MASTER'S THESIS

An Exploratory Study of the Learning Progression of Scratch Users

Zeevaarders, A (Ad)

Award date:
2020

Link to publication

An Exploratory Study of the Learning Progression of Scratch Users

Ad Zeevaarders

Student:
Date: July 2020

Open Universiteit
www.ou.nl

An Exploratory Study of the Learning Progression of Scratch Users

by

Ad Zeevaarders

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University, faculty of Management, Science and Technology
Master Software Engineering
to be defended publicly on Day Month DD, YYYY at HH:00 PM.

Student number:	852047990	
Course code:	IMA0002	
Thesis committee:	dr. Bastiaan Heeren (chairman),	Open University
	dr. ir. Fenia Aivaloglou (supervisor),	Open University

Acknowledgements

I wish to acknowledge the love and undying support given to me by my family, especially by my mother, my sister and my brother throughout my studies and this graduation thesis. They were and still remain a constant source of motivation. Without their support, I would never have been where I am today.

I also wish to express my sincere appreciation for the coaching, guidance and help I received from my supervisor, Professor Fenia Aivaloglou. Her motivational capabilities were outstanding and her expertise allowed me to really grow as an academic individual. I have learned so much from her, and I am truly grateful for that opportunity.

Contents

Summary

Block-based programming languages are taking the central stage in early programming education. Scratch is one of the most popular block-based languages, taught at schools and practiced independently. However, do Scratch users get better at programming over time? What areas do they progress in? What do they achieve before moving on? This study aims at uncovering the learning progression of Scratch users on the concepts of loops, conditional expressions, procedures and variables. We constructed a large dataset of over 112 thousand authors and their 1 million projects in order to analyze the development of programming and computational thinking concepts during their time spent on the platform. To collect the dataset, we built a webscraper and parser called Zemi, which we used for downloading and subsequently parsing author and project data into a relational database. Projects are parsed by decomposing the scripts they contain. A script is a sequence of command blocks. The Scratch blocks used in each script are recorded along with possible parameters as part of the block command. We then used the modification dates of projects to chronologically order the projects within each user's repository to allow for progression analysis. In our analysis, we first investigate the development of programming concepts by looking at block usage statistics. Second, we score and analyze the dataset using a computational thinking rubric. Last, we investigate the learning goals achieved by dropped-out authors.

Our results show that, while the users progress in Scratch, there is a positive trend in the use of all concepts that were examined, including the use of *variables* and *conditions*. Within the least utilized concepts, even after the 20th project of Scratch users, are *functions*, *conditional loops* and *logic operations*. *Forever loops* were grasped best by users, with over half of authors using them at their first project. Cloning functionality is almost twice as prevalent among users than defining functions though they share similar mechanics. Examining the users who have left the Scratch platform after creating at least the mean amount of nine projects, we measured utilization rates of over 85% percent across all concepts except functions and conditionals; here half had left without ever utilizing *functions* and a third left without ever utilizing *conditional loops*.

We recommend future research to investigate learning *trajectories* to see in which order concepts are attained. Another recommendation is investigating learning progression through variables not included in this study, such as those pertaining to code quality and code smells. Last, we recommend investigating the cause of the low use of logic operations, conditional loops and functions to see whether they stem from their inherent learning difficulty or if it is the result of how the Scratch platform was designed.

1 Introduction

Computational thinking (CT) is a problem-solving process which involves breaking down a problem, finding and generalizing patterns within it and designing a fitting algorithm to solve it. It has been established that the CT concepts and skills involved in program development are applicable to any domain where problem-solving is required and this makes it an active topic in literature and education. With computer science and computational thinking (CT) becoming increasingly mandatory in primary and secondary school curricula, there is a need to improve and seek new ways of teaching this subject. One way in which this has been done is by the adoption of *introductory programming languages* (IPLs), which are programming environments that provide a low entry barrier for potential learners. IPLs often serve the role of stepping stones towards more advanced programming languages, by supporting users in learning the basic concepts and constructs of computer programming. A subset of these IPLs are block-based. A block-based language uses visual blocks as its syntax to compose program functionality. Examples of block-based languages are Blockly, Alice and Scratch.

Quantitative and qualitative studies on the use of IPLs have been carried out to find out how kids learn (Funke et al. [2017]; Hermans and Aivaloglou [2017]), how they code (Aivaloglou and Hermans [2016]) what the quality of their development artifacts is (Boe et al. [2013]; Moreno-León et al. [2015]; Robles et al. [2017]), and what they struggle with (Grover and Basu [2017]; Mladenovic et al. [2018]; Swidan et al. [2018]). While there has been extensive work performed in the area of measuring programming comprehension and its implications to pedagogy (Boe et al. [2013]; Khairuddin and Hashim [2008]; Moreno-León et al. [2015]; Sorva [2012]), there are not as many studies aimed at analyzing the actual learning progression of young individuals participating in block-based programming environments. Studies that do study learning progressions in block-based environments often explain progression by comparing performance differences in age groups or school grades (Hermans and Aivaloglou [2017]; Seiter and Foreman [2013]; Šerbec et al. [2018]). In other cases, experimental courses or questionnaires are used to track short-term learning (de Souza et al. [2019]; Funke et al. [2017]; Mladenovic et al. [2018]; Troiano et al. [2019]) or to measure existing knowledge in small groups of learners (Grover and Basu [2017]; Swidan et al. [2018]). Abstract measures of proficiency, such as vocabulary breadth and depth, which are not directly related to specific CT concepts, have also been used to infer learning progression (Matias et al. [2016]; Scaffidi and Chambers [2012]; Yang et al. [2015]) and have even sparked debate over the measured progression of Scratch users and the necessity of a large sample size (Matias et al. [2016]; Scaffidi and Chambers [2012]). However, we are aware of no study so far that has quantitatively analyzed the actual learning progression of the individuals participating in the informal programming community at large, exploring it from the start of their learning trajectory to investigate which specific CT concepts and skills are improved upon through the experience they gain on the platform.

The goal of this thesis is to quantitatively explore the learning progress of young individuals programming with Scratch, with the purpose of finding out how and in which areas they progress as they participate in this informal programming community. Because Scratch was specifically designed for children between the ages of 8 and 16 as a first introduction to computer programming, we are also interested in exploring which aspects they have shown signs of learning before leaving the Scratch platform to move to more advanced programming environments. We are interested in exploring users' learning progression in specific programming concepts that have been found to be hard for young learners, like variables, expressions, loops and functions (Aivaloglou and Hermans [2016]; de Souza et al. [2019]; Grover and Basu [2017]; Hermans and Aivaloglou [2017]; Mladenovic et al. [2018]; Seiter and Foreman [2013]; Swidan et al. [2018]), as well as in quantifying the related demonstrated CT skills like abstraction & problem decomposition, flow control, logical thinking and data representation. We aim to apply an automated approach for analyzing a large body of scraped project portfolios in order to answer the following research questions:

RQ1 How do Scratch users progress in the use of elementary programming concepts such as variables, procedures, conditional expressions and loops?

RQ2 What is the learning progression of CT concepts in Scratch users, such as abstraction, data representation, flow control and logical thinking?

RQ3 Which CT concepts were practiced by users that have left the Scratch platform?

To answer our research questions, we scraped and analyzed the public Scratch project repositories belonging to 112 thousand authors and statically analyzed the 1 million projects authored by them for the use of different programming and computational thinking concepts. We then analyzed the resulting learning progress of the users who created the projects and visualized it. The contributions of this thesis are the following:

1. An open-sourced set of software tools for scraping the Scratch website for authors and their project repositories, including parsing logic for the two latest major Scratch versions,

2. A public dataset of 1 million projects created by 112 thousand authors, parsed and labelled with the results of the automated analysis[1],

3. An analysis of the repositories in the dataset in terms of learning progression in programming and computational thinking concepts.

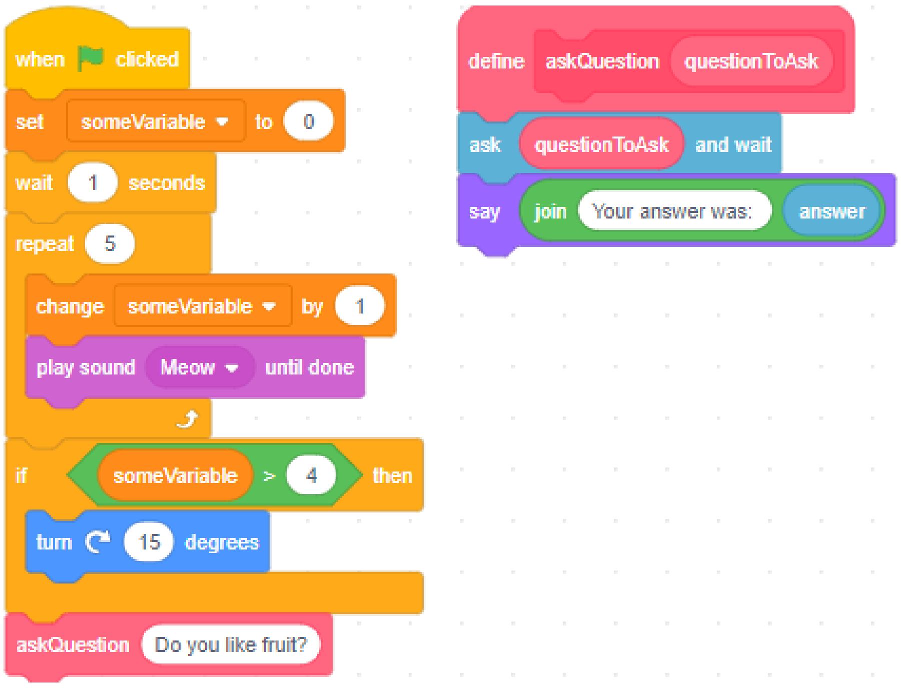
The rest of this thesis is structured as follows: In section 2, we discuss the background and related work pertaining to our study. Next, in section 3, we present the methods we used to answer our research questions. In section 4 we present the results of our progression analysis. In section 5, we discuss our findings, compare them to prior works and identify the limitations and threats to validity of our conducted study. We also offer some perspectives on possible future work. Finally, section 6 presents our most important conclusions.

[1]The dataset and scraping software can be found here: https://bit.ly/34p9dJ9

2
Background and related work

2.1. Relevant Scratch Concepts

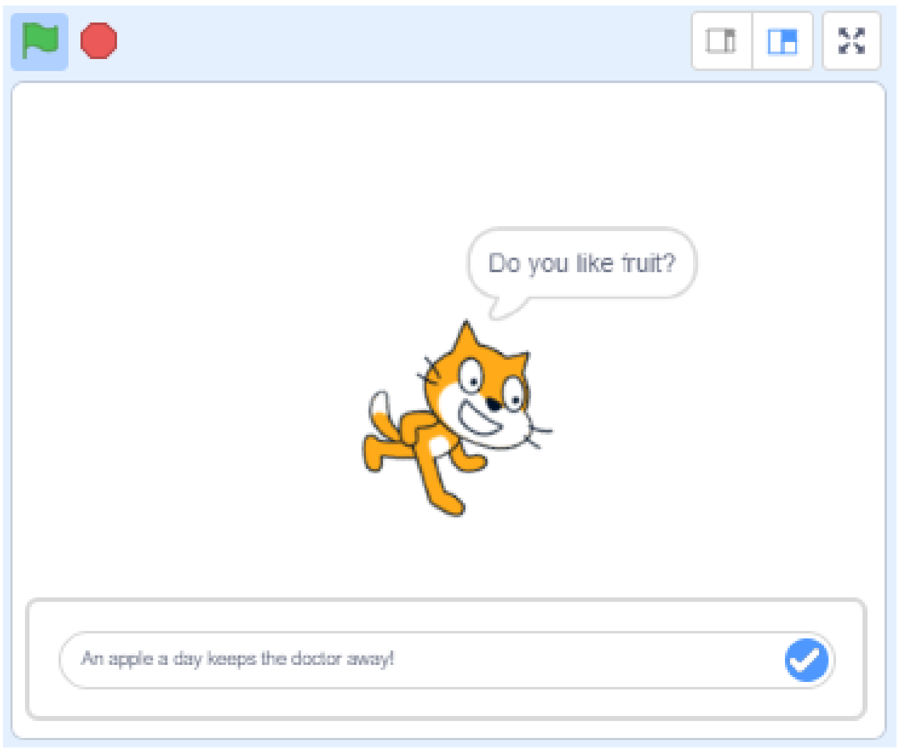
Scratch is a visual programming environment aimed at providing an easily approachable introductory programming experience to users. Scratch can be used as a stand-alone desktop client or as a browser client through the Scratch website. In Scratch, a project is called a *sketch* and programming syntax is represented by visual *blocks*. An example of a sketch is shown in Figure 2.1. Running the sketch by clicking the green flag starts execution (the output is shown in Figure 2.2) because of the top-most *event block* in Figure 2.1; this block is a *hat* block, which serve as event hooks. Blocks come in a variety of *shapes* and each block belongs to a *category*. Block categories group blocks that are similar in functionality, such as *Looks*, *Motion* and *Operators*. Block connectors serve as visual guides showing which blocks can be composed in sequence. Some blocks can be embedded within other blocks. The *if/then* block in Figure 2.1 has an *operator* block embedded into it as the condition to evaluate. This operator block itself has a variable embedded within it as part of the expression. A connected sequence of blocks, initiated by a hat block, forms a *script*. In Figure 2.1, two scripts can be seen, one of which is a user-defined procedure called *askQuestion*. It spawns an input field and captures the entered string in a variable. Scratch uses *sprites* and *stages* as visual components. Sprites are akin to actors and have many visual properties, which can be manipulated to animate it, change its position or make it change size. Stages form the visual backdrop of the program. Scripts are defined within sprites and stages and run in that scope, but can communicate by broadcasting and receiving. All these features combined allow for complex behavior such as parallel execution of scripts or visual effects, showing that Scratch supports advanced programming constructs as well. The projects that users create are stored as packages either locally on disk or in the Scratch cloud. Project code and metadata is converted to JSON structures, while media, like sprites, accompanying the project are saved in separate files. The file format differs between versions of Scratch, with the most recent format being .sb3, coinciding with the release of Scratch 3.0, the current major Scratch version.



```
when [flag] clicked
set someVariable to 0
wait 1 seconds
repeat 5
    change someVariable by 1
    play sound Meow until done
if someVariable > 4 then
    turn ⟳ 15 degrees
askQuestion Do you like fruit?

define askQuestion questionToAsk
ask questionToAsk and wait
say join Your answer was: answer
```

Figure 2.1: Example Scratch program highlighting several different syntax elements, including custom procedures.



```
Do you like fruit?

An apple a day keeps the doctor away!
```

Figure 2.2: Output of Scratch program composed in Figure 2.1.

2.2. Computational Thinking

Computational thinking is a broad term used in education to describe the cognitive skills and processes associated with problem solving involving areas such as problem decomposition, pattern recognition, abstraction and algorithms. It is an essential set of competencies for developing computer applications, but its strength lies in its applicability to any discipline that involves solving complex problems. This latter point is what has warranted its place in early education today. The actual skills, abilities and concepts involved in computational thinking are a subject of debate, as there exist many definitions. Google for Education [1] lists abstraction, algorithm design, automation, data analysis, data collection, data representation, decomposition, parallelization, pattern generalization, pattern recognition and simulation as the concepts belonging to CT. Brennan and Resnick [2012] define the CT concepts as sequences, loops, events, parallelism, conditionals, operators and data. The goal of computational thinking is that through the development of these concepts, a student attains a better set of tools to tackle open-ended and ill-structured problems in a variety of domains. CT has a presence in many national curricula, for example, in K-12 Computer Science in the US and in KS1-3 in the UK. Its classroom adoption furthered the need for CT assessment tools and though these have been receiving attention in literature, there is no clear consensus on CT assessment methodologies (Alves et al. [2019]).

2.3. Related Work

A number of studies have been carried out on the acquisition of computational thinking skills and the understanding of programming concepts by novice programmers in block-based environments. Specific programming concepts have received attention because they have been found to be hard for young learners, whereas inaccurate mental models, unfamiliarity of syntax and misconceptions are within the learning difficulties they commonly experience (Qian and Lehman [2017]).

