How Kids Code and How We Know: An Exploratory Study on the Scratch Repository

(anonymized submission)

ABSTRACT

Block-based programming languages like Scratch, Alice and Blockly are becoming increasingly common as introductory languages in programming education. There is substantial research showing that these visual programming environments are suitable for teaching programming concepts. But, what do people do when they use Scratch? In this paper we explore the characteristics of Scratch programs. To this end we have scraped the Scratch public repository and retrieved 250,000 projects. We present an analysis of these projects in three different dimensions. Initially, we look at the types of blocks used and the size of the projects. We then investigate complexity, used abstractions and programming concepts. Finally we detect *code smells* like large scripts, dead code and duplicated code blocks. Our results show that 1) most Scratch programs are small, however Scratch programs consisting of over 100 sprites exist, 2) programming abstraction concepts like procedures are not commonly used and 3) Scratch programs do suffer from code smells including large scripts and unmatched broadcast signals.

General Terms

terms

Keywords

Scratch, block-based languages, programming practices, code smells, static analysis

1. INTRODUCTION

Scratch [17] is a programming language developed to teach children programming by enabling them to create games and interactive animations. The public repository of Scratch programs contains over 14 million projects. Scratch is a *block-based* language: users manipulate blocks to program.

Block-based languages have existed since the eighties, but have recently found adoption as tools for programming education. In addition to Scratch, also Alice [4], Blockly[1] and App Inventor [19] are block-languages aimed at novice programmers.

Several studies have shown that block-based languages are powerful as tools for teaching programming [12, 15, 5, 16]. Previous works involving static analysis of Scratch programs have evaluated the application of various programming concepts in Scratch projects [9, 13]. Recent works have focused on bad programming practices within Scratch programs [11], and automated quality assessment tools have been proposed for identifying code smells [6] and bad programming practices [2, 13]. A recent controlled experiment found that long scripts and code duplication decreases a novice programmer's ability to understand and modify Scratch programs [7].

The goal of this paper is to obtain a deep understanding of how people program in Scratch, to analyze the characteristics of Scratch programs, and to quantitatively evaluate the use of programming abstractions and concepts. Moreover, knowing that bad programming habits and code smells can be harmful [7], we also want to explore whether they are common. To address this goal, we answer the following research questions:

RQ1 What are the size and complexity characteristics of Scratch programs?

RQ2 Which coding abstractions and programming concepts and features are commonly used when programming in the Scratch environment?

RQ3 How common are code smells in Scratch programs?

Our study is based on data from the Scratch project repository. By scraping the list of recent projects,[2] we have obtained 250,166 public Scratch projects and performed source code analysis on them. To the best of our knowledge, this is the first large-scale exploratory study of Scratch projects.

The contributions of this paper are as follows:

- A public data set of 233,491 non-empty Scratch projects (Section 3.1)
- An evaluation of the data set in terms of size, complexity, programming concepts and smells (Section 4)
- A discussion of the implications of our findings for educational programming language designers (Section 5)

[1] `https://developers.google.com/blockly/`
[2] `https://scratch.mit.edu/explore/projects/all/`

2. RELEVANT SCRATCH CONCEPTS

This paper is by no means an introduction into Scratch programming, we refer the reader to [3] for an extensive overview. To make this paper self-contained, however, we explain a number of relevant concepts in this section.

Scratch is a block-based programming language aimed at children, developed by MIT. Scratch can be used to create games and interactive animations, and is available both as a stand-alone and as a web application. Figure 1 shows the Scratch user interface. The main concepts in the Scratch programming environment are:

Sprites Scratch code is organized by 'sprites': two dimensional pictures each having their own associated code. Scratch allows users to bring their sprites to life in various ways, for example by moving them in the plane, having them say or think words or sentences via text balloons, but also by having them make sounds, grow, shrink and switch costumes. The Scratch program in Figure 1 contains two one sprites, the cat, which is Scratch's default sprite and a piano. The code in Sprite1 will cause the cat to move right when the right arrow is pressed, and when the green flag is clicked it will continuously sense touching the piano.

Scripts Sprites can have multiple code blocks, called scripts. The Scratch code in Figure 1 has two distinct scripts, one started by clicking on the green flag and one by pressing the space bar. It is possible for a single sprite to have multiple scripts initiated by the same event. In that case, all scripts will be executed simultaneously.

