it possible to exclude projects that contain only graphical media without any code.

To better understand how idioms are used, we collected additional project samples of different demographic groups to compare with the average trending projects. Specifically, we focused on two specific demographics 1) Scratch studios, comprising projects created by students in formal classroom settings and 2) Top community-favorite programmers, comprising projects created by highly experienced Scratch programmers in the community. After removed the empty projects (i.e., zero block found in projects), overall, we collected a total of 70K projects during March, 2021 (29,771 projects for top Scratchers, 43,340 projects for trending, 730 projects for studio).

For RQ3, we used a small subset of Scratch projects that have been remixed (forked) multiple

times and contained two common control-flow idioms: FOREVER IF/IF ELSE and NESTED IF ELSE. Because the analysis required comparing the original projects with their remixes, we collected a random sample of twenty remixes for each studied original project. Not all of twenty remixes meet the criteria. In summary, the RQ3 dataset comprises a total of 342 projects (59 original projects and their 283 remixes).

Table 4.1: Basic summary statistics of dataset of over trending.

Statistic	N	Mean	St. Dev	Min	Pctl(25)	Median	Pctl(75)	Max
numBlock	43,340	374.62	966.50	1	25	99	347	28031
numProcedure	43,340	2.77	9.94	0	0	0	1	285
numScript	43,340	29.13	65.68	0	4	10	29	2314
numSprite	43,340	6.77	10.01	0	2	4	8	332

4.3 Detecting programming idioms

We extended LitterBox [20] to implement a set of analysis routines, each detecting a specific programming idiom. An idiom detection routine operates on the AST of a parsed Scratch project. It traverses the AST, collects the information needed to determine whether the idiom is present based on the idiom's definition, and records any detected instances in a detection report.

4.4 Computing programming idioms metrics

To assess the prevalence of an idiom, we calculated the percentage of the projects in the dataset that contains at least one instance of the idiom. We performed this calculation for the projects in different project categories (e.g., games, animations, storytelling, etc.) and three sample datasets representing three different demographics (i.e., trending (general), top

Scratchers, and studio projects).

We explored how code changes within the body of the Forever IF/IF Else and Nested IF Else idioms across the original projects and their remixes. We detected these idioms in the original projects and then collected their block IDs and identified the corresponding code fragments in their remixes. We converted the code fragments to textual representations to make them amenable to differencing as plain text. Then, we calculated the type of changes in terms of deletion, insertion, and update, represented as percentages of each operation.

Chapter 5

Findings

In this section, we present the findings that answer our research questions.

5.1 Common programming idioms in Scratch (RQ1)

Identified by following the procedure outlined above, the following idioms are cataloged into Scratch-specific and general programming groups.

5.1.1 Scratch-specific Programming Idioms

1. Change And Clone: A Scratch clone inherits attribute values (e.g., local variable, position, visibility, etc.) from a sprite parent, which can be viewed as the clone's prototype. This idiom initializes the attributes of a soon-to-be created clone and immediately creates it. The idiom contains one or more side-effect command blocks (e.g., `change [variable v] by (1)`) that precede `create clone of [myself]` block.

2. Change And Wait: This idiom introduces delay in between command blocks that have immediate effects on the program output (e.g., setting an object's position). Executing these blocks in sequence would result in instant changes that are imperceptible to the human eye. To create an illusion of smooth changes over a period of time (e.g., a moving object), a wait block (e.g., `wait 0.01 sec`) is used to insert a small delay between each

included block (or a sequence of blocks that needs to be executed without delays) [19].

3. Repeat And Change: Scratch projects are often made up of several low-level, elementary animated elements that can be combined to create increasingly complex animations, often used by storytelling and game projects. This idiom is used in many elementary animations (e.g., fading a sprite's transparency, growing a sprite's size) by repeating a sequence of side-effect causing blocks (e.g., position, size, graphic effects) a specified number of times.

4. Sensor Wait Until No Sensor: To handle user input events, an infinite loop (i.e., `forever` block) is often used alongside *sensor* blocks to monitor specific user input events. However, using an `if` block to check whether a user input event (e.g., mouse clicks, key presses) is present can trigger event-based code more than once at a time, resulting in an unwanted program behavior in some scenarios. This idiom ensures that a stream of events of the same type will not re-trigger event-based code by using a `wait until [sensor block]` block to wait for the absence of the event first. Fig. 5.1 shows an example of this idiom.

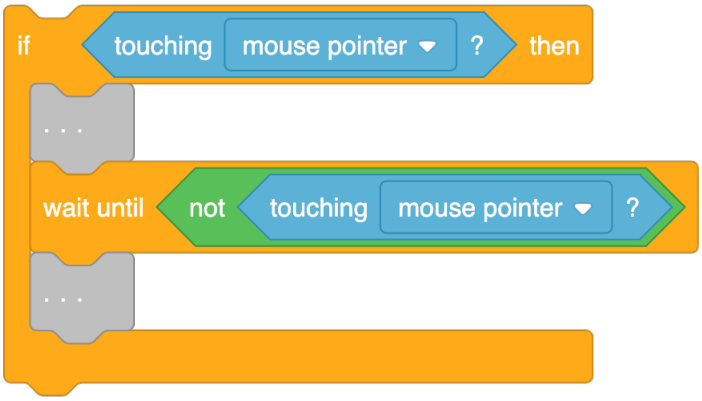

Figure 5.1: Sensor Wait Until No Sensor

5. SWITCH BACKDROP AND BROADCAST: The Stage serves not only as a parent object for all sprites in a project, but also as a visual container that renders the background (called a backdrop in Scratch). This idiom sets a backdrop as the visual context, and then coordinates sprites accordingly via a broadcast block.

6. BROADCAST AND STOP: This idiom diverts program flow from one script to other scripts when a certain condition is met. This idiom coordinates various parts of a program in response to a specific program state. Fig. 5.2 shows a common use case of this idiom. A script handles an exceptional case in which an error message is broadcast and the execution of that script is stopped.