2.4. Measuring comprehension

Grover and Basu [2017] investigated misconceptions regarding *loops*, *variables* and *boolean logic* by creating a set of assessment items (questions in a questionnaire) that can tell whether an individual holds a misconception or not. First, they investigated which learning goals related to variables, expressions and loops were used for middle school computer science classes, specifically grades 6 to 8. An example of a learning goal is the statement "Students will learn how variables change within loops". Then, a set of focal knowledges, skills and abilities (**FKSA**) were created that are aligned to these learning goals. An example of an FKSA is "The ability to describe what a given loop is doing". Next, the authors construct a 'conceptual assessment framework', which is a questionnaire with each question targeting some FKSA specificied in the previous step. The assessment items included questions aimed at interpreting Scratch programs, where Scratch code blocks are shown and students are asked to interpret it, similar to Swidan et al. [2018], but also broader questions aimed at measuring general algorithmic thinking and problem solving (Grover and Basu [2017]), such as a question showing logical expressions where possible answers are the results of evaluating that expression. The conceptual assessment framework was then instantiated

[1]https://edu.google.com/resources/programs/exploring-computational-thinking/#!ct-overview

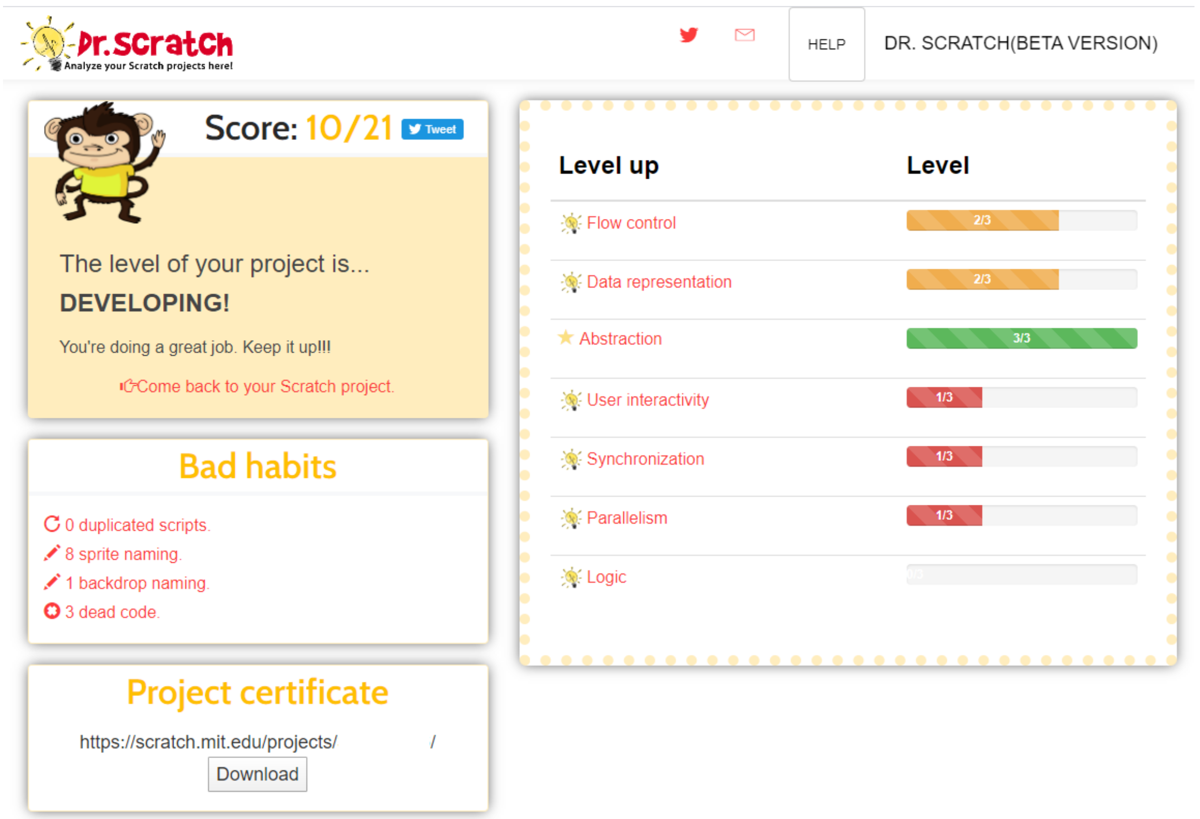
as a pencil-paper test in middle schools among 100 6th, 7th and 8th grade students. Not only were these students asked to fill in the questionnaires, but teachers were also asked how well they thought their students would do on each question and how well the learning goals pertaining to each question were covered in the classroom. The responses were coded according to a rubric. The results show that students harbor a set of misconceptions related to loops, variables and boolean logic. Students thought that actions within a loop are each repeated separately, instead of once for each iteration of the loop. Regarding variables, students thought variables were letters that symbolized some unknown number, akin to what is taught in introductory mathematics. Next to this, students took issue with the length of variable names and how a variable changed when placed in a loop. Loops were also misinterpreted as producing the exact same output every iteration. Regarding operators, students often misinterpreted the OR operator as an XOR operator, due to its use in natural language: "Red or blue" implies one color, but not both.

Swidan et al. [2018] also investigated the holding of misconceptions among younger children by conducting a multiple-choice questionnaire. Each question in this questionnaire tested the understanding of some programming concept in order to elicit holding of misconceptions. The programming concepts investigated in this study are those that have been proven to be difficult for learners: *variables*, *loops* and *conditional* statements. Examples of misconceptions regarding these concepts are "A variable can hold multiple values at a time" and "Loops terminate as soon as the condition changes to false". Questions are instantiated as images of a Scratch program and the subject is asked what happens when it is executed. Answers were designed to indicate whether a subject holds the correct understanding of a concept, holds a misconception, or fails to correctly interpret the question in the first place. The results show that the most common misconceptions among respondents are the sequentiality of code execution, variables being able to hold multiple values at a time and the effect input calls have on execution. One other notable misconception chilren had was that adjacent code (next to a loop, not in it) is executed while the loop is executing. The least common misconceptions among respondents were related to loops and conditions. Here, the low misconception rate was not caused by a profound understanding of loops and conditions, but due to respondents selecting answers that were simply wrong, not related to the holding of a misconception. Further qualitative analysis of misconceptions yielded interesting results: Given a sequential program summing two variables, children often focused on the mathematical operations involved instead of on the sequence of operations. When analyzing the effect age has on the holding of misconceptions, the authors conclude that older children answer more questions correctly. This does not mean younger children hold more misconceptions; it means they gave wrong answers that were not indicative of having a misconception. Participants were also asked if they had any prior programming experience and the results show that knowing Scratch and other languages increases the tendency to understand the concept correctly, while users knowing other languages (and not Scratch) were found to have a bigger tendency in holding a misconception. Subjects knowing only Scratch correlate with giving wrong answers that do not directly exhibit the holding of a misconception. Swidan et al. [2018] further note that holding a misconception is not binary, as some participants struggled with contradicting thoughts when filling in the rationale for their answer.

In a study by Fields et al. [2014], the use of programming concepts was examined in relation to the level of participation, the gender, and the account age of 5,000 Scratch programmers. Aivaloglou and Hermans [2016] transform a collection of over 250 thousand Scratch projects into a statistical dataset by importing them into a relational database and querying the records. This enabled the authors to conduct metrics-based analysis, where metrics like cyclomatic complexity and number of variables were extracted to find out the size and complexity characteristics of Scratch programs. In summary, Aivaloglou & Hermans studied the following: In order to find out the size and complexity characteristics of Scratch programs, the size of projects based on the number of blocks and the cyclomatic complexity of projects are investigated. To find out which coding abstractions or programming concepts are commonly used by Scratch users, usage of *procedures, variables, loops, conditional statements, user interactivity* and *synchronozation* were investigated. To find out how common code smells are in Scratch programs, the authors analyze projects for *dead code, duplicate code, large scripts* and *large sprites*. The authors found that that 78% of scripts within Scratch projects contain no decision points and that most projects are relatively small, with 75% of the projects analyzed containing at most 5 sprites, 12 scripts and 76 blocks. Use of procedures is underrepresented, as only 8% of the scraped projects utilized them. Dead code and code clones both affect a quarter of the scraped projects. Together with the low procedure use, it seems procedures as a form of abstraction are very underrepresented in the general Scratch population. Even though 77% of the projects contained loops, only 14% of those were conditional. Last, the study by Aivaloglou and Hermans [2016] is one of few that utilizes software engineering metrics relating to code quality, namely the McCabe cyclomatic complexity index.

2.5. Static analysis of Scratch projects

Several works have statically analyzed Scratch projects for indications of learning of specific programming concepts. One of the first was that of Maloney et al. [2008], who analyzed 536 Scratch projects for blocks that relate to various programming concepts, and found that within the least utilized ones are boolean operators and variables. Tool support for the static analysis of Scratch programs has also been proposed; The Hairball tool, developed by Boe et al. [2013], is a generic, Python-based automated analysis system for Scratch to improve classroom assessment and grading of Scratch projects, which is often performed manually. It serves a dual role in that students can use it for formative assessment and teachers can use it to support summative assessment. Hairball's architecture is centered on plugins. This way, new types of static analysis can be created as plugins by deriving from the Hairball base class. The authors developed four initial plugins with the goal to discover to what extent a program exhibits competence in an area of programming. These plugins label projects as correct, (semantically) incorrect or incomplete according to their analysis of the programming concept they embody. The first plugin evaluates if the initial state of a Scratch program is correctly set. For example, running the program in Figure 2.1 does not reset the rotation of the sprite when the script starts, causing it to keep rotating 15 degrees. The second plugin evaluates the synchronization between say and sound blocks, which detects if the blocks for playing a sound and having a sprite say something are sequenced correctly. The third plugin can determine if broadcast and receive blocks are matched properly, so that no signal lacks a receiver and vice versa. The last plugin evaluates if complex animation is properly implemented, due to this involving many different concepts such as loops, motion, timing and repetition. The plugins developed for Hairball do not actively score projects based on some computational thinking concept, but


Dr.Scratch
Analyze your Scratch projects here!
HELP
DR. SCRATCH(BETA VERSION)

Score: 10/21 Tweet

The level of your project is...
DEVELOPING!
You're doing a great job. Keep it up!!!
Come back to your Scratch project.

Bad habits
0 duplicated scripts.
8 sprite naming.
1 backdrop naming.
3 dead code.

Project certificate
https://scratch.mit.edu/projects/ /
Download

Level up	Level
Flow control	2/3
Data representation	2/3
Abstraction	3/3
User interactivity	1/3
Synchronization	1/3
Parallelism	1/3
Logic	

Figure 2.3: Dr. Scratch web interface output after uploading a project to analyze

look for correctness in specific patterns relating to programming concepts. Since Hairball can only detect (in)correct instances using a 4-point correctness scale, it cannot judge the quality of an implementation other than reporting it is completely correct. Moreover, Boe et al. [2013] note that, since Scratch projects often require ears and eyes to appreciate fully due to their audiovisual nature, an automated approach can only go so far.

Another static analysis tool is Dr. Scratch, a web application developed in a study by Moreno-León et al. [2015] which can, when given a Scratch project URL, analyse the development of computational thinking concepts and detect bad programming structures within the project source code. The output panel after having uploaded a Scratch project is shown in Figure 2.3. The authors see it as an improvement upon Hairball in that it does not require teachers, who wish to use it to analyze student projects in the classroom, to write the Python scripts necessary to create an assessment in Hairball. After analyzing a program, Dr. Scratch awards scores on different computational thinking concepts. The computational thinking concepts explored by Dr. Scratch are *abstraction*, *logical thinking*, *synchronization*, *parallelism*, *flow control*, *user interactivity* and *data representation*. For each of these concepts, the authors define three levels of proficiency: *basic*, *developing* and *master* that are awarded based on the presence of specific programming blocks. The scores for the individual computational thinking concepts are aggregated into a final grade expressed as either *basic*, *experienced* and *master*. Next to this, Dr. Scratch also presents tips on how to maximize a project's score (which might improve an author's computational thinking skill) for some particular concept like *parallelism*. Much like a modern compiler, it can also show details of the uploaded project that can help a user eliminate mistakes or bad habits, such as *dead* or *duplicated code*.

2.6. LEARNING PROGRESSIONS

More related to our work on the learning progressions of Scratch users is the work by Scaffidi and Chambers [2012], who explored the effectiveness of Scratch in teaching elementary programming skills. For this, the authors used several models that each attempt to model some aspect of (computational) skill and adapted them not only to fit Scratch as a context, but to arrive at quantitative measures that facilitate the use of statistical tests. To collect data, the authors use a custom scraper that randomly scraped a set of 250 users. For each of these users, the scraper collected the first project and a random set of projects after the first one. In total, 1791 projects were collected. The projects are analyzed in terms of breadth (number of different block categories used), depth (number of blocks from a specific category used) and finesse, three measures of technical programming skill originally derived by Huff et al. [1992]. It is found that the average depth and breadth decreases over time. In other words, Scratch users use less different blocks from less categories over time in their projects. A replication study by Matias et al. [2016], using a full dataset of Scratch projects until 2012, challenged the results by Scaffidi & Chambers, showing that the average breadth and depth increased over time and attributing the results by Scaffidi et al. to poor sample size and data collection methods. Scaffidi et al. also inspect the social and engagement aspects of programming with Scratch with what they call the Onion Model, a taxonomy of software developers that assigns a level of engagement and social skill in software development, originally developed to analyze contributors in open-source systems (Ye and Kishida [2003]). The taxonomy contains 8 levels of participation, ranging from passive users and readers, to active developers and core members. To adapt it to a Scratch-specific context, classification rules were created that can assign a level to a Scratch user based on their project metadata. An example of the rules within this classifier is the following: "If a user created more than 25 non-empty Scratch galleries, then they are a Project Leader." The results show that a third of subjects were passive users that only created a handful of animations. About half of the subjects were peripheral users that had created more projects than passive users, but did not participate in galleries. A fifth of the subjects were active users that had contributed to different galleries. Lastly, only three 'Project Leaders' were identified, which was in line with previous results. The authors also conclude that remixing is relatively rare in the Scratch community. Scaffidi and Chambers [2012] also attempt to measure if Scratch users become more efficient at writing code. More specifically, they look for increases in programming speed as a programmer's experience increases. The authors analyze the project save histories, which are included with Scratch project files. Each save records the date and time of the save and the time between saves is summed to arrive at a total time spent on a single Scratch animation. This total time spent is divided by the number of Scratch primitives uses in the project and yields an estimated time spent per primitive. The results show that users spent about 21 minutes more every month on their projects, but did not become more efficient in using primitives over time. After manually analyzing the scraped projects, the authors concluded that there was overall less functionality in later projects.

Related to learning progression, Seiter and Foreman [2013] proposed a model for assessing the development of computational thinking in Scratch users. This rubric-based model assesses computational thinking in primary grade students and was tested on a hand-picked sample of 150 Scratch projects. First, a student's work is coded with evidence variables. Evidence variables are categories of Scratch components directly observable in code. An example of this is the *looks* category, which expresses the manipulation of sprites and the things they say. A student utilizing only *say* or *think* blocks will receive a 'basic' score in this category, while manipulating the color or size of a sprite is awarded with a 'proficient' score. The scores for each of the evidence variables of a project are then mapped onto Scratch-specific design patterns, these are: *Animate Looks*, *Animate Motion*, *Conversate*, *Collide*, *User Interaction* and *Maintaining Score*. These design patterns are then related to a set of computational thinking concepts. Utilizing this model, called the PECT model, the authors are able to assess computational thinking knowledge on different levels of granularity. The computational thinking concepts investigated by the model are *Procedures and Algorithms*, *Problem Decomposition*, *Parallelization*, *Abstraction* and *Data Representation*. Seiter et al. note that while their model uses concepts related to computational thinking abstractions, it does not measure *process skills* like debugging and testing, as these cannot be found or measured using only program source code. The computational thinking concepts use the presence of design patterns as evidence of understanding of the concepts. The study then attempted to model learning progressions of students by comparing the results of the applied rubric across grades 1 to 6. This was possible since the data was originally collected through teacher galleries and a conscious effort was made to select projects from different grades. The authors found that the *Conversate*, *Animate Looks* and *Animate Motion* were uniformly used across grades, while *Collision* and *Maintaining Score* were underrepresented until grades 5 and 6. *Maintaining Score* was especially underrepresented and this was attributed to the pattern requiring an understanding of variable creation and assignment, which are related to the Data Representation concept. Around grades 3 and 4, students begin to understand and use the Animation pattern more uniformly and this result is roughly the same for the Animate Looks pattern. In summary, design patterns requiring understanding of parallelization, conditionals and, especially, variables were under-represented by all grades apart from grades 5 and 6.