Events Scratch is *event-driven*: all motions, sounds and changes in the looks of sprites are initiated by events called Hat blocks[3]). The canonical event is the `when Green Flag clicked`, activated by clicking the green flag at the top of the user interface. In addition to the green flag, there are a number of other events possible, including key presses, mouse clicks and input from a computer's microphone or webcam. The Scratch code in Sprite1 in Figure 1 contains two events: `when Green Flag clicked` and `when <right arrow> key pressed`, each with associated blocks.

Signals Events within Scratch can be user generated too: users can broadcast a message, for example when two sprites touch each other, like in Figure 1. All other sprites can then react by using the `when I receive` Hat block. In Figure 1, Sprite1 broadcasts 'bump' when the cat touches the Piano.

Custom blocks Scratch users can define their own blocks, which users can name themselves, called custom blocks. The creation of custom blocks in Scratch is the equivalent of defining procedures in other languages [13]. Because the term 'procedures' is common in related work, we will refer to custom blocks as 'procedures' in the remainder of this paper. Procedures can have input parameters of type string, number, and boolean. When a user defines a procedure, a new Hat block called `define` appears, which users can fill with the implementation of their block.

[3] `http://wiki.scratch.mit.edu/wiki/Hat_Block`

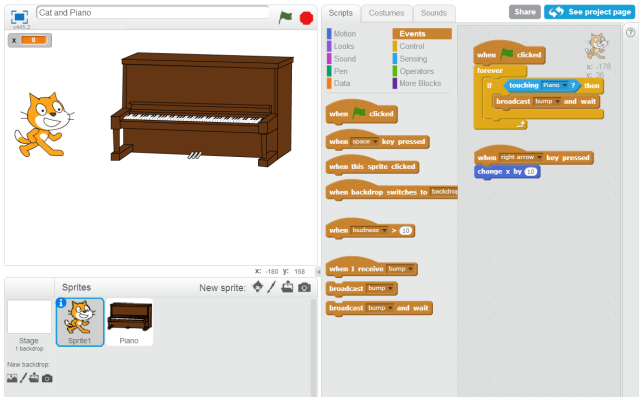

Figure 1: The Scratch user interface consisting of the 'cat' sprite on the left, the toolbox with available blocks in the category 'Events' in the middle and the code associated with the sprite on the right.

3. RESEARCH DESIGN AND DATASET

The main focus of this study is to understand how people program in Scratch by analyzing the characteristics of Scratch projects. To answer our three research questions, we conducted an empirical quantitative evaluation of project data we collected from the Scratch project repository. In the following paragraphs we describe the dataset, the process and the tools we used for analyzing it, and the methods we followed for detecting code smells.

3.1 Dataset

We obtained the set of Scratch projects by using a scraping program. Our scraping program called Kragle starts by reading the Scratch projects page[2] and thus obtains project ids of projects that were most recently shared. Subsequently, Kragle retrieves the JSON code for each of the listed projects.

We ran Kragle on March 2nd 2016 for 24 hours and, in that time, it obtained a little over 250,000 projects. Out of the 250,166, we failed to parse and further analyze 2,367 projects due to technical difficulties with the provided JSON files. Kragle, as well as all scraped projects and our analysis files are available.[4]

Once we obtained the Scratch projects, we parsed the JSON files according to the specification of the format.[5] This resulted in a list of used blocks per project, with the sprites and the stage of the project. We cross referenced also all blocks with the Scratch wiki to determine the shapes and the category of all blocks. For example, `When Green Flag Clicked` is a *Hat block* from the *Events category*.

3.2 Data analysis

All scraped project data, including the list of used blocks and parameters, were imported in a relational database. We used SQL queries, which are also made available,[4] for filtering, aggregating and extracting all statistical data required to address our three research questions. We also randomly sampled and manually inspected edge cases in the results,

[4] `https://github.com/ScratchLover42/ICER-Data-Code` (temporary link to protect the anonymity of the authors)

[5] `http://wiki.scratch.mit.edu/wiki/Scratch_File_Format_(2.0)`

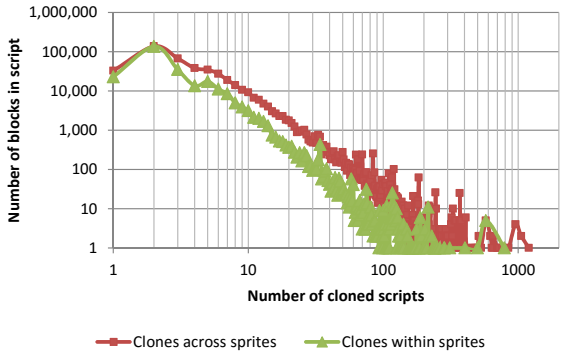

Figure 2: Number of cloned scripts of different block sizes across and within sprites