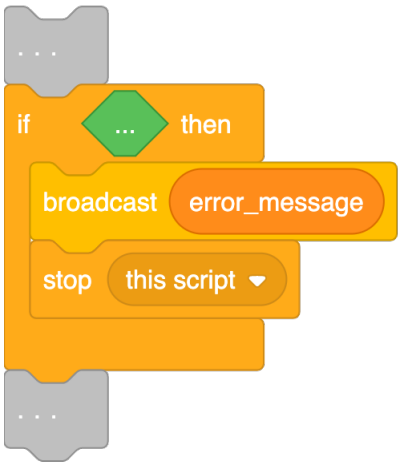

Figure 5.2: BROADCAST AND STOP

5.1.2 General programming idioms

7. ITERATE LIST: Similarly as in other programming languages, a *list* is a useful data structure for storing multiple pieces of information. Scratch provides a set of basic command blocks for reading a list value at a given index as well as manipulating the

stored values (e.g., adding and deleting item by its value from the list). This idiom is used to iterate the list values and perform actions on each accessed value [2].

8. Delete List Element By Value: Scratch supports storing in a list values of string and number types only. Additionally, all operations on a list are based on either the item indices or the item values. This specific list-based idiom removes all list items that matches a specified item value [4]. Fig. 5.3 shows the generic form of this idiom.

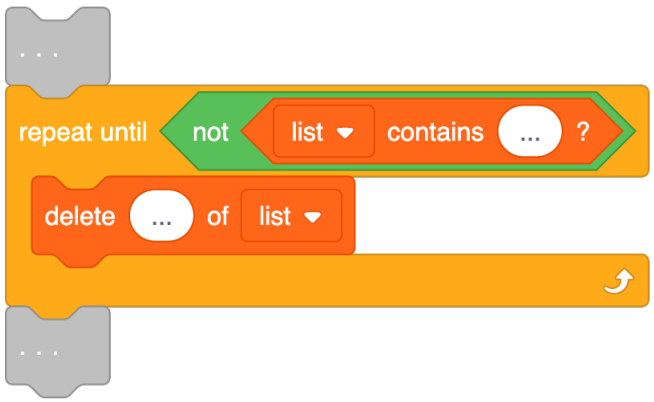

Figure 5.3: Delete List Element By Value

9. Forever IF/IF Else: This idiom provides a way to continuously monitor a given Boolean condition [27]. For *if-then* inside the body of *forever* block, it perform no action when the condition is false and perform specified actions whenever the condition is true. The need for this usage scenario is so common that historically Scratch introduced it as one of its control blocks but later removed it in Scratch 2.0 [1]. A less common variation of this idiom, when *if-then-else* block is used in place of *if-then* block, provides a way to also perform specific actions continuously when the monitored condition is not met.

10. Nested IF Else: Selecting one among multiple code fragments to execute based on their conditions is a common programming idiom in any language [27]. Some programming languages allows multiple *else if* to be inserted between *then* and *else* parts to specify

additional conditions. In Scratch, this multi-branch control structure is achieved by repeatedly nesting the *if-then-else* block within the *else* part of the previous *if-then-else* block. Fig. 5.4 shows an example of this idiom.

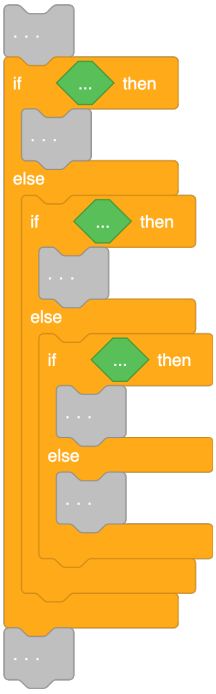

Figure 5.4: NESTED IF ELSE

11. REPEAT ASK: Scratch provides the `ask () and wait` block to prompt for text input from the user [27]. The most recent text input is then stored in the "answer" block. This idiom performs input validation by continuously asking for user input until a valid text input is submitted [27]. Fig.5.5 shows an example of this idiom.

5.2 Prevalence of programming idioms (RQ2)

Table 4.1 provides the statistical summary of the *trending* dataset. We calculate the mean, standard deviation, and five-number summary of the basic Scratch program elements (blocks,

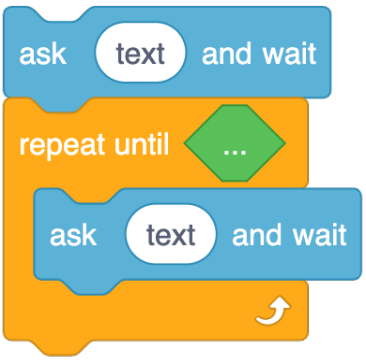

Figure 5.5: Repeat Ask

Table 5.1: Prevalence of programming idioms in trending, top Scratchers and studio datasets

Programming idiom	Trending	Top Scratchers	Studio
Change And Wait	60.23%	58.43%	64.79%
Repeat And Change	58.31%	63.96%	42.19%
Forever IF/IF Else	39.79%	40.86%	52.88%
Change And Clone	17.54%	20.48%	5.75%
Nested IF Else	10.69%	12.68%	5.75%
Broadcast And Stop	2.86%	3.82%	4.25%
Switch Backdrop And Broadcast	2.52%	3.01%	9.18%
Sensor Wait Until No Sensor	1.14%	1.72%	0
Delete List Element By Value	0.10%	0.14%	0
Repeat Ask	0.05%	0.08%	0.27%
Iterate List	$< 0.01\%$	0.03%	0

procedures, scripts, and sprites) in 43,340 project samples. The size of the project samples in terms of number of blocks varies widely, as indicated by a high standard deviation of *numBlocks*. On average, a project in our dataset contains 374 blocks, 29 scripts, and 7 sprites.

Table 5.1 presents the prevalence of each programming idiom in three datasets: *trending*, *top Scratchers*, and *studio projects*.