Yang et al. [2015] modelled learning trajectories as the cumulative vocabulary use of a Scratch user over time, without focusing on the development of specific computational thinking concepts. The authors make the assumption that attaining a wider Scratch vocabulary over time is indicative of learning. After generating vocabulary growth trajectories, the different trajectories are clustered to find similar learning patterns. The authors further state that Scratch blocks should not be treated equally in the sense that rarely used blocks represent more advanced programming knowledge. To this end, weights were assigned to all different Scratch blocks by using the inverse document frequency. The quality of the assumption that infrequently used blocks are a learning indicator can be argued, as the highest weights were given to blocks like *setwhirlto*, which ìs the parameterized version of setting an effect on a sprite. The whirl effect is actually one among many options when choosing to set a visual effect in Scratch. Other visual effects include pixelate and mosaic effects, and we believe the assigned weights would be more representative if measured by the usage of the set effect block itself, instead of its specific parameters like *whirl*. Among the heavier weights is also *jokeoftheday*, an experimental Scratch block, which we believe is not more indicative of Scratch programming comprehension than, for example, the *abs* block, which returns the absolute value of an integer and received a weight 5 times lower than *jokeoftheday*. To generate the actual learning trajectories, the first 50 projects of users were put in sequence and analyzed for cumulative vocabulary use. These trajectories varied widely from one another and so the different trajectories were clustered using K-means++, leading to four clusters: A, B, C and D. Users in cluster D showed the highest initial vocabulary knowledge, broadest vocabulary use and overall the fastest learning trajectory. Users in cluster A had the lowest initial vocabulary (only two different blocks used in first project compared to 17 in cluster D) and progressed the slowest. Clusters B and C were in between A and D. The authors note that they did not include age or gender in the analysis, and a possible interpretation for the difference in clusters could be that they represent age groups. For each user in the clusters, the 100 most frequent words used in project descriptions and comments were extracted. Cluster D had the highest amount of unique words and contained the most words indicative of game design and programming. The word 'poor' was used very frequently in cluster A and the authors note this might indicate users in that cluster are aware of the poor quality of their work. By analyzing the block use of the different clusters, it was identified that users in Cluster D used rarer blocks across the board and that users from cluster A use common blocks more often, even more so than users in cluster D. The authors conclude by noting that the different clusters can be seen as different canonical learning patterns each associated with a certain Scratch subpopulation. Finally, the authors put forth a proposal for a system that recommends programming blocks and entire projects to users in the identified clusters, so that their learning trajectory can be aligned to the trajectory patterns of similar, but more advanced users within their cluster.

de Souza et al. [2019] investigated the evolution of computational thinking in young individuals learning programming using project source code that was generated during several 12-week game building workshops. The workshop activities were centered around game development, targeting a different game with increasing complexity each week. The games targeted the use of specific Scratch concepts, such as animation and collision, defined by a teaching rubric. Project data was imported into a database, including used blocks and their categories, dead code and cyclomatic complexity metrics. By describing the games created in the workshop with self-organizing maps, they can be analyzed for use of different blocks, block categories, dead code and similarity. Notably, the maps show the different ways kids created a solution. For instance, it is found that some subjects used Forever and some used Repeat/Repeat Until blocks to solve the same problem, indicating subtle differences in understanding. The results show that most dead code pertained to the Looks, Control, Sensing and Operator block categories. High cyclomatic complexity was often caused due to subjects copying sprites (and their scripts containing decision points) instead of having the same sprite change costumes. The blocks most used in the workshop were Forever, If & If/Else Repeat/Times/Until, Stop, Wait and Wait Until blocks.

3 METHODS

To answer our research questions, we created a scraper and parser tool called Zemi, which we used to scrape the Scratch website for authors and their complete repository of public projects. We then parsed these into a relational database. Then, using an automated learning comprehension rubric, we assigned comprehension scores to the collected projects based on static analysis of the project source code. We quantitatively analyzed our dataset of repositories for signs of improvement over time on programming and CT concepts. Finally, we analyzed the repositories in our dataset of users that had left Scratch, to examine what they learned during their stay. The process is described in more detail in the following paragraphs.

3.1. DATASET

We obtained data on the authors and their project repositories by scraping the Scratch platform. In order to conduct our progression analysis, we collected a large dataset of authors and their project repositories.[1] We obtained this data scraping the Scratch platform by querying its public API[2] for authors and projects, which we then parsed into a relational database for further analysis. The scraping and parsing is done by Zemi[3], which is able to scrape author & project metadata and project source code and parse the results into an attached MySQL database.

3.1.1. SCRAPING

To collect complete project repositories of users, random scraping of projects is infeasible. Therefore, Zemi starts by scraping an initial set of random front-page authors from the Scratch API. Then, those authors' friends and followers are recursively scraped. Each author's data is then parsed into a relational database to form the set of authors whose repositories will be scraped. Next, Zemi scrapes project metadata for each author's project. Project metadata includes the project name, view count, remix details and creation and modification dates. This metadata is subsequently used to download the actual source code through the API, which is then stored as files on disk. To collect our dataset, Zemi

[1]https://drive.google.com/open?id=1UCPQQkXTmn7ADaJtOuTFQ0CNuwcqNOcE
[2]https://github.com/LLK/scratch-rest-api/wiki
[3]https://github.com/ospani/zemitoolkit

started scraping on the 1st of September 2019, until the 27th of October, 2019, and scraped 195,767 authors and 7,109,821 projects.

3.1.2. Parsing and Filtering

Zemi parses projects by decomposing its JSON representation. Projects are first split into scripts, which symbolize any connected sequence of blocks. Next, the scripts are split into blocks. Each block's command and order within the script is recorded, along with its parameters. Nested blocks, such as if-blocks or expressions, are unwound and flattened using a nesting depth specifier.

Scratch uses three different formats (.sb1, .sb2 and .sb3) to save projects. These formats correspond to Scratch's major software versions. The sb2 and sb3 formats are both plain but different JSON structures, and constitute the majority of the projects on Scratch. To handle these different formats, separate parsing logic was written. The sb1 format is binary, and could therefore not be parsed. Since the project metadata does not specify the project format, it can only be identified at parsing time by reading the project source code file. Each project format uses its own set of block opcodes. For example, the opcode for appending an item to a list is '*append:toList:*' in sb2 projects, but is called '*data_addtolist*' in sb3 projects. To ensure uniform analysis of both sb2 and sb3 projects, Zemi uses a block mapping that specifies for each sb2 opcode its equivalent sb3 opcode.

The filters that were subsequently applied to the dataset were both on the scraped authors and on their projects:

1. Authors that were found to have sb1 projects or projects that we had failed to parse, which amounted to 582,143 projects in total, were excluded from the dataset and from further analysis. This filter was applied because the order in which projects were developed by the authors is important for our learning progression analysis, and we therefore need complete author repositories, for which we can analyze all included projects.

2. For the remaining authors, the filter that was subsequently applied to their projects was that of empty projects and remixed projects, which were excluded from further analysis. This filter was applied because it is not possible to determine what an remixer's own contribution is relative to the original project, especially since original projects can evolve after they have been remixed, and version information is not provided by the Scratch platform.

3. After excluding remixes, we further filtered out authors whose repositories consisted solely of remixes.

The filtering process resulted in the dataset that was used for the analysis, consisting of the repositories of 112,208 authors, containing a total of 1,019,310 self-created, non-empty projects. The source code of these projects was then parsed, resulting in approximately 172 million blocks divided over 21 million scripts, which were stored in a relational database.

3.2. Concepts Evaluation

To find out which programming concepts were utilized and possibly improved upon by authors, we investigated the usage of procedure definitions and calls, If & If/Else blocks, Repeat Until & Repeat Times blocks, Forever blocks and variables.

Targeted concept	Dr. Scratch concept	Basic (Level 1)	Developing (Level 2)	Proficiency (Level 3)
Loops	Flow Control	Sequence of blocks	Repeat, forever	Repeat until
Expressions	Logical Thinking	If	If-else	Logic operations
Variables	Data Representation	Modifiers of sprite properties	Operations on variables	Operations on lists
Functions	Abstraction & Problem decomposition	More than one script and more than one sprite	Definition of blocks	Use of clones

Table 3.1: Targeted concepts projected onto Dr. Scratch concepts, including its scoring rubric

To arrive at quantitative measures of the use of computational thinking concepts for our collected dataset, we required an automated comprehension scoring model. There has been little consensus on how to best assess computational thinking (Grover and Pea [2013]) because of the lack of consistent scoring criteria and the multitude of CT definitions available (Alves et al. [2019]). Instead of synthesizing yet another comprehension model or assessment framework, we opted to review those already existing and used successfully in literature. To guide this review, we specified the set of criteria a model should meet to be used in our study's context:

1. Assesses selected concepts
2. Is automated or able to be automated
3. Assigns quantitative comprehension measures
4. Is compatible with Scratch blocks

We reviewed relevant models (Boe et al. [2013]; Funke et al. [2017]; Moreno-León et al. [2015]; Seiter and Foreman [2013]; Von Wangenheim et al. [2018]) from the mapping study on CT comprehension models by Alves et al. [2019] and chose Dr. Scratch's rubric as the comprehension model, as it directly satisfied all our criteria. Dr. Scratch (Moreno-León et al. [2015]) is a web application where learners can upload a Scratch project and have it evaluated for different CT concepts and software metrics. Figure 2.3 shows a part of the Dr. Scratch web interface after having uploaded a project to score. The tool uses a rubric to assign one of four levels (None, Basic, Developing, Proficiency) to each CT concept based on static analysis of the source code. We used the Dr. Scratch rubric to score our collected projects on *Abstraction and Problem Decomposition*, *Parallelism*, *Logical thinking* and *Data Representation* by translating its conditions for each of the proficiency levels to SQL queries and running them against our dataset of projects. For example, for a project to receive a 'Basic' score in Logical Thinking, it has to contain an if-block. The corresponding SQL query checks if any blocks with the if-block opcode are in the project's collection of blocks. The mapping of our concepts to those used by Dr. Scratch, and the relevant conditions for each proficiency level, are shown in Table 3.1. We scored the projects for each concept by evaluating the conditions for the highest level first. If those conditions are satisfied, then the score is returned. If not, then the requirements for the next highest level are evaluated.

3.3. Progression Analysis

To track an individual's programming progression, a model that is able to analyze learning progressions is preferable. However, from our related work survey, we did not find any

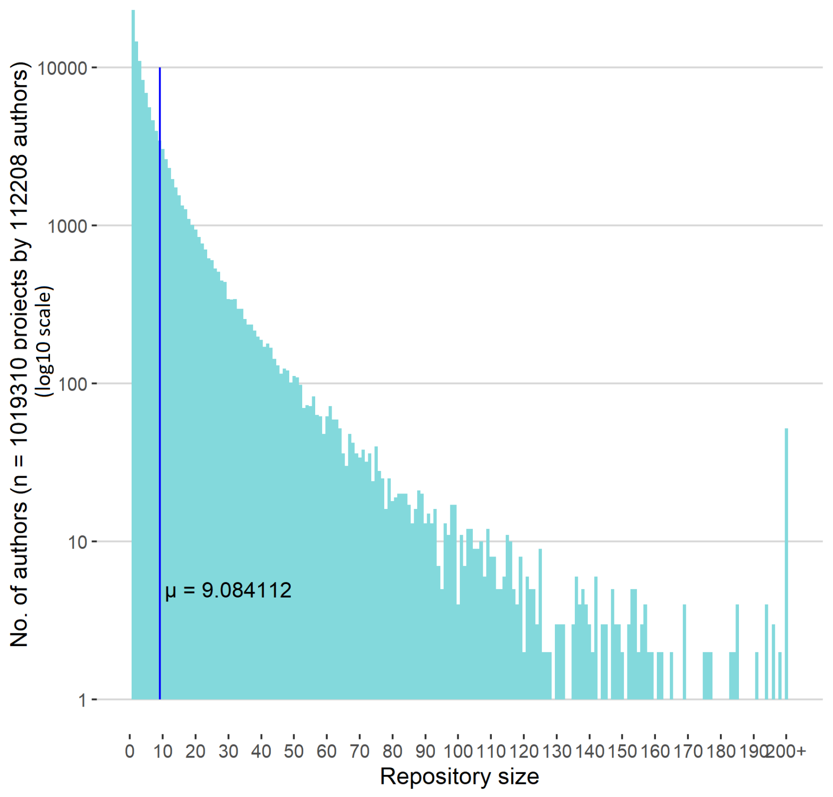


Figure 3.1: Repository sizes in number of projects of analyzed repositories

methods or frameworks that describe how to generate such trajectories in our study's context. Studies that do analyze the progression of different concepts do this by differentiating between grades or age groups (Hermans and Aivaloglou [2017]; Seiter and Foreman [2013]; Šerbec et al. [2018]). These latter two variables are data we cannot obtain by scraping the Scratch website. Therefore, to explore for signs of improvement over time, we chose to order the projects within each author's repository by their Modified date. This makes it possible to follow progression from the first to last project edited. We did not use the Created date to sort projects due to the possibility of authors coming back to older projects and editing them. We then performed a statistical analysis, using a combination of SQL queries and R scripts to generate the information and visualizations necessary to answer our research questions. We further compared the first 3 with the last 3 projects of the authors who had at least 6 projects in their portfolios to gain further insight in their learning progression.

3.4. Identifying Dropped Out Users

We defined a dropped out user by calculating the difference in days between subsequent project modification dates for all the projects in all repositories we scraped. At the 95th percentile the difference in days is 41. We then took the latest project modification date in our dataset (27th of October, 2019) and subtracted 41 days from it. This date (16th of September, 2019) was used as the cutoff date for inactivity. An author that has no project with a modified date after that cutoff date is considered a dropout. In total, we analyzed

87,461 dropped-out author repositories containing 864,287 projects. For those authors, we performed two types of analyses:

1. Progression analysis, following the process of Section 2.5;
2. Analysis of the concepts they never used during their time on Scratch. To do that, we analyzed all the projects of each dropout, including dropouts that had less projects. We separately examined users who left with more than 9 projects, where 9 is the mean repository size in our analyzed sample of authors, as shown in Figure 3.1.

4 Results

4.1. RQ1: Progression of programming concept usage

In the analysis below, we visualized information for the first ten projects of each author, including authors with less than 10 projects. The mean repository size in our sample was 9.08, as shown in Figure 3.1 and we chose 10 projects as any more would inhibit visualization. Furthermore, after 10 projects the number of projects started decreasing to negligible levels. The graphs up until the 20th project are shown in Appendix A. For each of our targeted concepts, we defined the set of Scratch blocks that corresponded to it and plotted the proportions for the first 10 projects in each author's repository. For our analysis, repositories containing less than 10 projects were included as well. In total, we analyzed 112,208 unique author repositories containing 1,019,310 projects. The number of projects in each repository up until the 10th project is 589,273. The resulting visualizations are discussed in the following paragraphs.

Figure 4.1 shows the distribution of *conditional expression* usage in the form of *If* and *If/Else* blocks. These blocks were referenced in 42.6% of authors' first projects, and show a slight upward trend towards 49.7% at the 10th project. This proportion remains stagnant even at the 20th project, where the proportion is 49.9% out of all 12876 20th projects.

For *functions*, shown in Figure 4.2,we searched for projects that contained scripts that were function definitions. Of the first projects, only 7.58% contained procedures, though this increased to 12.5% at the 10th project, towards 14.5% at the 20th project.

Regarding *loops*, we analyzed the Forever (Figure 4.3), Repeat Times (Figure 4.4) and Repeat Until (Figure 4.5) usage.

Forever loops were encountered in 64.3% of authors' first projects, showing a slight upward trend towards 72% at the 10th project, increasing to 74.2% at the 20th project.

Repeat Times loops were encountered in 35.6% of the first projects, also showing a slight upward trend towards 42.4% at the 10th project. The progression slope is continued, towards 45.6% at the 20 th project.

Repeat Until loops, which require a conditional expression as a parameter, were encountered in just 14.5% of authors' first projects, again showing a slight upward trend to 20% at the 10th project. After the 10th project, the progression slope becomes flatter with the proportion increasing to 22.6% at the 20th project.

Figure 4.6 shows the distribution of *variable* usage. User-created variables were used in 34.2% of authors' first projects and showed a slight upward trend from there, remaining

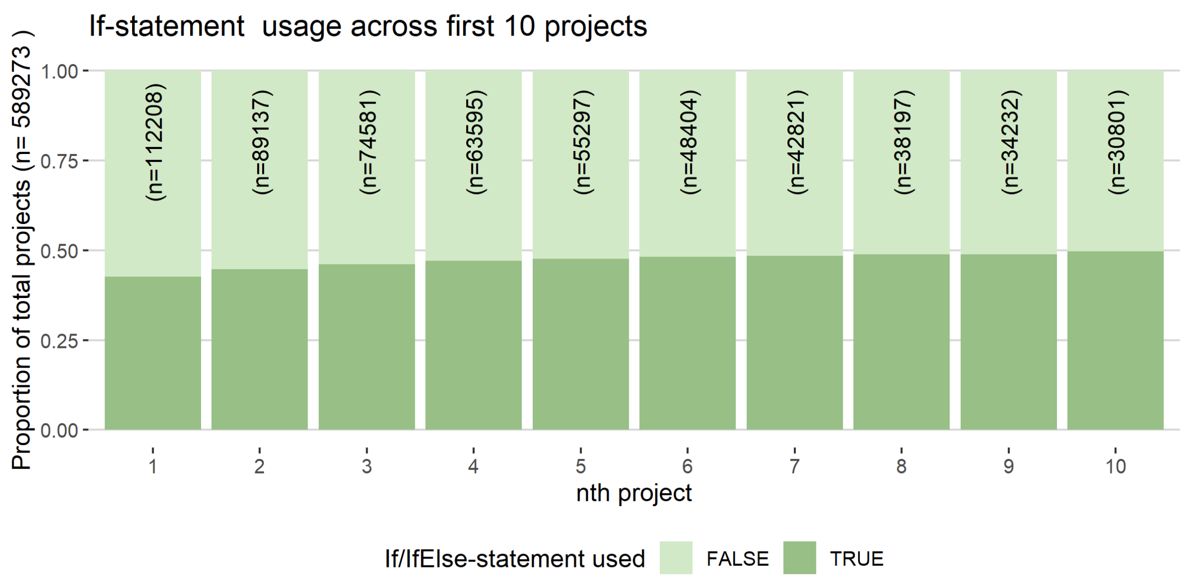


Figure 4.1: Distribution of If/If-Else usage

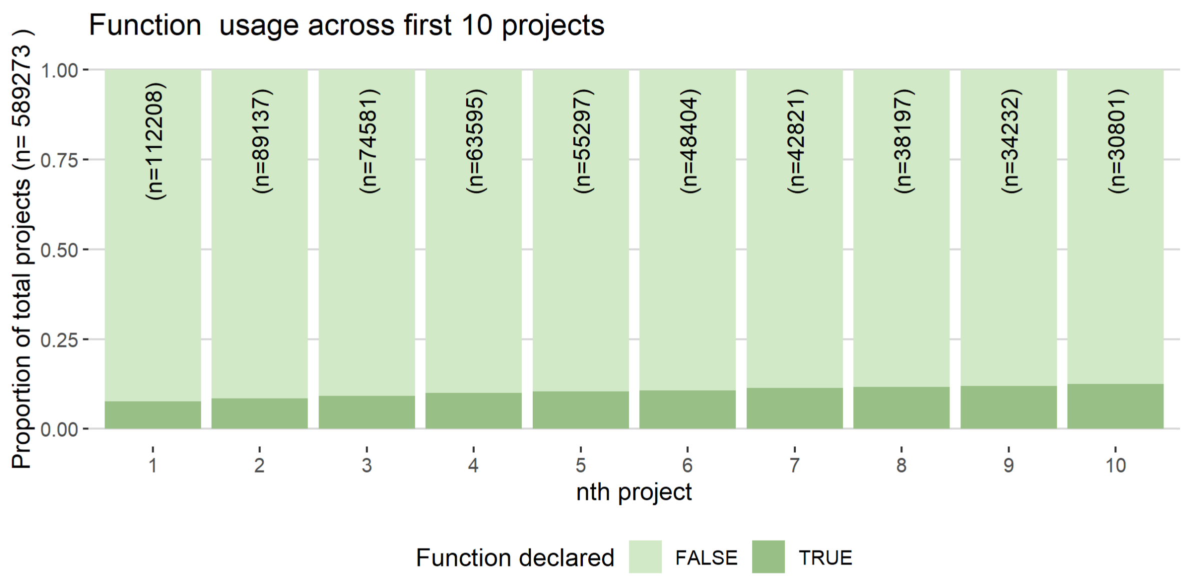


Figure 4.2: Distribution of function declarations

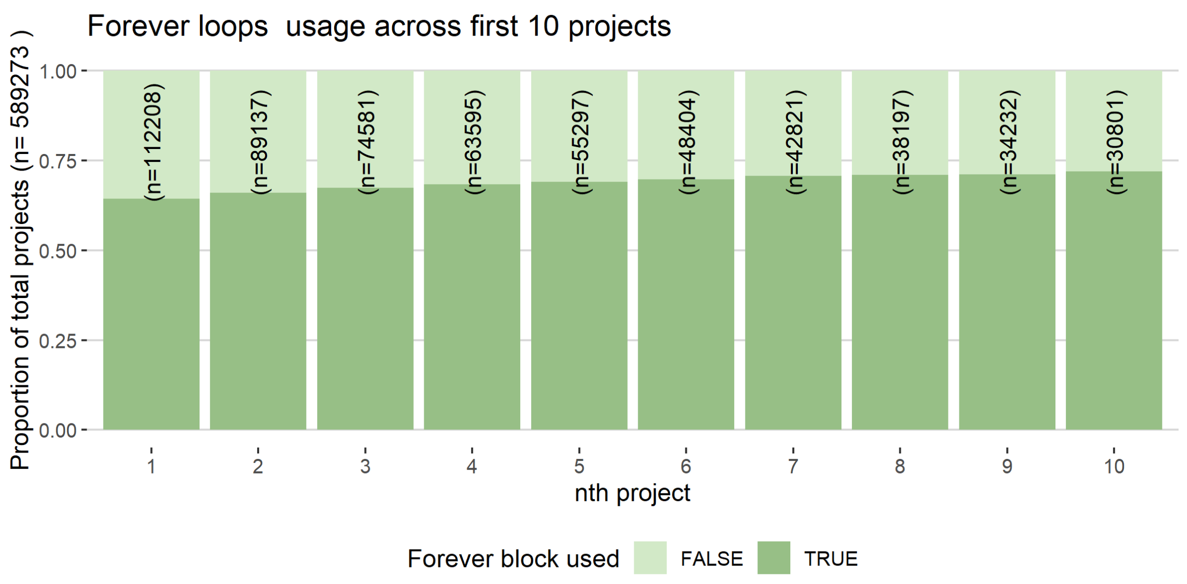


Figure 4.3: Distribution of Forever usage

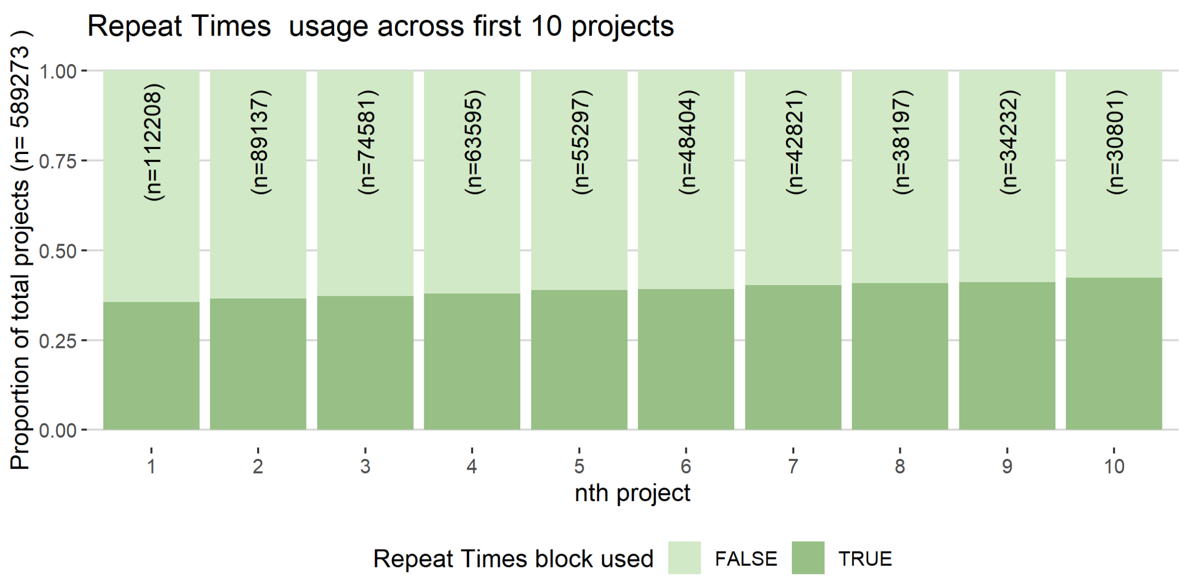


Figure 4.4: Distribution of Repeat Times usage

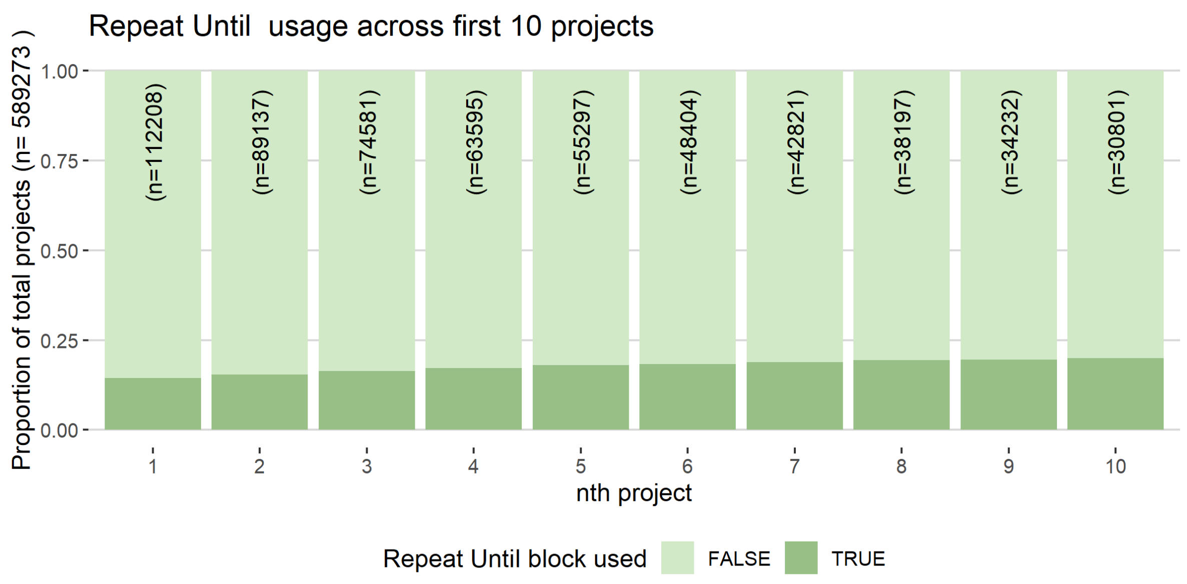


Figure 4.5: Distribution of Repeat Until usage

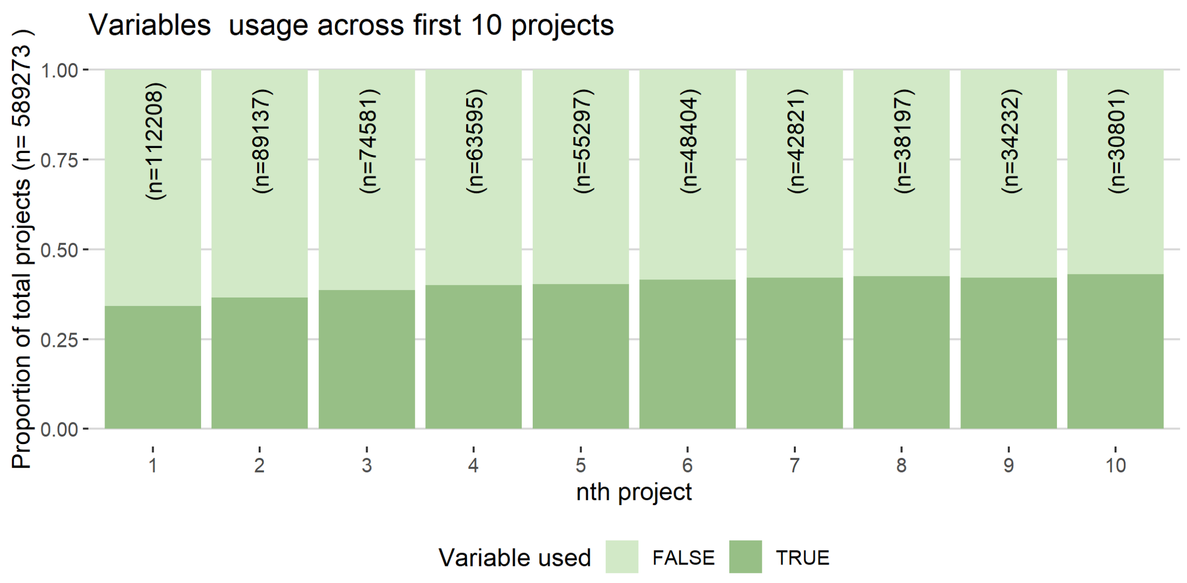


Figure 4.6: Distribution of variable usage

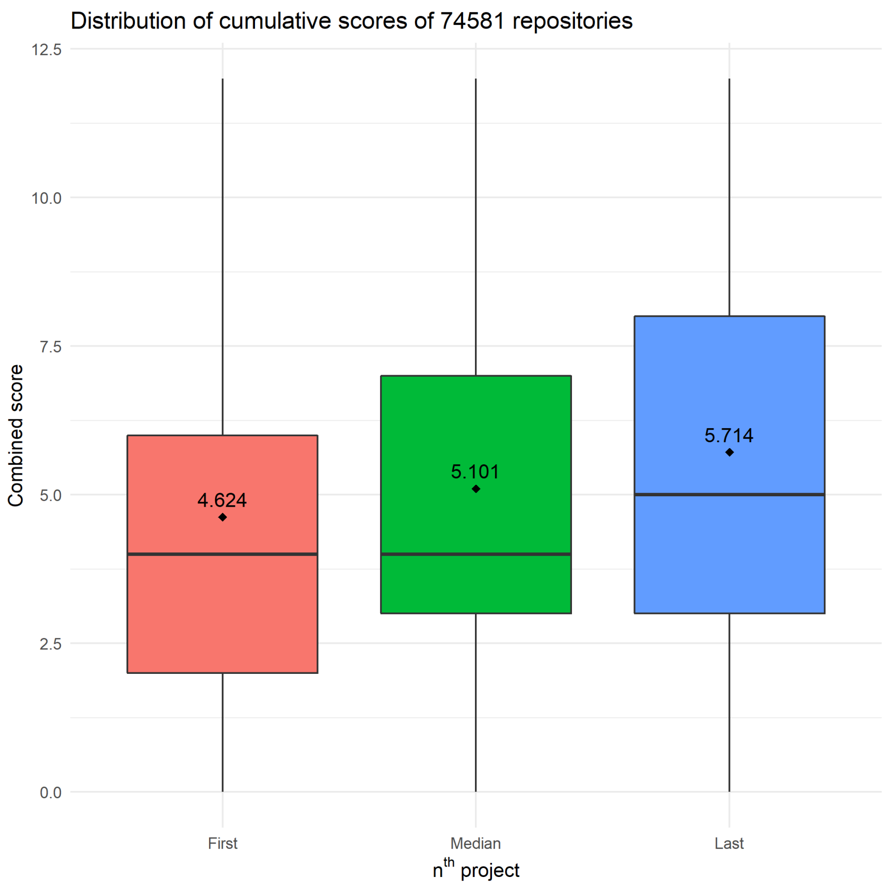


Figure 4.7: Combined CT concept scores across first, median and last projects in each repository of at least size 3. The diamonds show the average.

stagnant at 43% around the 10th project and afterwards, being used in 44.4% of the 12876 20th projects.

4.2. RQ2: Computational Thinking rubric evaluations

For each of the targeted computational thinking concepts, we visualized the distribution of scores assigned by the Dr. Scratch rubric of Table 3.1. Moreover, we compared the scores of the first and last two, three and five projects of each author with four, six or ten or more projects respectively and visualized them in Figures 4.8 to 4.10. We added multiple figures because analyzing higher numbers of first and last projects results in overall less repositories analyzed, since less repositories will have the required size. For example, there are 63595 repositories with a size of at least 2, but only 48404 repositories with a size of 4. The results are discussed in the following paragraphs. Additionally, Figure 4.7 shows the distribution of the combined scores for the four concepts at the first, median and last projects of authors with at least three projects in their repository. Here, we observe increasing scores as users create more projects. Notably, the distribution of the second quarter nearly doubles in size between the median and last project.