Overall, each programming idiom is similarly prevalent across the three datasets. The top three most prevalent idioms are Change And Wait, Repeat And Change, and Forever IF/IF

Table 5.2: Prevalence of programming idioms in each project category subset within the trending dataset

Programming idiom	Animations	Art	Games	Music	Stories	Tutorials
Change And Wait	76.62%	41.45%	65.50%	48.46%	73.26%	51.76%
Repeat And Change	70.38%	42.51%	70.67%	49.08%	68.60%	46.46%
Forever IF/IF Else	26.92%	23.55%	74.44%	32.18%	27.11%	53.18%
Change And Clone	19.70%	7.71%	30.51%	13.41%	21.92%	11.27%
Nested IF Else	9.20%	3.15%	22.87%	6.21%	14.11%	7.87%
Broadcast And Stop	1.78%	0.80%	9.43%	2.27%	0.96%	2.30%
Switch Backdrop And Broadcast	3.66%	1.06%	2.78%	1.7%	3.31%	2.24%
Sensor Wait Until No Sensor	0.82%	0.39%	3.18%	0.74%	0.92%	0.85%
Delete List Element By Value	0.03%	0.05%	0.43%	0.06%	0.01%	0.05%
Repeat Ask	0	0	0.21%	0.01%	0.06%	0.01%
Iterate List	0	0	0	0	0.01%	0

ELSE, respectively. Each of these three idioms are present in over one third (33%) of project samples. Notably, the top two idioms are often present in more than half of the project samples.

Moderately prevalent idioms are CHANGE AND CLONE and NESTED IF ELSE. These idioms are present from 9% to 21% of the project samples. Broadcast-related idioms (i.e., SWITCH BACKDROP AND BROADCAST and BROADCAST AND STOP) are less prevalent, with their presence around 2%-4% of the project samples. Finally, list-related idioms (i.e., DELETE LIST ELEMENT BY VALUE, ITERATE LIST) and REPEAT ASK are uncommon, detected in less than 0.3% of the sample projects in all three datasets.

To better understand the distribution of programming idioms, we further investigated how prevalent the identified idioms are in the trending dataset across different project categories. Table 5.2 presents the results. CHANGE AND WAIT and REPEAT AND CHANGE, commonly used idioms for creating animation, are similarly prevalent across different project categories, especially in animations, games, and stories. These idioms were detected in a range of 41%

to 77% of the projects. The prevalence of Forever IF/IF Else varies noticeably across project categories. Specifically, less than 28% of the animation, art, and storytelling projects contain this idiom, while 74.44% of games and 53.18% tutorials projects contain this idiom at least once. Especially, those two idioms are less than 10% in the art category, and they show 3.15% for Nested IF Else and 7.71% for Change And Clone.

Broadcast-related idioms (Switch Backdrop And Broadcast and Broadcast And Stop) appear in less than 10% of projects in each category. We found that only Broadcast And Stop in game projects is 9.43%, which is higher than Broadcast-related idioms in other project categories. The idioms related to the list data structure (i.e., Iterate List) are uncommon, found only in stories, appearing in only 0.01% of all projects. Other uncommon idioms include Sensor Wait Until No Sensor, and Repeat Ask whose prevalence is less than 4% across project categories, used in less than 1% of all projects.

5.3 Idioms and code changes through remixing (RQ3)

We explored how code changes within the body of two common control-flow idioms. We considered three types of operations: block insertion, block deletion, and value update.

Fig. 5.6 shows a partial example of the analyzed code changes. The original and its remixed instance of the Forever IF/IF Else idiom appear on the left and right sides, respectively. This example shows how code change in two places. The first change is the insertion of two more blocks within the "if-then" block. The other change is the modified Boolean expression of the existing "if" block, with extra operator blocks that include several sensing-related blocks. Our analysis captures these changes as 3 insertions.

We analyzed a total of 367 instances of Nested IF Else and 739 instances of Forever IF/IF

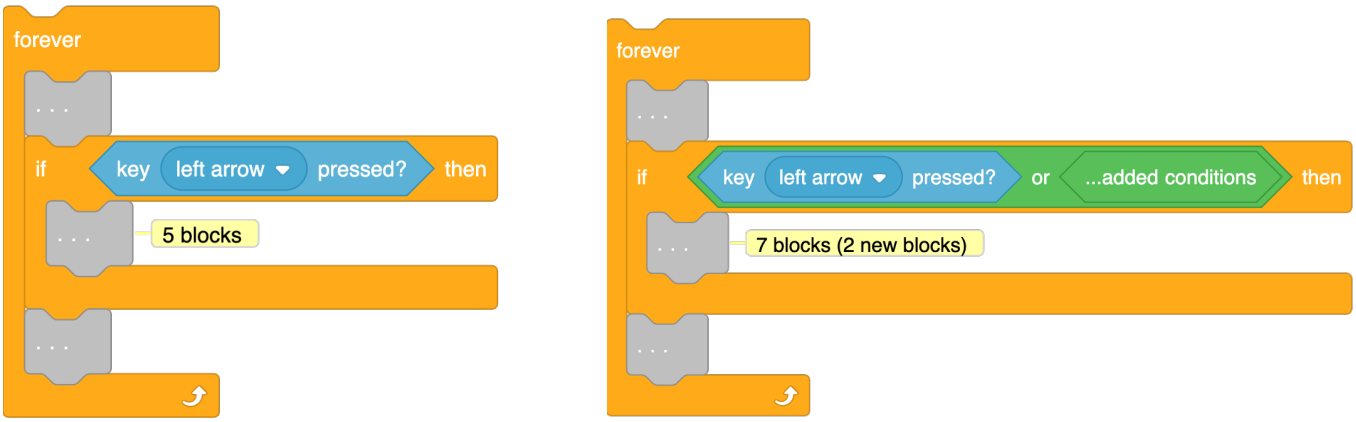

Figure 5.6: Insertion example for FOREVER IF/IF ELSE in original and its remix

ELSE. The two programming idioms have dissimilar characteristics of code changes across the original projects and their remixes. Tables 5.3 and 5.4 summarize the change operations that occurred within the body of the analyzed idiom instances. The results on both of these tables show that the majority of code changes are deletions with ∼82% for NESTED IF ELSE and ∼54% for FOREVER IF/IF ELSE. Although the percentages of deletions in both idioms are similarly high, the percentages of insertions are markedly different. Block insertions account for almost 27% of all code changes in FOREVER IF/IF ELSE as compared to about 5% in NESTED IF ELSE.

Table 5.3: Prevalence of operations found in NESTED IF ELSE

Operation	% of change of line
Delete	81.66%
Update	13.58%
Insert	4.76%

Table 5.4: Prevalence of operations found in Forever IF/IF Else

Operation	% of change of line
Delete	53.95%
Insert	26.55%
Update	19.50%

Chapter 6

Discussion

In this section, we discuss how the prevalence of programming idioms and code changes in the remixes may be leveraged by computing learners and educators.