The concept of *flow control* (Figure 4.11) was utilized the most. Here, 62.1% of users achieved a level of at least 2 in their first project, meaning they utilized *repeat times* and *repeat forever* blocks. A small upward trend is visible for level 3, which is the utilization

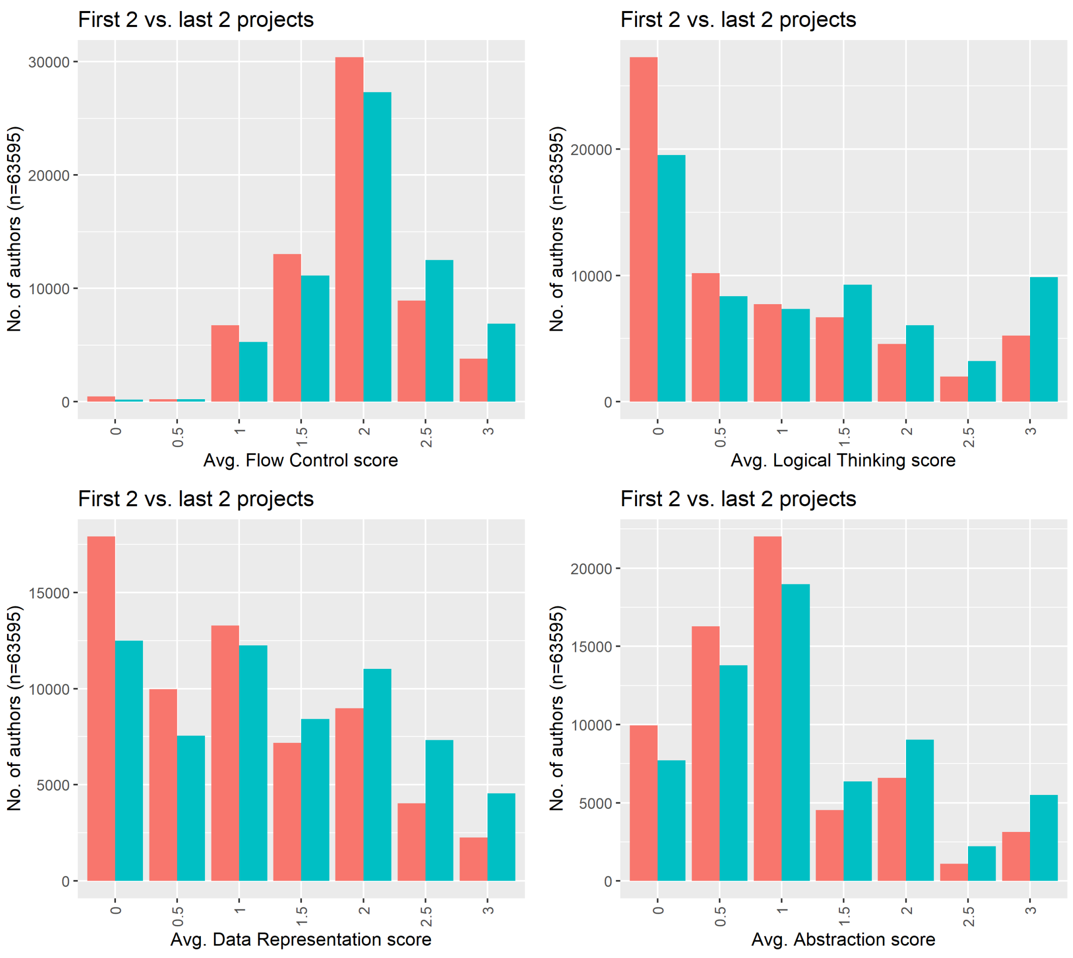


Figure 4.8: Average CT concept scores of first 2 (left/red bars) and last 2 projects (right/blue bars) in author repositories

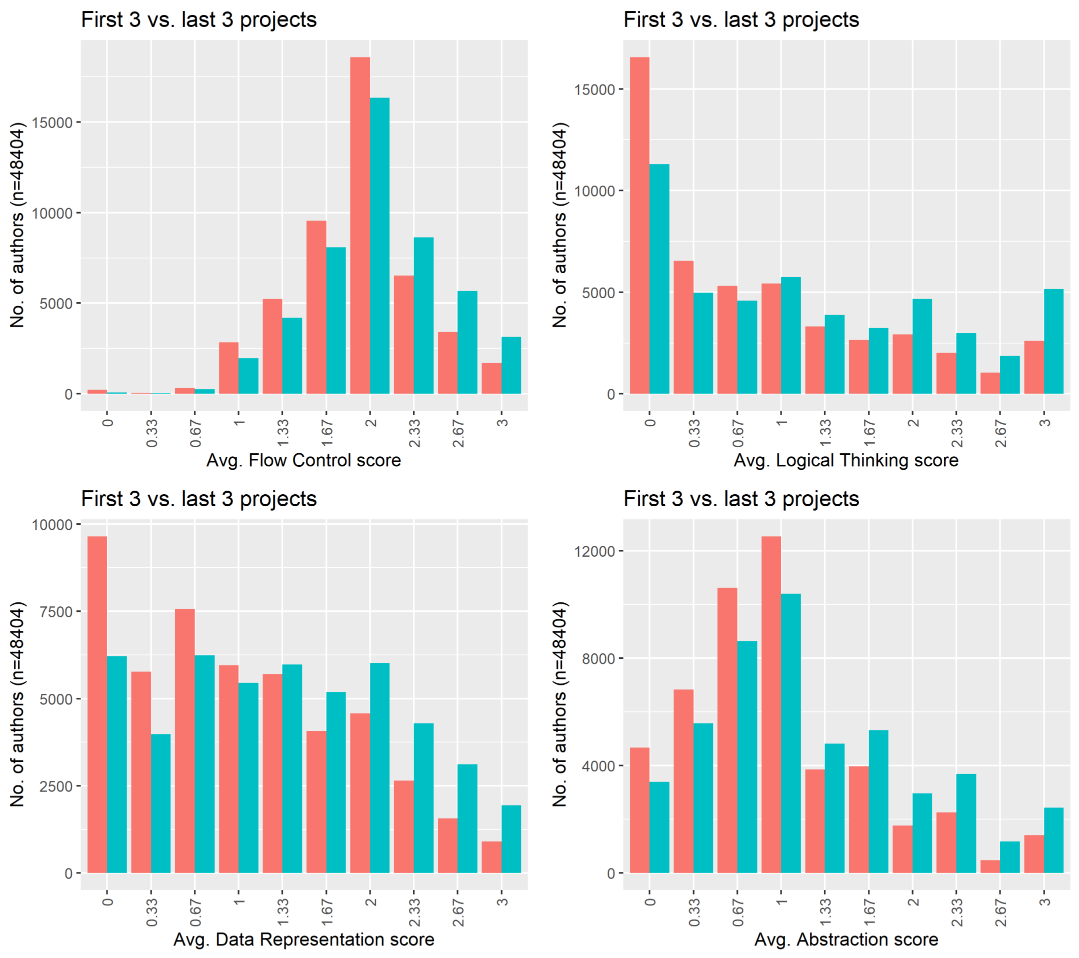


Figure 4.9: Average CT concept scores of first 3 (left/red bars) and last 3 projects (right/blue bars) in author repositories

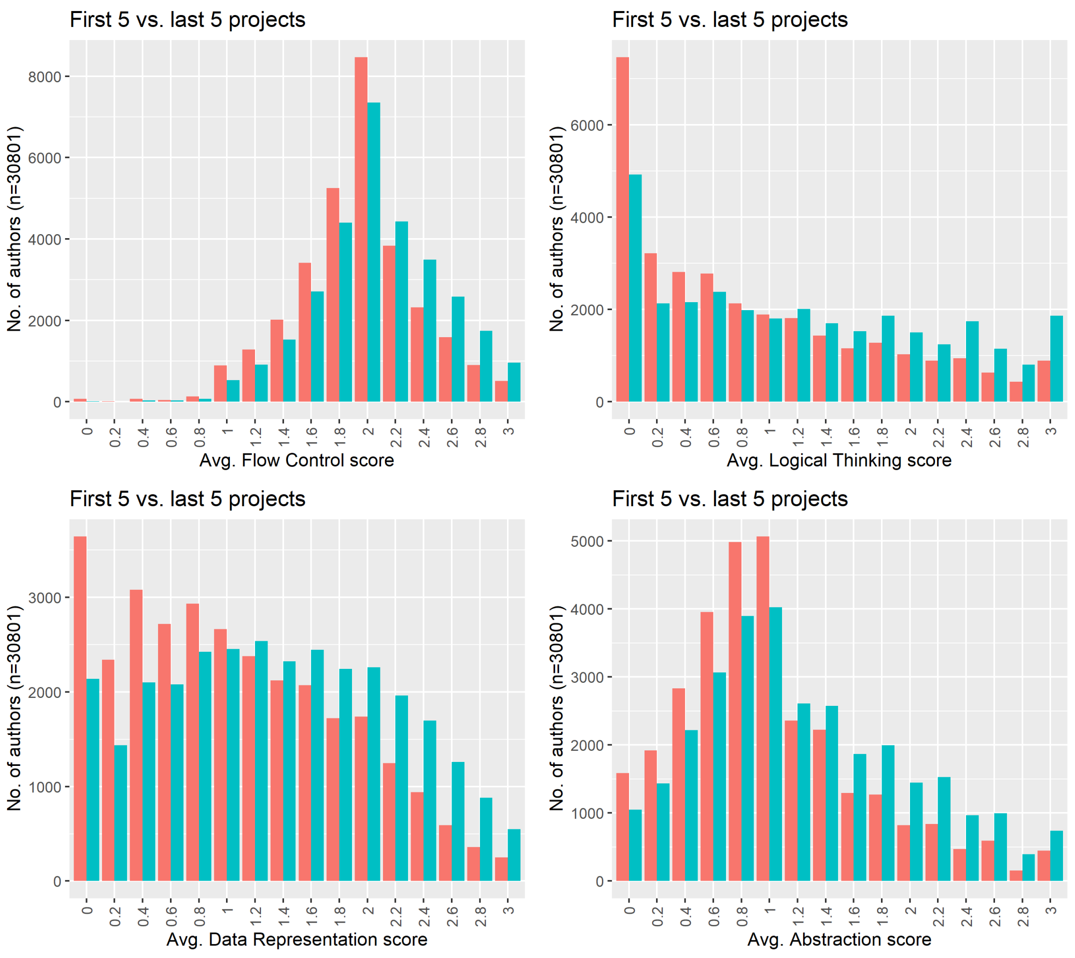


Figure 4.10: Average CT concept scores of first 5 (left/red bars) and last 5 projects (right/blue bars) in author repositories

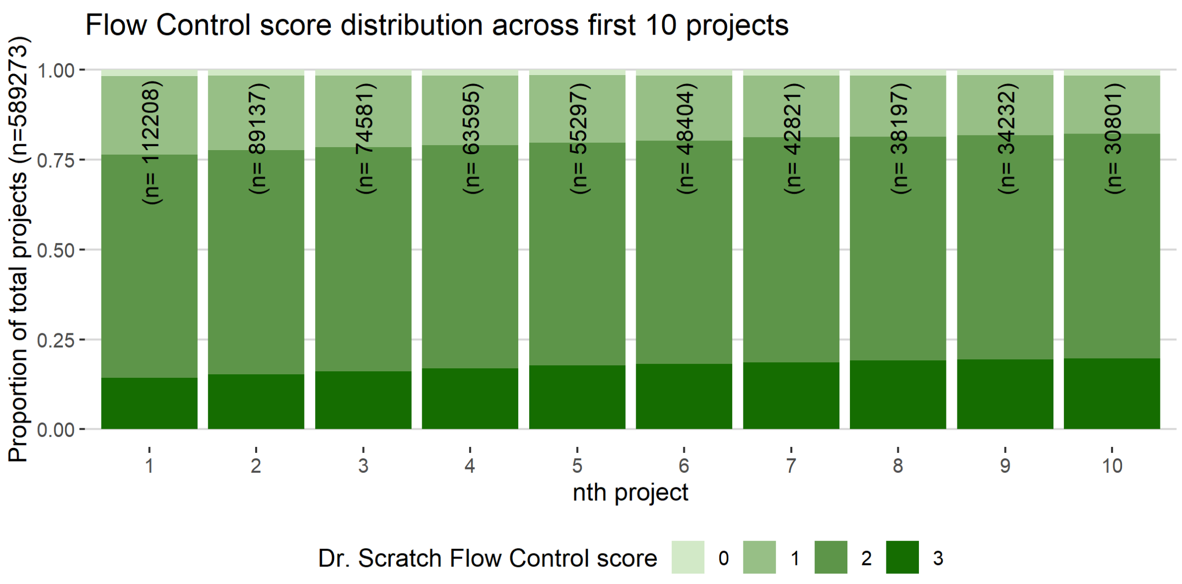


Figure 4.11: Distribution of flow control scores

of *Repeat Until* blocks. The usage of *If/If-Else* blocks is also presented in Figure 4.1. Level 1, which is the easiest score to attain as it requires at least one sequence of blocks, shows a downward trend. In Figures 4.8 to 4.10, the distribution of average flow control scores attains a more negative skew, showing a shift towards higher scores.

For *logical thinking* (Figure 4.12), an upward trend is visible for level 3, which is defined as use of logic operations. Here use of AND, OR and NOT blocks is seen to increase from 16.1% to 24.1%. For level 2, which is the use of If/Else blocks, the proportions increase slightly, by 1.31% over 10 projects. Level 1, which is the use for If blocks, sees a slight decrease of 2.1%. The amount of projects that did not utilize any logical thinking concepts also decreased slightly. The average logical thinking scores of the first and last projects of each repository are shown in Figures 4.8 to 4.10. Here, a fairly large drop in projects with level 0 is observed and the distribution of last projects shifts towards higher scores.

For *data representation* (Figure 4.13), an upward trend for level 3, which is the of lists, is observed. Level 2, the use of variables, shows a slight upward trend as well. Level 1, the modification of sprite properties, decreases slightly, starting at 21.4% at the first project, towards 20.8% at the 10thproject. The projects that did not utilize data representation concepts decreased by 8.4% over 10 projects. The average Data Representation scores of the first and last projects in Figures 4.8 to 4.10 show a general increase in scores. Here, the drop in projects that received a score of 0 could imply gradual adoption of variables and list usage.

Last, the *abstraction* concept (Figure 4.14) shows an upward trend for level 3, which is the use of clones. For level 2, the definition of procedures, a slight upward trend is observed as well. Figure 4.2 shows the individual scores for level 2. Level 1, which requires a project to have more than one script and more than one sprite, shows a slight downward trend due to adoption of higher levels, as the proportion of users who did not utilize any abstraction concepts remained stagnant, even up until the 20th project. The average abstraction scores of the first and last projects are shown Figures 4.8 to 4.10. Here, a proportional decrease in average scores up until level 1 is observed.