6.1 Identifying Scratch Idioms

Similarly to prior approaches, we found the task of identifying common Scratch programming idioms quite challenging. What comprises an idiom can be a highly subjective question. The issue at hand is to determine the prevalence threshold, exceeding which an idiom would be considered for inclusion. Some idioms prominently featured in language tutorials may not manifest themselves prominently in actual application codebases. In fact, a related work effort reports a similar observation regarding a different set of idioms [10]. Their results show that the top three recurring patterns appear in only 19% to 34% of over 200k project samples, while the rest of the patterns appear in less than 10% of the samples.

This insight suggests that the Scratch application codebase is rife with non-idiomatic coding practices. One possible explanation for this insight is that Scratch programmers tend to code in more experimental styles typical of bottom-up programming practices, more concerned with the end result rather than with the specifics and quality of their code.

This phenomenon also reflects how workarounds similar to those documented in Scratch's

learning resources [6] may be commonly found among projects created by novice programmers. To identify the common programming idioms as they pertain to actual programming practices, one might explore applying the software mining techniques [9] on the actual application codebases rather than systematically consulting textbooks and tutorials.

6.2 Novice Programming Practices with Programming Idioms

Our findings suggest that novice programmers can successfully recognize and make use of programming idioms. Some idioms are highly prevalent (i.e., Change And Wait, Repeat And Change, and Forever IF/IF Else), while several others (e.g., Iterate List, Repeat Ask, etc.), not so much. The low prevalence of many idioms may be due to their more specific use cases, and advanced usage scenarios. Nevertheless, the presence of the observed highly recurring idioms suggests that novice programmers, similar to professional programmers, find idioms natural to learn and apply in their coding practices. A similar observation has been discussed in prior work about the naturalness of event-based programming styles among Scratch novice programmers [22].

Although highly recurring idioms are applicable regardless of the project categories, many of the studied idioms are often domain-specific. Any Scratch project has at least some graphics, making animation-related idioms (i.e., Change And Wait, Repeat And Change) highly prevalent. However, some idioms are more prevalent within certain project categories. For example, Forever IF/IF Else in the games and tutorials category represent more than 50% of all projects and tend to be less common in the art category (∼23%).

Tutorial projects present an interesting case, as they seem to reflect the average of all cat-

egories. The likely impetus for creating these projects is the natural desire of programmers to share their knowledge and expertise with their fellow programmers. It is perhaps for this reason, that tutorial projects often contain the idioms commonly used across all categories (e.g., art, game, storytelling) that we studied.

6.3 Programming Idioms and Introductory Computing

The low-prevalent idioms suggest possible opportunities for educational interventions. For example, the fact that ITERATE LIST is not commonly present suggests that list data structures are underused in the application codebase. Perhaps introductory learners are unaware of this data structure and its common applications. Hence, it might be more effective to teach programming constructs in terms of their relevant idioms, so learners can quickly start becoming familiarized with the usage of programming constructs and apply them appropriately in their programming practices. In the case of the list data structure, educators can more explicitly introduce the list idioms as part of tutorials and sample projects to promote the usage of this important data structure in introductory programming. Similarly, REPEAT ASK, used for validating text, a common idiom in other programming languages for securing the program input. Perhaps better familiarity is all that is required for not only using the *ask* and *answer* blocks, but also for the practice of validating input becoming a standard tool for Scratch programmers.

6.4 Remixing Practices with Programming Idioms

Khawas et al.[25] explored the overall changes between the projects and their remixes. They found that programmers often insert blocks than delete them. However, the insertions exceed

deletions only by a small margin. In our work, we also made it a point to study how the code fragment within the idiom's body changes between projects and their remixes. Specifically, we determined that in two common flow control idioms, programmers more frequently delete than insert code blocks in the remixes (i.e., 81.66% in Nested IF Else, 53.95% in Forever IF/IF Else).

From Tables 5.4 and 5.3, the insertion in Nested IF Else is lower than in Forever IF/IF Else, and the potential reason for this discrepancy is that Nested IF Else contains specific operations in potential scenarios, which programmers usually keep or delete as a whole. The Forever IF/IF Else idiom allows more editing flexibility, so programmers tend to actually insert code blocks into this idiom. The dissimilar percentages of code insertions between the two common flow-control idioms raise an interesting question: do these differences correlate with the degree of cognitive effort required to understand each idiom?

As we determined, the majority of changes for these two idioms in the remixes are deletions. This observation suggests that programmers remix projects by retaining existing control structures, from which they replace much of the contained code blocks. Driven by this observation, one may consider providing students with generic forms or examples of common idioms (i.e., skeletal idiom code). Then one can guide students to fill in the necessary logic, so as to more effectively master these idioms. This observation may present opportunities for educators and designers of language learning environments.

Chapter 7

Threats to validity

The validity of our analysis results may be endangered by a few factors. The programming idioms we documented and studied are limited by the Scratch programming materials available at the time of the study. New prevailing idioms may be uncovered with any changes in the language features, the community's programming practices or educational interventions in this language.

To ensure the accuracy of our idiom detection and to avoid introducing false positives, we implemented a suite of test cases for each detection algorithm. We also implemented more generalized detectors to detect idioms with certain variants.

For investigating changes within idioms, we only select two most common control flow idioms. Hence, our findings may not be representative of the actual changes across all common idioms introduced by programmers in their remixes.

Chapter 8

Related work

In this section, we review the most closely related prior research efforts.

Our study of programming idioms in Scratch is among many prior research efforts that aim at better understanding and supporting programmers in this domain. Related works in this area apply program analysis to Scratch, such as the studies of code smells [34] and software bugs [20]. A common approach to automatically analyze coding patterns is to leverage static program analysis, which operates on the abstract syntax tree (AST) representation of a program. Like programming idioms, bug patterns and code smells exhibit unique structural characteristics that can be exploited by automatic detection approaches. Although designed for different purposes, some of the prior works provide open-source reusable program analysis infrastructures to build upon. In particular, in our research, we took advantage of LitterBox [20], a state-of-the-art Scratch program analysis framework. We were able to extend its core functionality to implement our analyzer of programming idioms.

Several prior works have also focused on programming idioms in Scratch. In particular, Amanullah and Bell conducted a series of studies of "elementary programming patterns," a concept similar to our "programming idioms." Their early work proposed two main categories of elementary programming idioms as a way to help students avoid programmings problems (e.g., code smells, bug patterns) and assess the prevalence of these patterns in a large dataset of Scratch projects [10]. These findings show that elementary patterns are uncommon among projects created by inexperienced programmers [11]. Their later work suggests that remixing

can help programmers learn elementary patterns. Specifically, when instances of elementary pattern were detected in an original project, the same pattern instances were also found, often in greater number, in its project remixes [12]. Their most recent work applies elementary programming patterns to measure how comprehensive a set of Scratch teaching materials are in covering their patterns [13].

Our work differs from the works conducted by Amanullah and Bell in a number of ways.

Firstly, our goal is to comprehensively document common Scratch programming idioms, as based on a representative corpus of canonical educational materials. Specifically, we draw our idioms from popular Scratch educational materials, considering not only general programming but also Scratch-specific idioms.

Secondly, our large-scale study uses several dissimilar sets of projects to better understand existing idioms usage patterns. Specifically, we applied our analysis to projects of different demographics (i.e., trending, studios, top Scratch programmers) and categories (i.e., games, animations, etc.).

Finally, we investigate the types of changes in the idioms' body between projects and their remixes. Our case study can thus shed light on how these idioms encourage remixing, with the goal of promoting the Scratch communal learning.

Chapter 9

Conclusions and future work

Our work sheds light on common Scratch programming idioms. Our large-scale study experiment assesses not only the prevalence of Scratch programming idioms, but also how programmers tend to change code within two common idioms between the remixed projects and their sources. The results of our work identify common programming idioms, a piece of knowledge that can benefit novice programmers as a way to help them learn the language faster. The net effect would be promoting effective programming practices among introductory learners, writing idiomatic high quality code.

Possible future work directions include exploring common programming idioms in this domain from the perspective of novice programmers as well as applying software mining and natural language techniques to extract idioms from existing application codebases [9].

Bibliography

[1] Forever if (). https://en.scratch-wiki.info/wiki/Forever_If_()_(block).

[2] Example for iterating the list. https://programming-idioms.org/idiom/6/iterate-over-list-values.

[3] Online programming idioms. https://programming-idioms.org/.

[4] Example for removing elements from the list. https://programming-idioms.org/idiom/6/iterate-over-list-values.

[5] Scratch's statistics. https://scratch.mit.edu/statistics/, .

[6] List of block workarounds. https://en.scratch-wiki.info/wiki/List_of_Block_Workarounds, .

[7] 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*, pages 53–61, 2016.

[8] Carol V Alexandru, José J Merchante, Sebastiano Panichella, Sebastian Proksch, Harald C Gall, and Gregorio Robles. On the usage of pythonic idioms. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 1–11, 2018.

[9] Miltiadis Allamanis and Charles Sutton. Mining idioms from source code. In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering*, pages 472–483, 2014.

[10] Kashif Amanullah and Tim Bell. Analysing students' scratch programs and addressing issues using elementary patterns. In *2018 IEEE Frontiers in Education Conference (FIE)*, pages 1–5. IEEE, 2018.

[11] Kashif Amanullah and Tim Bell. Analysis of progression of scratch users based on their use of elementary patterns. In *2019 14th International Conference on Computer Science & Education (ICCSE)*, pages 573–578. IEEE, 2019.

[12] Kashif Amanullah and Tim Bell. Evaluating the use of remixing in scratch projects based on repertoire, lines of code (loc), and elementary patterns. In *2019 IEEE Frontiers in Education Conference (FIE)*, pages 1–8. IEEE, 2019.

[13] Kashif Amanullah and Tim Bell. Teaching resources for young programmers: the use of patterns. In *2020 IEEE Frontiers in Education Conference (FIE)*, pages 1–9. IEEE, 2020.

[14] A. Anthropy. *Make Your Own Scratch Games!* No Starch Press, 2019. ISBN 9781593279363. URL https://books.google.com/books?id=QVv6DwAAQBAJ.

[15] Michael J Clancy and Marcia C Linn. Patterns and pedagogy. *ACM SIGCSE Bulletin*, 31(1):37–42, 1999.

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

[17] Jinan AW Faidhi and Stuart K Robinson. An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Computers & Education*, 11(1):11–19, 1987.

[18] Deborah A. Fields, Michael Giang, and Yasmin Kafai. Programming in the wild: Trends in youth computational participation in the online scratch community. In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*, WiPSCE '14, pages 2–11. Association for Computing Machinery. ISBN 978-1-4503-3250-7. doi: 10.1145/2670757.2670768.

[19] G. Ford, M. Ford, and S. Ford. *Hello Scratch!: Learn to Program by Making Arcade Games*. Manning Publications, 2017. ISBN 9781617294259. URL https://books.google.com/books?id=Te4jvgAACAAJ.

[20] Christoph Frädrich, Florian Obermüller, Nina Körber, Ute Heuer, and Gordon Fraser. Common bugs in scratch programs. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, pages 89–95, 2020.

[21] Joseph Gil and David H Lorenz. Design patterns vs. language design. In *European Conference on Object-Oriented Programming*, pages 108–112. Springer, 1997.

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

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

[24] Paul A hner, John Sweller, and Richard E Clark. Why minimal guidance during instruction does not work: An analysis of the failure of constructuvist, discovery, problem-based, experimental, and inquiry-based teaching. *Educational Psychologist*, 42(2), 2006.

[25] Prapti Khawas, Peeratham Techapalokul, and Eli Tilevich. Unmixing remixes: The how and why of not starting projects from scratch. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 169–173. IEEE, 2019.

[26] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):1–15, 2010.

[27] M. Marji. *Learn to Program with Scratch: A Visual Introduction to Programming with Games, Art, Science, and Math*. Informatica (E-libro–2014/09). No Starch Press, 2014. ISBN 9781593275433. URL https://books.google.com/books?id=sYvlAwAAQBAJ.

[28] Eugene W Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1 (1-4):251–266, 1986.

[29] Karl J Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *ACM Sigcse Bulletin*, 8(4):30–41, 1976.

[30] Howard A Peell. An apl idiom inventory. In *Proceedings of the international conference on APL: APL in transition*, pages 362–368, 1987.

[31] Alan J Perlis and Spencer Rugaber. Programming with idioms in apl. *ACM SIGAPL APL Quote Quad*, 9(4-P1):232–235, 1979.