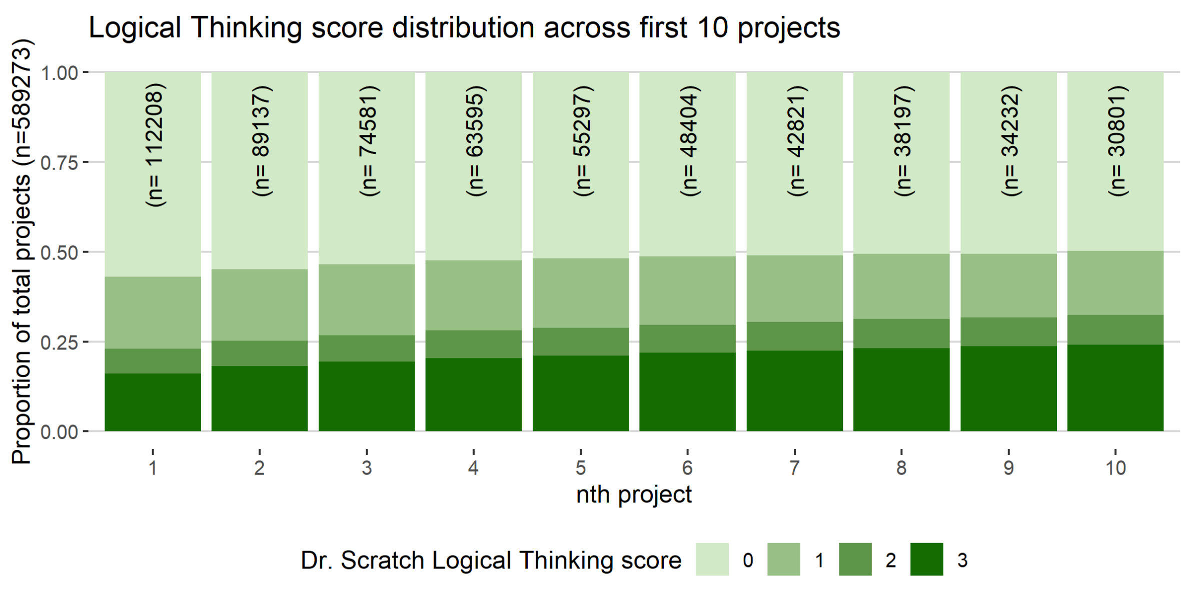


Figure 4.12: Distribution of logical thinking scores

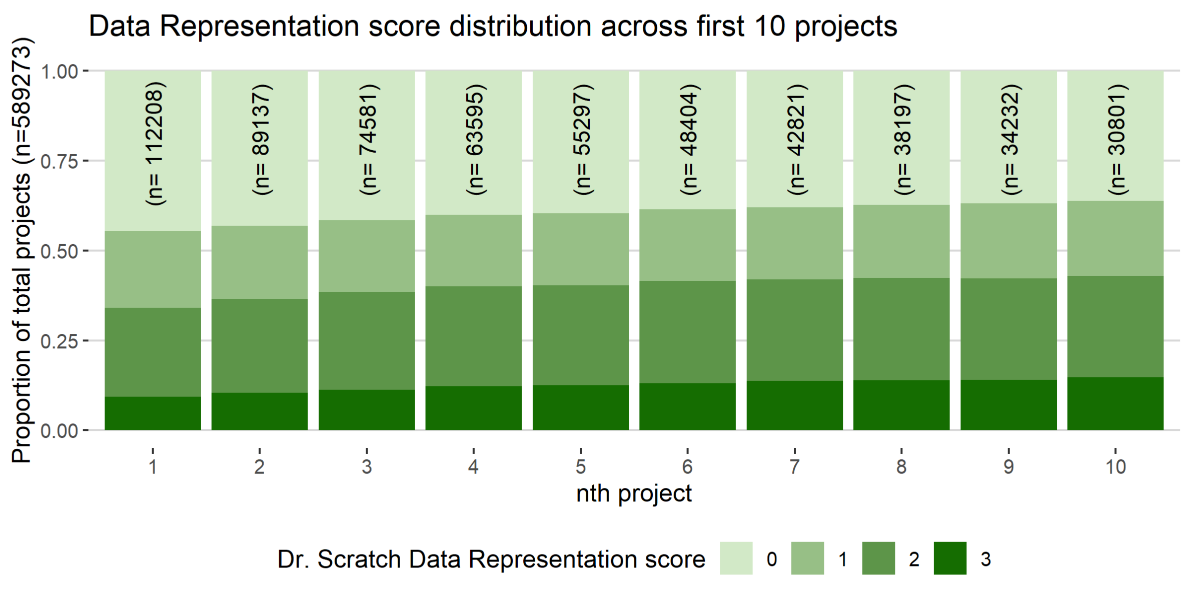


Figure 4.13: Distribution of data representation scores

Table 4.1: Summary of programming concept usage for first 10 projects of dropped-out users

Population	87461	81147	67359	57043	49340	42943	37841	33558	29958	26862
Project	1	2	3	4	5	6	7	8	9	10
If/If-Else	**T:** 41%	44.4%	45.6%	46.5%	46.9%	47.5%	47.9%	48.4%	48.3%	48.8%
	F: 59%	55.6%	54.4%	53.5%	53.1%	52.5%	52.1%	51.6%	51.7%	51.2%
Functions	**T:** 7.26%	8.38%	8.96%	9.65%	10.2%	10.5%	10.9%	11.3%	11.4%	12%
	F: 92.7%	91.6%	91%	90.4%	89.8%	89.5%	89.1%	88.7%	88.6%	88%
Forever	**T:** 64%	65.8%	67.2%	68.1%	68.9%	69.6%	70.4%	70.7%	71%	71.7%
	F: 36%	34.2%	32.8%	31.9%	31.1%	30.4%	29.6%	29.3%	29%	28.3%
Repeat Times	**T:** 35.2%	36.2%	36.9%	37.4%	38.5%	38.7%	39.7%	40.1%	40.3%	41.5%
	F: 64.8%	63.8%	63.1%	62.6%	61.5%	61.3%	60.3%	59.9%	59.7%	58.5%
Repeat Until	**T:** 13.6%	15.2%	16.1%	16.7%	17.6%	17.9%	18.3%	18.9%	18.9%	19.3%
	F: 86.4%	84.8%	83.9%	83.3%	82.4%	82.1%	81.7%	81.1%	81.1%	80.7%
Variables	**T:** 32.7%	36.2%	38%	39.2%	39.5%	40.6%	41.3%	41.7%	41.7%	41.9%
	F: 67.3%	63.8%	62%	60.8%	60.5%	59.4%	58.7%	58.3%	58.8%	58.1%

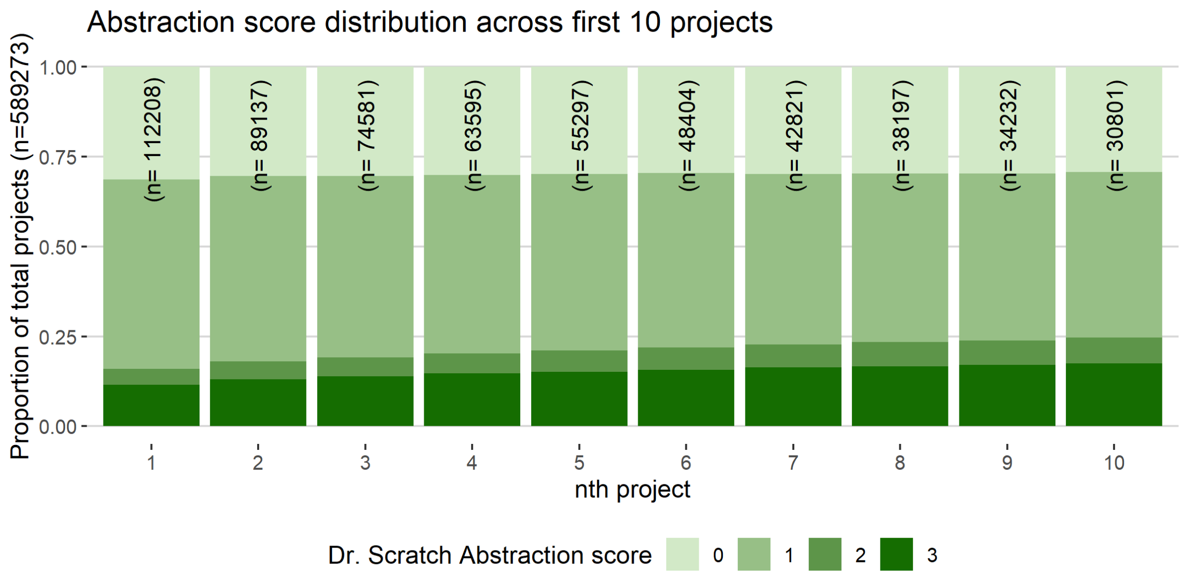


Figure 4.14: Distribution of abstraction & problem decomposition scores

Table 4.2: Summary of CT concept usage for first 10 projects of dropped-out users

Population	87461	81147	67359	57043	49340	42943	37841	33558	29958	26862
Abstraction	1	2	3	4	5	6	7	8	9	10
0	32.6%	31%	30.8%	30.5%	30.4%	29.9%	30.2%	30.2%	30.3%	29.9%
1	52%	51.4%	50.5%	49.8%	49.1%	48.8%	47.8%	47%	46.8%	46.6%
2	4.33%	4.83%	5.16%	5.48%	5.82%	6.06%	6.2%	6.58%	6.42%	6.87%
3	11%	12.8%	13.6%	14.2%	14.8%	15.2%	15.8%	16.2%	16.4%	16.8%
Flow Control	1	2	3	4	5	6	7	8	9	10
0	1.88%	1.69%	1.71%	1.69%	1.63%	1.7%	1.71%	1.73%	1.58%	1.71%
1	22.2%	21%	20.1%	19.6%	18.9%	18.4%	17.6%	17.5%	17.1%	16.6%
2	62.5%	62.3%	62.4%	62.2%	62.1%	62.3%	62.6%	62.2%	62.7%	62.7%
3	13.4%	15%	15.8%	16.5%	17.4%	17.6%	18%	18.6%	18.6%	19%
Data Representation	1	2	3	4	5	6	7	8	9	10
0	45.8%	43.5%	42.2%	40.9%	40.3%	39.2%	38.7%	38%	37.8%	37.1%
1	21.7%	20.4%	20.1%	20%	20.2%	20.2%	20.2%	20.5%	21.1%	21.1%
2	23.8%	25.8%	26.7%	27.2%	27.2%	27.8%	27.8%	28.1%	27.6%	27.6%
3	8.72%	10.3%	11.1%	11.9%	12.2%	12.7%	13.3%	13.5%	13.5%	14.2%
Logical Thinking	1	2	3	4	5	6	7	8	9	10
0	58.6%	55.2%	53.9%	52.9%	52.4%	51.9%	51.6%	51.1%	51.2%	50.6%
1	19.6%	19.9%	19.7%	19.4%	19.3%	18.9%	18.6%	18.4%	17.9%	17.9%
2	6.65%	7.01%	7.38%	7.71%	7.6%	7.79%	7.88%	8.07%	7.93%	8.22%
3	15.2%	17.9%	19%	19.9%	20.7%	21.3%	21.9%	22.5%	23.0%	23.3%

4.3. RQ3: Concepts practiced by dropped-out users

The results of the programming concepts and CT concepts analysis for dropped-out users are presented in Tables 4.1 and 4.2 respectively. As with the full repositories, we analyzed the first ten projects of each dropout, including dropouts that had less projects, totalling 513,512 out of 864,287 projects by 87,461 dropouts. Next to this, we analyzed the proportion of dropped-out users with a repository size of 9, equal to or above our sample mean (See Figure 3.1), which is 29,958 authors with 647,249 projects. We call these complete dropouts in our analysis below. A summary of the statistics presented below is shown in Table 4.3.

Regarding *if/if-else* statements, there is a small progression in usage in the first 10 projects. The amount of dropouts that never utilized them is 20,780, or 23.76% of the total dropped-out authors. Notably, the number of complete dropouts that never utilized these is 2,530, or only 8.44% of complete dropouts, signifying that 91.56% of Scratch users who created at least 9 projects had used an if/if-else statement at least once.

Regarding *function* definitions, the usage proportion of 7.26% nearly doubled over 10 projects, towards 12% at the 10th project. The amount of dropouts that never defined a function is 59,858, or 68.44% of all dropped-out authors. This number is lower for complete dropouts. Here, 13,838 authors, or 46.19% of complete dropouts, never defined functions. This means 53.81% of users that created at least 9 projects defined a function at least once during their stay.

Regarding *forever* loops, the usage increases slightly over the first 10 projects, from 64% to 71.7% at the 10th project.The amount of dropouts that never utilized a Forever loop is 8,392, or 9.59%. The amount of complete dropouts that never used these is 307, or 1.025% of complete dropouts. This means 98.975% of dropouts with a repository containing at least 9 projects used a forever loop at least once.

Regarding *repeat times* loops, usage grows from 35.2% at the first, to 41.5% at the 10th project. The amount of dropouts never using them is 22,524, or 25.75%. This number is lower for complete dropouts, where it is 1,698, or 5.67% of the complete dropouts, meaning 94.33% of dropped-out authors with a repository of at least 9 projects used this kind of loop at least once before leaving.

For *repeat until* loops, an increase of 5.7% over the first 10 projects is seen, towards 19.3% at the 10th project. The amount of dropouts that never used them is 48,263, or 55.18%. The proportion is smaller for complete dropouts, where only 9,982 or 33.32% never used them.

Use of *variables* saw a notable increase of 3.5% between the first and second project, then increasing slightly towards 41.9% near the 10th project. The total amount of users never utilizing them is 26,538, or 30.34% of the dropped-out authors. Notably, only 3,555 or 11.87% of complete dropouts never used variables.

Table 4.3: Summary of programming concept usage results for dropped-out users, and dropped-out users which created at least the mean size of 9 projects (called complete dropouts)

Concept	**Number (percentage) of dropouts not utilizing concept**	**Number (percentage) of complete dropouts not utilizing concept**
If/If-Else	20,780 (23.76%)	2,530 (8.44%)
Functions	59,858 (68.44%)	13,838 (46.19%)
Forever loops	8,392 (8.59%)	307 (1.025%)
Repeat times	22,524 (25.75%)	1,698 (5,67%)
Repeat until	48,263 (55.18%)	9,982 (33.32%)
Variables	26,538 (30.34%)	3,555 (11.87%)

5 Discussion

Overall, our findings indicate that Scratch users progress in the use of several CT concepts like abstraction & problem decomposition, flow control, logical thinking and data representation as they create more projects. Repeat forever blocks were among the most popular in our sample. Their high frequency of use might imply that this block and its use is easily understood, but it could also be attributed to the fact that it is required for the execution of several types of Scratch programs.

Our analysis of the learning progression was made from three viewpoints: that of elementary programming concepts, using a rubric that evaluates computational thinking levels, and through the concepts practiced by dropped-out users. Working with this approach, the benefits that we find are twofold. First, it captures that not all programming concepts should or need to be employed in every project to convey that a user is advanced. Focusing solely on the progression as evaluated by the use of programming concepts or the rubric can be misleading, because it does not give a full image of the learning progression. For example, examining the use of variables throughout the first 20 projects of users we found a positive trend from 34.2% in the first projects to 44.4% of the 20th projects, which is also in line with the findings of Aivaloglou and Hermans [2016], where variable usage was found in 31.51% of Scratch projects. This view alone would be misleading, because it could be interpreted as an under-utilization of variables from users even after creating 20 projects. However, when examined from the last viewpoint, that of the concepts practiced by dropped-out users, we find only 30% having never used them and, more importantly, that the majority (88.13%) of the users who have left Scratch after 'seriously' using it (creating at least the mean amount of nine projects) have used variables at least once, indicating that users do practice the concept of variables while using Scratch.

The second benefit from adopting this approach is that it enabled us to better capture progression and the lack of it. For example, the under-utilization of logic operations was not brought up when examining the elementary concepts, since logic operations can be used in conditional expressions, loops, and other expressions, but as an element in the logical thinking dimension of the computational concepts rubric. It is also observed here that the high If/If-Else block usage contrasts with the low conditional loop usage, since both require a conditional expression. This could be attributed to lack of understanding of how the conditional expression operates with loops.

The under-utilization of functions was evident across all viewpoints, with their use remaining as low as in 14.5% of 20th projects, with almost half of the users who have left the Scratch platform after creating at least nine projects never having created a function. Low function usage was also reported in existing work in the Scratch repository (Aivaloglou and Hermans [2016]), where only 7.7% of 233,491 projects were found to utilize them, as well as by Troiano et al. [2019], who observed little progression in abstraction concepts beyond level 1 in their analysis of the Dr. Scratch scores of 317 projects created in 8th grade, where only 18 projects used functions. Our analysis confirms those findings in a large set of users and their learning progressions. This very essential programming concept is therefore rarely practiced, which can be attributed both to limitations imposed by the Scratch environment, like the local scope of procedures, and to the difficulty for internalizing certain computational thinking concepts before a certain age (Seiter and Foreman [2013]).

5.1. Data and Software

In terms of the produced dataset, we believe that it can be useful beyond the purposes of our study, especially because it contains entire user repositories instead of random projects. To the best of our knowledge, no other study has conducted an integrated scraping and parsing effort on this scale. Next to this, no study so far seems to have acknowledged or addressed the differences in the Scratch project versions. Many measures can be derived from our constructed dataset: cyclomatic complexity, length of scripts, and other metrics related to code quality can be easily calculated from the relational database structure. Vocabulary breadth and depth, as used in Scaffidi and Chambers [2012] can also be directly derived. Many more variables exist within our dataset, which can be used to further analyze Scratch users and the projects they create, such as the use of sprites and stages, procedure arguments, the used parameters for blocks and more. Analysis activities are further supported by the version-agnostic structuring of our dataset, as we used a translation table for all block commands. For each Scratch block, the translation table contains the sb2 and sb3 representation for that command, making it possible to compare projects of different versions, for example. More importantly, it can be extended to possibly fit future Scratch version as well.