[32] Michael Smit, Barry Gergel, and H James Hoover. Code convention adherence in evolving software. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 504–507. IEEE, 2011.

[33] A. Sweigart. *Scratch 3 Programming Playground: Learn to Program by Making Cool Games*. No Starch Press, 2021. ISBN 9781718500228. URL https://books.google.com/books?id=L967DwAAQBAJ.

[34] 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)*, pages 43–51. IEEE, 2017.

[35] E.A. Vlieg. *Scratch by Example: Programming for All Ages*. Apress, 2016. ISBN 9781484219461. URL https://books.google.com/books?id=kxoPDQAAQBAJ.

[36] David Weintrop and Uri Wilensky. To block or not to block, that is the question: students' perceptions of blocks-based programming. In *Proceedings of the 14th international conference on interaction design and children*, pages 199–208, 2015.

Appendices

Appendix A

Example for obtaining Scratch projects

In our work, we assessed the prevalence of programming idioms with a large, diverse dataset of over 70k Scratch projects. The following code snippets were used to automate our data collection process. We consulted an unofficial documentation for Scratch API usages [1] when specifying queries to fetch specific types of Scratch projects.

```
import fetch from 'node-fetch';
import fs from 'fs';
import { sleep, findFilesByExt } from './utils.js';
import * as child_process from 'child_process';
import { DOMParser, XMLSerializer } from 'xmldom';
import HashMap from 'hashmap';
import lineByLine from 'n-readlines';

const getProjectQueryUrl = function (limit, offset, mode, q) {
   return `https://api.scratch.mit.edu/explore/projects
      ?limit=${limit}&offset=${offset}&language=en&mode=${mode}&q=${q}`;
};
```

[1] http://towerofnix.github.io/scratch-api-unofficial-docs/

```
const getStudioQueryUrl = function (limit, offset, mode, q) {
    return `https://api.scratch.mit.edu/explore/studios
    ?limit=${limit}&offset=${offset}&language=en&mode=${mode}&q=${q}`;
};

// https://scratchdb.lefty.one/v2/user/rank/global/loves/1
const getUsersURL = function (country, type, page) {
    return `https://scratchdb.lefty.one/v2/user/rank/
    ${country}/${type}/${page}`;
}

const getProjectsFromUserURL = function (username, limit, offset) {
    return `https://api.scratch.mit.edu/users/${username}/projects
    ?limit=${limit}&offset=${offset}`;
}

const getProjectIdFromStudioURL = function (studioId, limit, offset) {
    return `https://api.scratch.mit.edu/studios/${studioId}/projects
    ?limit=${limit}&offset=${offset}`;
}

// mode: {trending, recent, popular}, q={animation, art, games, music, stories,
    tutorials}
// 30, trending, all different categories.
const retrieveProjects = async function (projectEntryStartOffset, numProjects,
    mode, q) {
    const limit = 10; // 10 projects per request
```

```
let projectList = [];
let projectCount = 0;
// offsets calculation
let endOffset = Math.ceil(numProjects / limit);

for (let offset = 0; offset < endOffset; offset++) {
    let adjustedOffset = (projectEntryStartOffset + offset * limit);
    var projects;
    try {
        projects = await fetch(getProjectQueryUrl(limit, adjustedOffset, mode,
            q)).then((response) => response.json());
    } catch (error) {
        console.log(error);
    }
    await sleep(500);
    // skip void projects.
    if ('code' in projects)
        continue;
    for (const p of projects) {
        projectList.push(p.id);
        projectCount++;
        if (projectCount >= numProjects) {
            return projectList;
        }
    }
}
console.log('retrieving successfully');
```

```
    return projectList;
}

const retrieveProjectsFromFamous = async function (country, type, page,
    numProjects) {
    const limit = 10; // 10 projects per request
    const projectEntryStartOffset = 0;
    //  offsets calculation
    var endOffset = Math.ceil(numProjects / limit);
    var usersJson = await fetch(getUsersURL(country, type, page)).then((response)
        => response.json());
    var usersList = [];
    await sleep(500);

    for (const user of usersJson.users) {
        usersList.push(user.info.username);
    }
    for (const username of usersList) {
        var projectIdList = [];
        console.log("Collecting projects from username: " + username);
        for (let offset = 0; offset < endOffset; offset++) {
            let adjustedOffset = (projectEntryStartOffset + offset * limit);
            const projectIds = await fetch(getProjectsFromUserURL(username, limit,
                adjustedOffset)).then((response) => response.json());

            for (const project of projectIds) {
                projectIdList.push(project.id.toString());
```

```
            }
        }
        // console.log(projectIdList);
        console.log("# of id is " + numProjects + "# of id collected is " +
            projectIdList.length);
        const data = JSON.stringify(projectIdList);
        fs.writeFileSync('dataset_famous/' + page + "-" + username + "-" +
            projectIdList.length + ".json", data);
    }
}

const categories = ["animations"];
const projectEntryStartOffset = 0;
const numProjects = 100; // next projectEntryStartOffset;
const numStudios = 500; // 500 * 20 * 6
const splitNum = 50;
const mode = 'trending';

const downloadPorjectsFromExplore = async function (categories,
    projectEntryStartOffset, numProjects, splitNum, mode) {
    for (var start = projectEntryStartOffset; start < projectEntryStartOffset +
        numProjects; start += splitNum) {
        var json_data = {};
        var count = 0;
        for (const category of categories) {
            console.log("Downloading " + category + " projects");
            var projects;
```

```
            try {
                projects = await retrieveProjects(start, splitNum, mode, category);
            } catch (error) {
                console.log(error);
            }
            await sleep(500);

            json_data[category] = [];
            for (const id of projects) {
                count += 1;
                json_data[category].push(id.toString());
            }
        }

        console.log('Total # of projects = ' + (splitNum * categories.length) +
            ", # of collected projects = " + count);
        const data = JSON.stringify(json_data);
        const end = start + splitNum;
        fs.writeFileSync('dataset/dataset-' + mode + "-" + start + "-" + end +
            ".json", data);
    }
}
```

Appendix B

Example for idiom analyzer

To automatically detect programming idioms in Scratch programs, we extended LitterBox [20], a state-of-the-art program analysis tool originally developed to detect bug patterns in Scratch. The following code snippets document our implementations for detecting the studied programming idioms as well as their associated test cases.

```
// Detector for change and wait.
public class ChangeAndWait extends AbstractIssueFinder {
    private String NAME = "change_and_wait";

    @Override
    public void visit(IfThenStmt node) {
        List<Stmt> stmts = node.getThenStmts().getStmts();
        for (int i = 1; i < stmts.size(); i++) {
            if (checkChangeAndWait(stmts, i - 1, i)) {
                addToBlock(stmts, i - 1, i);
            }
        }
        visitChildren(node);
    }
```

```
@Override
public void visit(Script node) {
    List<Stmt> stmts = node.getStmtList().getStmts();
    for (int i = 1; i < stmts.size(); i++) {
        if (checkChangeAndWait(stmts, i - 1, i)) {
            addToBlock(stmts, i - 1, i);
        }
    }

    visitChildren(node);
}

@Override
public void visit(RepeatForeverStmt node) {
    List<Stmt> stmts = node.getStmtList().getStmts();
    for (int i = 1; i < stmts.size(); i++) {
        if (checkChangeAndWait(stmts, i - 1, i)) {
            addToBlock(stmts, i - 1, i);
        }
    }
    visitChildren(node);
}

@Override
public void visit(RepeatTimesStmt node) {
    List<Stmt> stmts = node.getStmtList().getStmts();
    for (int i = 1; i < stmts.size(); i++) {
```

```
        if (checkChangeAndWait(stmts, i - 1, i)) {
            addToBlock(stmts, i - 1, i);
        }
    }
    visitChildren(node);
}

@Override
public void visit(UntilStmt node) {
    List<Stmt> stmts = node.getStmtList().getStmts();
    for (int i = 1; i < stmts.size(); i++) {
        if (checkChangeAndWait(stmts, i - 1, i)) {
            addToBlock(stmts, i - 1, i);
        }
    }
    visitChildren(node);
}

private boolean isValid(Stmt stmt) {
    return stmt instanceof ChangeVariableBy || stmt instanceof
        SpriteMotionStmt || stmt instanceof SpriteLookStmt || stmt instanceof
        ActorLookStmt;
}

private boolean checkChangeAndWait(List<Stmt> stmts, int i, int j) {
    return isValid(stmts.get(i)) && (stmts.get(j) instanceof WaitSeconds);
}
```

```
    private void addToBlock(List<Stmt> stmts, int i, int j) {
        BlockMetadata ChangeBlock = stmts.get(i).getMetadata();
        NonDataBlockMetadata ChangeBlockData = (NonDataBlockMetadata) ChangeBlock;
        addBlock(ChangeBlockData.getBlockId());

        BlockMetadata waitBlock = stmts.get(j).getMetadata();
        NonDataBlockMetadata waitBlockData = (NonDataBlockMetadata) waitBlock;
        addBlock(waitBlockData.getBlockId());
    }

    @Override
    public String getName() {
        return NAME;
    }
}

// Test cases for corresponding idiom.
public class ChangeAndWaitTest implements JsonTest {
    @Test
    public void testEmptyProgram() throws IOException, ParsingException {
        Program empty = getAST("./src/test/fixtures/emptyProject.json");
        ChangeAndWait parameterName = new ChangeAndWait();
        Set<Issue> reports = parameterName.check(empty);
        Assertions.assertEquals(0, reports.size());
    }
```

```
    @Test
    public void testOneProgram() throws IOException, ParsingException {
        Program program = getAST("./src/test/fixtures/ChangeAndWaitOne.json");
        ChangeAndWait parameterName = new ChangeAndWait();
        List<String> blocks = parameterName.findBlocks(program);
        for (String block : blocks) {
            System.out.println(block);
        }
        Assertions.assertEquals(8, blocks.size());
    }

    @Test
    public void testTwoProgram() throws IOException, ParsingException {
        // Multiple Sprite.
        Program program = getAST("./src/test/fixtures/ChangeAndWaitTwo.json");
        ChangeAndWait parameterName = new ChangeAndWait();
        List<String> blocks = parameterName.findBlocks(program);
        for (String block : blocks) {
            System.out.println(block);
        }
        Assertions.assertEquals(24, blocks.size());
    }
}

// Detector for forever if.
public class ForeverIf extends AbstractIssueFinder {
    private String NAME = "forever_if";
```

```
    @Override
    public void visit(RepeatForeverStmt node) {
        List<Stmt> stmts = node.getStmtList().getStmts();
        for (int i = 0; i < stmts.size(); i++) {
            if (stmts.get(i) instanceof IfElseStmt || stmts.get(i) instanceof
                IfThenStmt) {
                addToBlock(node);
                break;
            }
        }
        visitChildren(node);
    }

    private void addToBlock(RepeatForeverStmt node) {
        BlockMetadata IfElseBlock = node.getMetadata();
        NonDataBlockMetadata IfElseBlockData = (NonDataBlockMetadata) IfElseBlock;
        addBlock(IfElseBlockData.getBlockId());
    }

    @Override
    public String getName() {
        return NAME;
    }
}

// Finder (ScratchVisitor) for NestedIfElse.
```

```
public class NestedIfElseFinder implements ScratchVisitor {
    public static final String NAME = "nested_if_finder";
    private IfElseStmt IfElseFound = null;
    private String blockId = null;

    public IfElseStmt find(Program program, String blockId) {
        Preconditions.checkNotNull(program);
        this.blockId = blockId;
        program.accept(this);
        return IfElseFound;
    }

    @Override
    public void visit(IfElseStmt node) {
        if (((NonDataBlockMetadata)
            node.getMetadata()).getBlockId().equals(blockId)) {
            IfElseFound = node;
        }
        visitChildren(node);
    }
}

// Find code fragments by using block ID.
 static void segementPrograms(CommandLine cmd) throws ParseException,
     IOException, ParsingException {
        if (!cmd.hasOption(PROJECTPATH)) {
            throw new ParseException("Input path option '" + PROJECTPATH + "'
```

```
        required");
    }
    if (!cmd.hasOption(BLOCK_ID)) {
        throw new ParseException("Block Id option '" + BLOCK_ID + "'
            required");
    }
    if (!cmd.hasOption(BLOCK_TYPE)) {
        throw new ParseException("Block id type option '" + BLOCK_TYPE + "'
            required");
    }

    String input = cmd.getOptionValue(PROJECTPATH);
    String blockId = cmd.getOptionValue(BLOCK_ID);
    String blockIdType = cmd.getOptionValue(BLOCK_TYPE);
    // System.out.println(blockId);
    Program program = getAST(input);
    AbstractNode found = null;
    if (blockIdType.equals("forever_if")) {
        ForeverIfFinder pFinder = new ForeverIfFinder();
        found = pFinder.find(program, blockId);
    } else {
        NestedIfElseFinder pFinder = new NestedIfElseFinder();
        found = pFinder.find(program, blockId);
    }
    ByteArrayOutputStream os = new ByteArrayOutputStream();
    PrintStream ps = new PrintStream(os);
    ScratchBlocksVisitor visitor = new ScratchBlocksVisitor(ps);
```

```
    visitor.ignoreScript();
    try {
        found.accept(visitor);
    } catch (Exception e) {
        System.out.println("");
        return;
    }
    String result = os.toString();
    System.out.println(result);
}
```