To support the replicability of our findings we have open-sourced every program and script we used, including the SQL queries used for scoring the projects, the queries to determine block usage and the filtration queries. Additionally, Zemi itself is completely open-source and available on GitHub. Next to this, we also offer all of our analysis scripts, from exporting the SQL data, to importing it in R, calculating the metrics and creating the visualizations presented in this thesis. We are confident that our study is reproducible to the fullest extent possible.

5.2. Threats to validity

The dataset that was constructed and used for the analysis contains all projects that the scraped users had made public, but not their private projects. There is no way of knowing about the private contents of user's repositories. In our analysis, this might have influenced repository size and dropout calculation, as well as the ordering of the projects, as public projects alone do not capture the full extent of an author's activity. The exclusion of remixes might have influenced the same aspects. However, including remixes would have skewed our analysis results, since remixed projects should not be considered authored projects and it is not possible to distinguish with the available information a remixer's own contribution relative to the original project.

Regarding the ordering of projects, we used the latest modification date of projects to order them chronologically. Another option would be use the project creation dates. However, since users can go back and edit their projects at any time, that might not capture the true chronology of a user's activity. Regarding our sample of authors, we might have captured only a local sample of Scratch authors due to our scraping method: by scraping friends and followers, and their friends and followers and so on, we might collected only authors that have some relationship towards each other.

Another threat pertains to the measured rubric levels. We applied the rubric scoring procedure top-down, meaning that the conditions for obtaining the highest level of proficiency are evaluated first. If the conditions are satisfied, then that proficiency level is assigned. For example, a project using clones, but no custom procedures will receive an Abstraction score of 3, even if the levels below it were never attained.

6 Conclusion

In this thesis we presented a quantitative study on a large body of scraped Scratch repositories. Starting with a webscraper that scrapes authors and projects from the Scratch website, we collected a large dataset of authors and their entire project repositories. Each author's entire repository was scraped to enable chronological analysis of projects within it, where we used the Modified date of projects to see in what order they were worked on. Each project was parsed by recording the blocks used in it in a relational database. Because Scratch projects have different versions, we maintained a translation table to handle all version's blocks uniformly. Authors whose repository contained unparseable or sb1-versioned projects were excluded from the analysis, as well as all empty and remixed projects. This resulted in a set of 112,208 repositories containing 1,019,310 projects. By analyzing the projects within these repositories, using a combination of SQL queries and R scripts, for use of *loops*, *expressions*, *variable* and *function* concepts, both individually and using a CT rubric, we explored their author's progression. Across the board, the results show an increase in concept utilization and CT scores for active and dropped-out users alike. In the following paragraphs, we answer each of our research questions.

RQ1: How do Scratch users progress in the use of elementary programming concepts such as variables, procedures, conditional expressions and loops? We observe a progression towards higher scores in the first ten projects of all the public projects analyzed. Logic operations, conditional loops and functions are used very little. The progression is most profound at the start of a user's project repository. In the first three projects, authors saw the highest growth in variables and if/if-else statement utilization. Notably, even though the use of functions was underrepresented, its utilization still nearly doubled over the first ten public projects.

RQ2: What is the learning progression of CT concepts in Scratch users, such as abstraction,data representation, flow control and logical thinking? Authors slowly progress in the utilization of all CT concepts. Authors attained the highest scores on the flow control concept, due to level 1 requiring a sequence of blocks and level 2 requiring repeat forever and repeat times blocks, which were grasped better by authors in general. Related to the abstraction & problem decomposition concept, we observe that cloning (level 3) is overall more prevalent than use or definition of functions (level 2), though their mechanics in the Scratch editor are similar. Notably, the proportion of users with a score of 0 remains stagnant, showing little progression. The data representation concept showed the most uniform distribution of scores, with variable and list utilization increasing. Notably, virtually no progression was measured for level 1 (modification of sprite variables) even though the proportion of projects with score 0 decreased. This could be the result of a different type of learning trajectory, where modification of sprite properties is not the next logical step in progressing the data representation concept.

RQ3: Which CT concepts were practiced by users that have left the Scratch platform? All concepts were improved upon by dropped-out users and at a faster rate than regular users. Authors that left Scratch with a repository size of at least the mean size of 9 projects (complete dropouts) utilize all concepts the most compared to all dropped-out users. For complete dropouts, utilization rates for If/If-Else, Forever loops, Repeat times loops are all above 85%. Slightly less than half of those authors never used functions, and a third never utilized repeat until loops. Conditionals like repeat until loops are still underutilized, with just 45% of dropped-out users using them, the use of If/If-Else blocks, which also require a conditional, was quite high, with only a quarter of those dropouts having never used them. Though the concept utilization rates by dropouts are lower than that of complete dropouts, they are notably higher than the rates for the complete population, especially in the area of function definitions, repeat times and repeat until statements.

Bibliography

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. ACM, 2016. 1, 2, 8, 31, 32

Nathalia Da Cruz Alves, Christiane Gresse Von Wangenheim, and Jean CR Hauck. Approaches to assess computational thinking competences based on code analysis in k-12 education: A systematic mapping study. *Informatics in Education*, 18(1):17, 2019. 6, 15

Bryce Boe, Charlotte Hill, Michelle Len, Greg Dreschler, Phillip Conrad, and Diana Franklin. Hairball: Lint-inspired static analysis of scratch projects. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 215–220. ACM, 2013. 1, 8, 9, 15

Karen Brennan and Mitchel Resnick. New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 annual meeting of the American educational research association, Vancouver, Canada*, volume 1, page 25, 2012. 6

Alexandra A de Souza, Thiago S Barcelos, Roberto Munoz, Rodolfo Villarroel, and Leandro A Silva. Data mining framework to analyze the evolution of computational thinking skills in game building workshops. *IEEE Access*, 2019. 1, 2, 12

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*, pages 2–11. ACM, 2014. ISBN 978-1-4503-3250-7. 7

Alexandra Funke, Katharina Geldreich, and Peter Hubwieser. Analysis of scratch projects of an introductory programming course for primary school students. In *2017 IEEE global engineering education conference (EDUCON)*, pages 1229–1236. IEEE, 2017. 1, 15

Shuchi Grover and Satabdi Basu. Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic. In *Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education*, pages 267–272. ACM, 2017. 1, 2, 6

Shuchi Grover and Roy Pea. Computational thinking in k–12 a review of the state of the field. *Educational Researcher*, 42:38–43, 02 2013. doi: 10.3102/0013189X12463051. 15

Felienne Hermans and Efthimia Aivaloglou. Teaching software engineering principles to K-12 students: a MOOC on Scratch. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET)*, pages 13–22. IEEE, 2017. 1, 2, 16

Sid L. Huff, Malcolm C. Munro, and Barbara Marcolin. Modelling and measuring end user sophistication. In *Proceedings of the 1992 ACM SIGCPR Conference on Computer Personnel Research*, SIGCPR '92, pages 1–10, New York, NY, USA, 1992. ACM. ISBN 0-89791-500-3. doi: 10.1145/144001.144011. URL `http://doi.acm.org/10.1145/144001.144011`. 10

Nurul Naslia Khairuddin and Khairuddin Hashim. Application of Bloom's taxonomy in software engineering assessments. In *Proceedings of the 8th WSEAS International Conference on Applied Computer Science*, pages 66–69, 2008. 1

John H. Maloney, Kylie Peppler, Yasmin Kafai, Mitchel Resnick, and Natalie Rusk. Programming by choice: Urban youth learning programming with scratch. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, pages 367–371. ACM, 2008. ISBN 978-1-59593-799-5. 8

J Nathan Matias, Sayamindu Dasgupta, and Benjamin Mako Hill. Skill progression in scratch revisited. In *Proceedings of the 2016 CHI conference on human factors in computing systems*, pages 1486–1490. ACM, 2016. 1, 2, 10

Monika Mladenovic, Ivica Boljat, and Žana Žanko. Comparing loops misconceptions in block-based and text-based programming languages at the k-12 level. *Education and Information Technologies*, 23:1483–1500, 07 2018. doi: 10.1007/s10639-017-9673-3. 1, 2

Jesús Moreno-León, Gregorio Robles, et al. Dr. scratch: a web tool to automatically evaluate scratch projects. In *WiPSCE*, pages 132–133, 2015. 1, 9, 15

Yizhou Qian and James Lehman. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Trans. Comput. Educ.*, 18(1), October 2017. doi: 10.1145/3077618. URL `https://doi.org/10.1145/3077618`. 6

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

Christopher Scaffidi and Christopher Chambers. Skill progression demonstrated by users in the scratch animation environment. *International Journal of Human-Computer Interaction*, 28(6):383–398, 2012. 1, 2, 10, 32

Linda Seiter and Brendan Foreman. Modeling the learning progressions of computational thinking of primary grade studentsf. In *Proceedings of the ninth annual international ACM conference on International computing education research*, pages 59–66. ACM, 2013. 1, 2, 10, 15, 16, 32

Irena Nančovska Šerbec, Špela Cerar, and Alenka Žerovnik. Developing computational thinking through games in scratch. *Education and Research in the Information Society*, pages 21–30, 2018. 1, 16

Juha Sorva. *Visual program simulation in introductory programming education.* G4 monografiaväitöskirja, Aalto University School of Science, 2012. URL `http://urn.fi/URN:ISBN:978-952-60-4626-6`. 1

Alaaeddin Swidan, Felienne Hermans, and Marileen Smit. Programming misconceptions for school students. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 151–159. ACM, 2018. 1, 2, 6, 7

Giovanni Maria Troiano, Sam Snodgrass, Erinç Argımak, Gregorio Robles, Gillian Smith, Michael Cassidy, Eli Tucker-Raymond, Gillian Puttick, and Casper Harteveld. Is my game ok dr. scratch? exploring programming and computational thinking development via metrics in student-designed serious games for stem. In *Proceedings of the 18th ACM International Conference on Interaction Design and Children*, pages 208–219, 2019. 1, 32

Christiane Gresse Von Wangenheim, Jean CR Hauck, Matheus Faustino Demetrio, Rafael Pelle, Nathalia da Cruz Alves, Heliziane Barbosa, and Luiz Felipe Azevedo. Codemaster–automatic assessment and grading of app inventor and snap! programs. *Informatics in Education*, 17(1):117–150, 2018. 15

Seungwon Yang, Carlotta Domeniconi, Matt Revelle, Mack Sweeney, Ben U Gelman, Chris Beckley, and Aditya Johri. Uncovering trajectories of informal learning in large online communities of creators. In *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*, pages 131–140. ACM, 2015. 1, 11

Yunwen Ye and Kouichi Kishida. Toward an understanding of the motivation of open source software developers. In *Proceedings of the 25th international conference on software engineering*, pages 419–429. IEEE Computer Society, 2003. 10

Appendix A

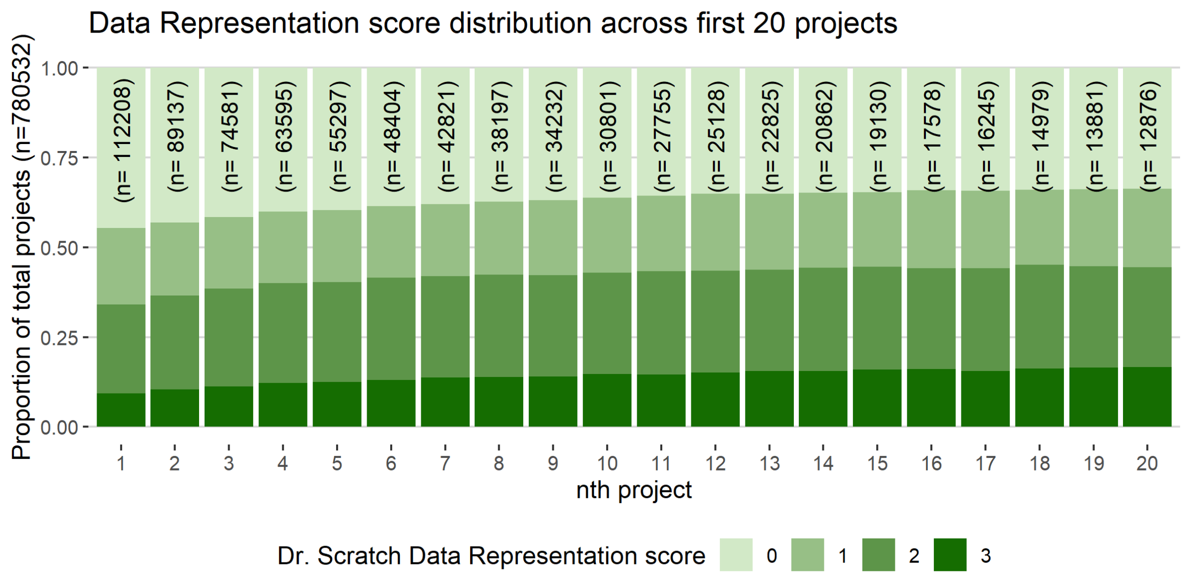


Figure 1: Distribution of data representation scores across first 20 projects

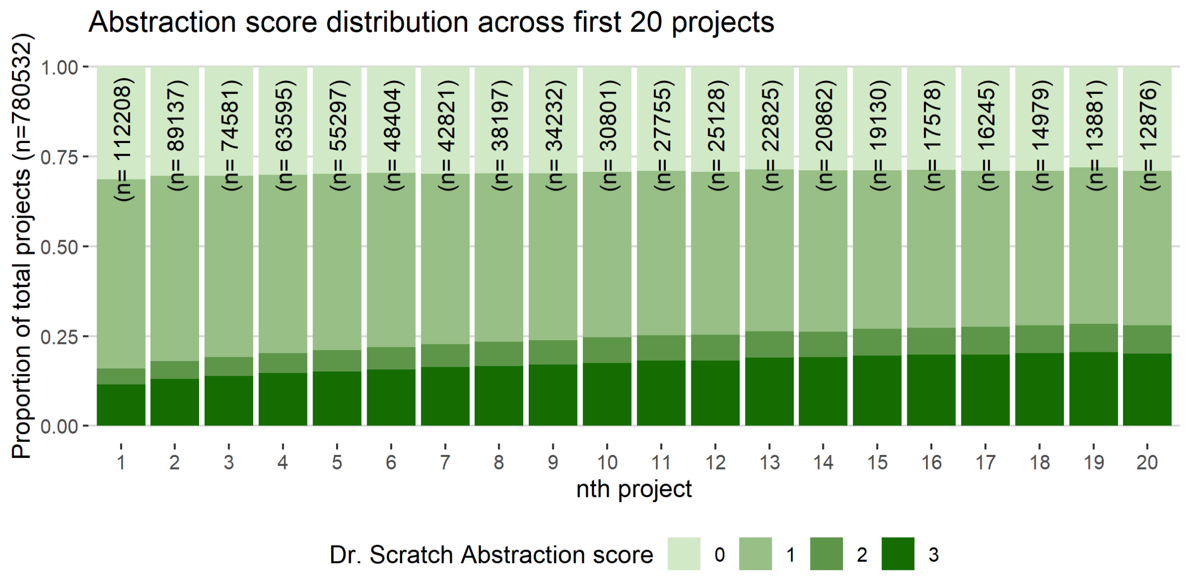


Figure 2: Distribution of abstraction & problem decomposition scores across first 20 projects

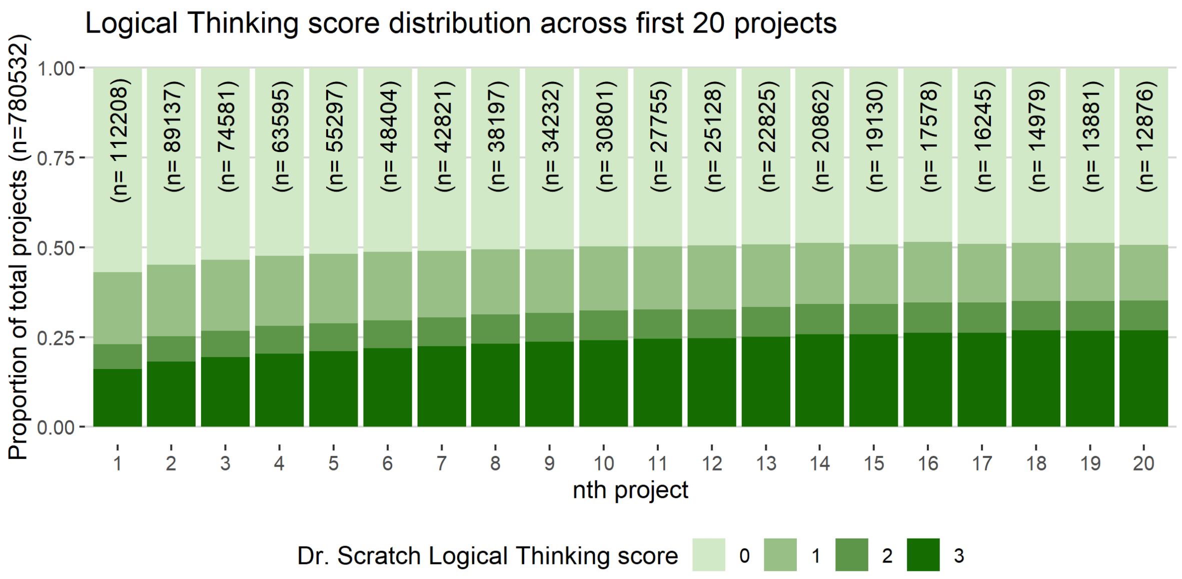


Figure 3: Distribution of logical thinking scores across first 20 projects

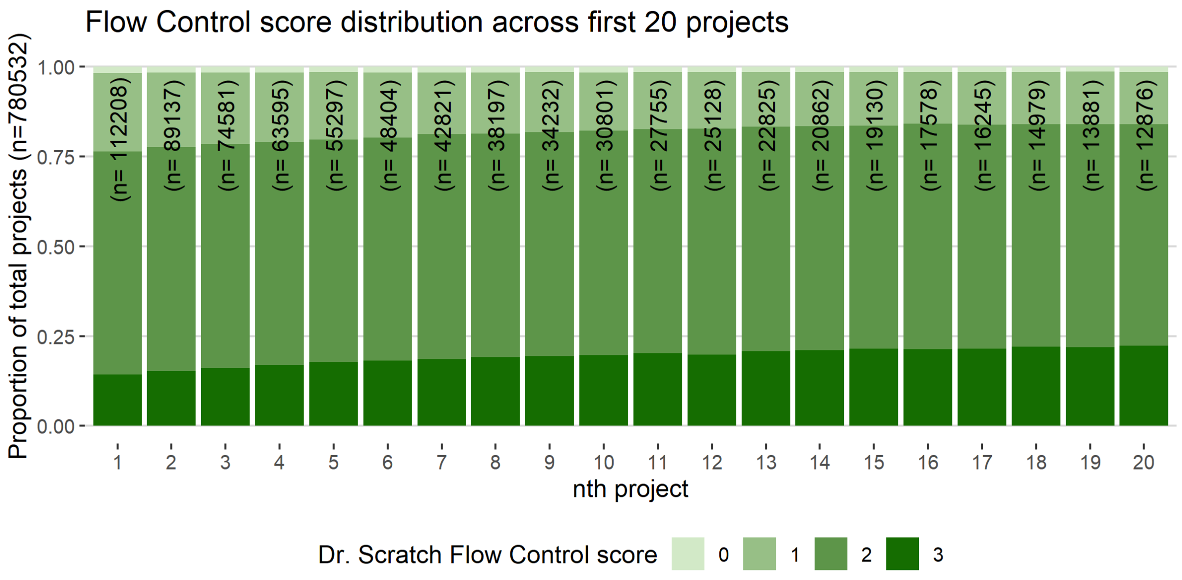


Figure 4: Distribution of flow control scores across first 20 projects

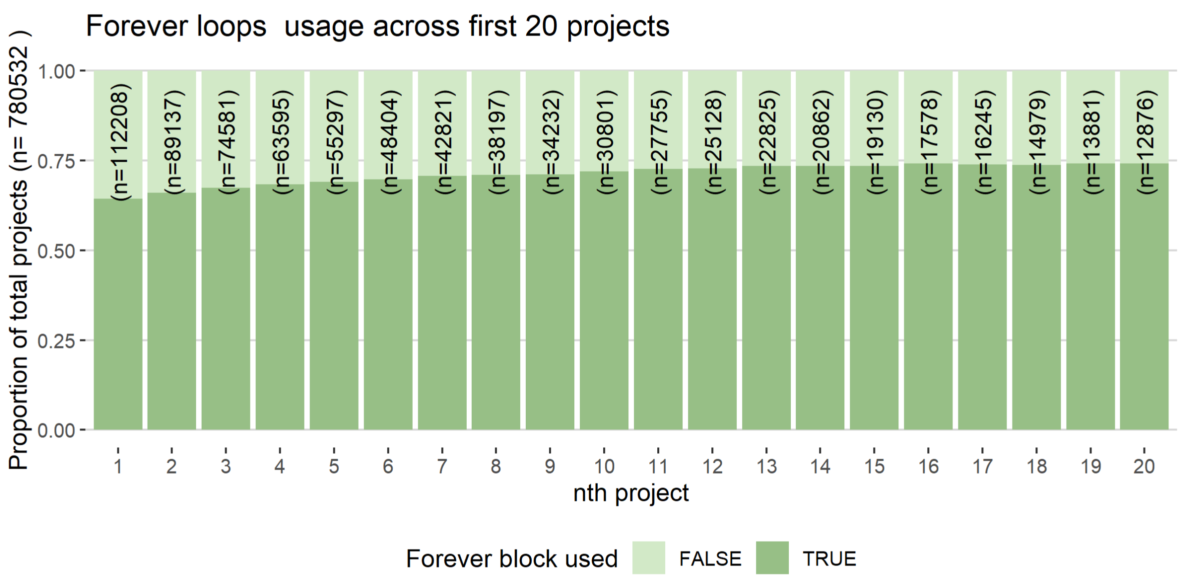


Figure 5: Caption

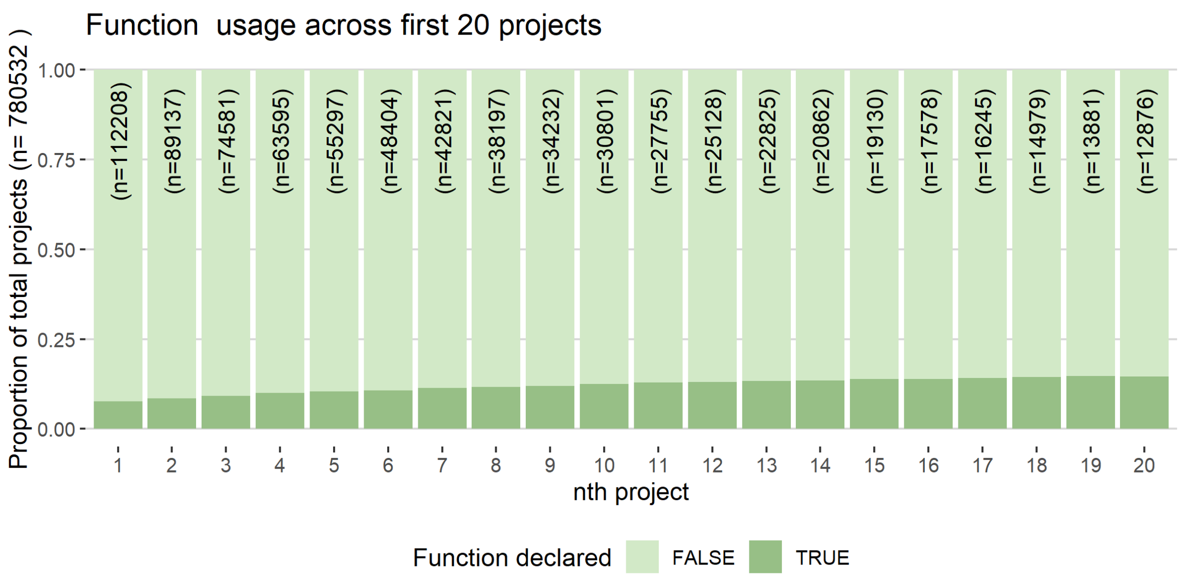


Figure 6: Caption

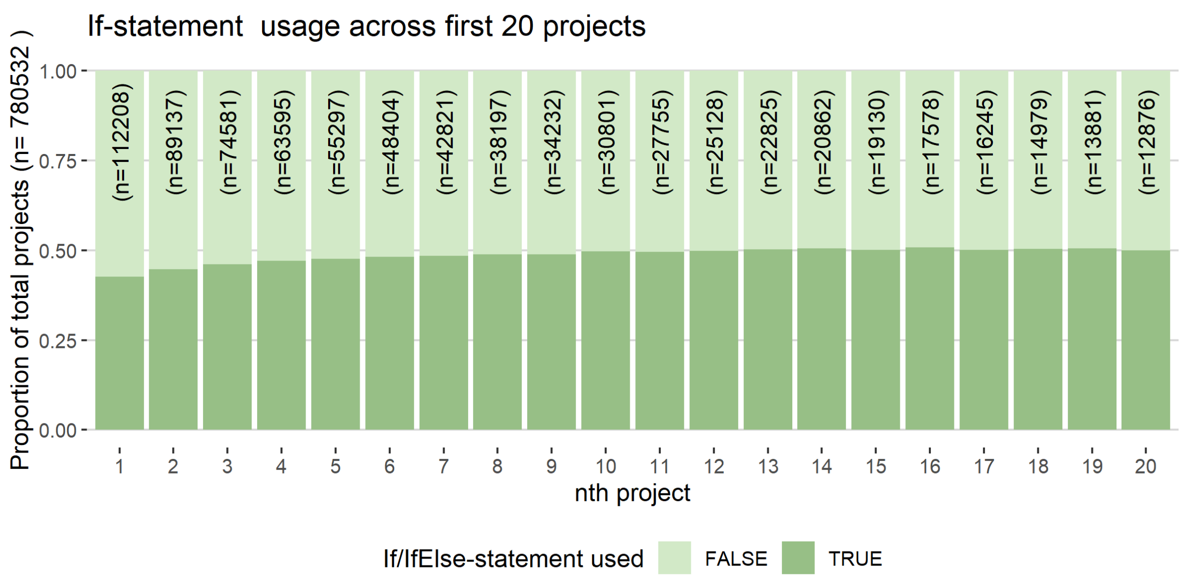


Figure 7: Caption

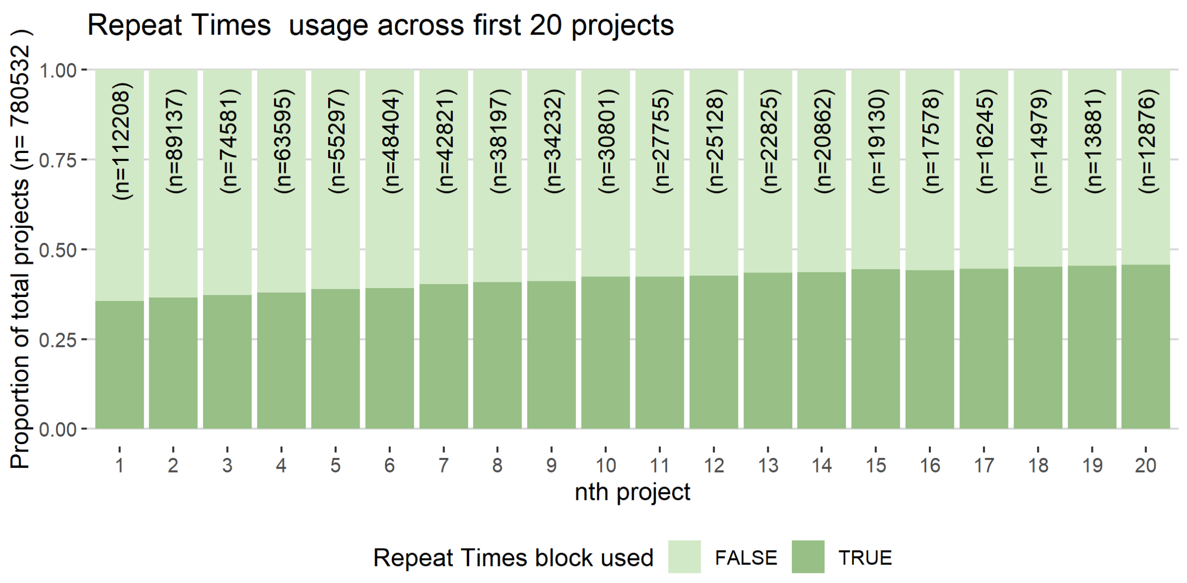


Figure 8: Caption

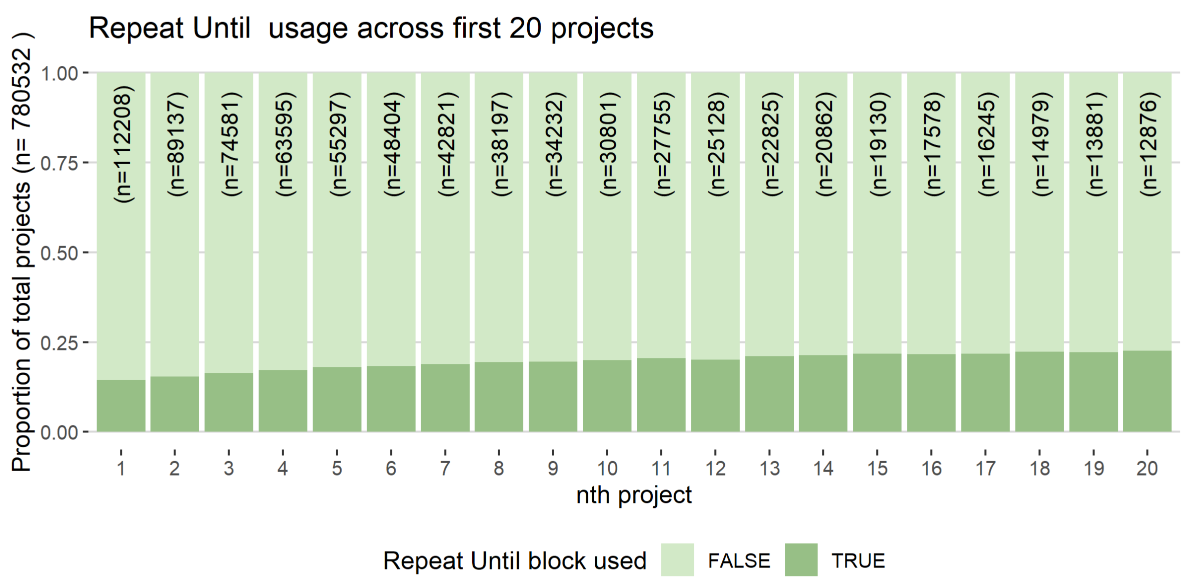


Figure 9: Caption

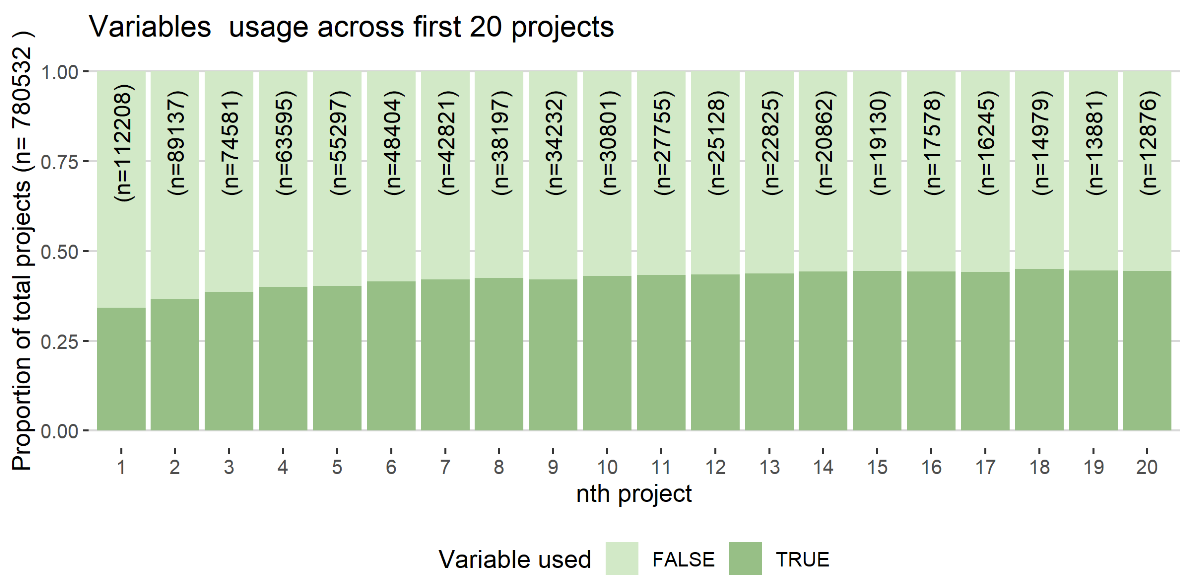


Figure 10: Caption