Appendix C

Example for generating a summary for dataset

To automatically process the analysis results, we implemented a Python program that aggregated the analysis results, used in subsequent interpretations in the Results section of this thesis. The following snippets document the implementation details of the program.

```
import os
import json
import numpy as np

def remove_empty_projects(input_path, empty_hold_path):
    for root, dirs, files in os.walk(input_path):
        for file in files:
            if file.endswith("_result.json"):
                file_path = os.path.join(root, file)
                try:
                    with open(file_path, "r") as f:
                        data = json.loads(f.read())
                    for item in data["metrics"]:
                        if item == 'block_count':
                            if int(float(data["metrics"][item])) == 0:
```

```
                        print("we are processing " + file)
                        os.rename(file_path, os.path.join(
                            empty_hold_path, file))
            except Exception as err:
                print(err)

def run_basic_summary(input_path):
    json_data = {"block_count": [], "procedure_count": [],
        "script_count": [], "sprite_count": []}
    for root, dirs, files in os.walk(input_path):
        for file in files:
            if file.endswith("_result.json"):
                file_path = os.path.join(root, file)
                try:
                    with open(file_path, "r") as f:
                        data = json.loads(f.read())
                    for item in json_data:
                        json_data[item].append(data["metrics"][item])
                except Exception as err:
                    print(err)

    for item in json_data:
        print("-" * 20 + item + "-" * 20)
        np_data = np.array(json_data[item]).astype("float")
        print("item = %s, len = %d " % (item, np_data.size))
        print("mean = %.2f, std = %.2f" % (np.mean(np_data), np.std(np_data)))
        print("min = %.2f, max = %.2f" % (min(np_data), max(np_data)))
```

```
        print("25 = %.2f, 50 = %.2f, 75 = %.2f" % (np.percentile(
            np_data, 25), np.percentile(np_data, 50), np.percentile(np_data, 75)))
        print("-" * (40 + len(item)))

def count_num_result_files(input_path):
    count = 0
    for root, dirs, files in os.walk(input_path):
        for file in files:
            if file.endswith("_result.json"):
                count += 1
    return count

def calculate_percentage_all(input_path):

    # datasets = ["output_dataset", "output_dataset_famous",
        "output_dataset_studio"]
    datasets = ["output_select_studio"]
    patterns = ["change_and_wait", "repeat_and_change", "forever_if",
        "change_and_clone", "nested_if_else",
    "broadcast_and_stop", "switch_backdrop_and_broadcast",
        "sensor_wait_until_no", "delete_element_from_list", "repeat_ask",
        "Iterate_list"]
    suffix = "summary_for_all.json"
    for item in patterns:
        row = [item]
        for type in datasets:
            dataset_path = os.path.join(input_path, type)
```

```
            total_files = count_num_result_files(dataset_path)
            summary_file_path = os.path.join(dataset_path, suffix);
            # print("-" * 10 + type + "-" * 10)
            # print("total files = " + str(total_files))
            try:
                with open(summary_file_path, "r") as f:
                    data = json.loads(f.read())
                if item not in data:
                    # print(item + " = 0 %")
                    continue
                else:
                    row.append(str(round((data[item] * 1.0 / total_files * 100),
                        2)) + " %")
                    # print('idioms = %s, num = %d, percentage = %.3f' % (item,
                        data[item], (data[item] * 1.0 / total_files * 100)))
            except Exception as err:
                print(err)
            # print("-" * (20 + len(type)))
        print("&".join(row))

def calculate_percentage_in_category(input_path, datasets):
    categories = ["animations", "art", "games",
        "music", "stories", "tutorials"]
    patterns = ["change_and_wait", "repeat_and_change", "forever_if",
        "nested_if_else", "change_and_clone",
    "broadcast_and_stop", "switch_backdrop_and_broadcast",
        "sensor_wait_until_no", "delete_element_from_list", "repeat_ask",
```

```
        "Iterate_list"]
    for item in patterns:
        row = [item]
        for type in datasets:
            dataset_path = os.path.join(input_path, type)
            for category in categories:
                category_path = os.path.join(dataset_path, category)
                category_summary_file = os.path.join(
                    category_path, category + "_summary.json")
                total_files = count_num_result_files(category_path)
                # print("-" * 10 + category + "-" * 10)
                # print("total count = " + str(total_files))
                try:
                    with open(category_summary_file, "r") as f:
                        data = json.loads(f.read())
                    if item not in data:
                        row.append("0")
                    else:
                        row.append(str(round((data[item] * 1.0 / total_files * 100),
                            2)) + "\%")
                        # print('idioms = %s, num = %d, percentage = %.2f' % (item,
                            data[item], (data[item] * 1.0 / total_files * 100)))
                except Exception as err:
                        print(err)
        print(" & ".join(row))

input_path = "path_to_your_dataset"
```

```
# datasets = ["output_dataset_studio", "output_dataset"]
datasets = ["output_dataset"]

empty_hold_path = ""path_to_your_dataset""

# remove_empty_projects(input_path, empty_hold_path)
# run_summary(input_path)\
# print(count_num_result_files(input_path))
# calculate_percentage_in_category(input_path, datasets)
# calculate_percentage_all(input_path)
```