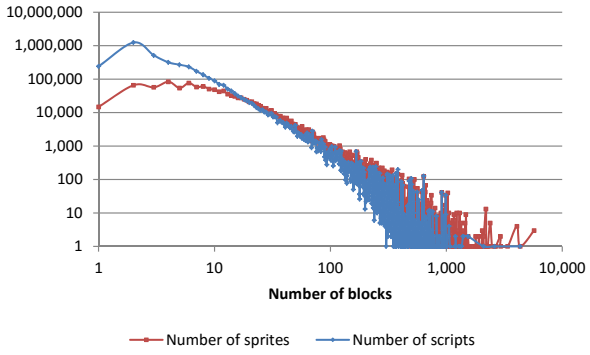

Figure 3: Size of sprites and scripts in number of blocks

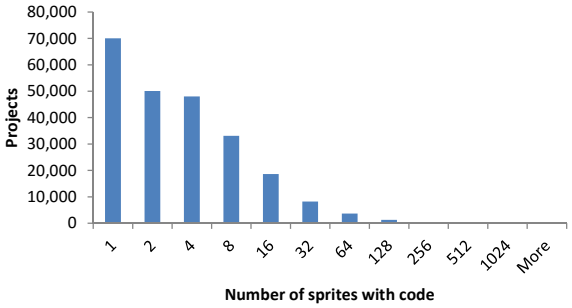

Figure 4: Number of sprites in the analyzed projects

for example empty or overly complex projects. Data for these cases are provided as part of the dataset.[4]

For RQ1 we measured the size of projects based on the number of blocks in scripts and sprites and we calculated descriptive statistics, which are presented in Section 4.1. For measuring the complexity of the scripts we used the McCabe cyclomatic complexity metric [10], a quantitative measure of the number of independent paths through a program's source code. This is calculated per script by counting the number of decision points in the script plus one. In Scratch, decision points can be the `if` and `if else` blocks.

For RQ2, we used the data on the code blocks and their categories to perform statistical analysis of applied programming abstractions and concepts. Similarly to [9], we consider the use of certain blocks to indicate that a programming abstraction or concept is being used in a certain project. In Section 4.2 we present the results related to the utilization of procedures, variables, loops, conditional statements, user interactivity and synchronization.

For RQ3, we focused on four types of code smells: duplicated code, dead code, large script and large sprite. For the duplicated code smell analysis, our first step was to specify what we consider a code clone in the context of Scratch programming: a script that is composed of a set of blocks of the same type connected in the same way and is repeated within or across sprites of the same project. For the identification of clones we did not take into account the values of the parameters that may be used in the blocks, so that two blocks that only differ in the values of parameters are considered to be equal. We also examined the case of clones with the same parameter values, and we refer to them as *exact clones*. The next step in the analysis was to determine the minimum size of the scripts that are considered clones instead of incidentally similar. We examined the number of detected clones for different script sizes and present the results in Figure 2. Based on this distribution, we opted to adopt the number also used by the authors in [13], which is the minimum size of 5 blocks per script.

To examine the long method and the large class smells, we consider them in the context of Scratch as large script and large sprite smells respectively. For these two smells we use the number of blocks as the size metric. Figure 3 presents the number of blocks in the scripts and the sprites of our dataset. We used these numbers to split the dataset and retrieve the top 10% largest scripts and sprites, as is commonly done in both source code analysis [1] and analysis of end-user programming artifacts like spreadsheets [8]. Using that strategy, we set the thresholds for the calculation of the large script and large sprite: it is 18 blocks and 59 blocks respectively. The results we obtained using these thresholds are presented in Section 4.3.

4. RESULTS

In the following sections, for each of the research questions, we describe the results obtained through the analysis of the 247,798 Scratch projects in our dataset.

4.1 Program Size and Complexity

The dataset contains a relatively small number of projects without any code: 14,307 (5.77%). Through random manual sampling we found that in some cases these projects contains only sprites and costumes, but no code, while in other projects they were entirety empty apart from the Scratch cat added by default. Since these projects are empty in terms of code we excluded them from further analysis, leaving the final number of analyzed non-empty projects to 233,491.

In Table 1 we summarize the statistics for the analyzed metrics. We use the mean value and the five-number summary to describe the dataset in terms of the number of sprites with code per project (including the stage sprite) and the number of scripts and blocks per project. Figures 4, 5 and 6 plot the distribution of these size metrics.

We find that the majority of Scratch projects are small; 75% of the projects have up to 5 sprites, 12 scripts and 76 blocks, while one fourth of the projects have up to 12 blocks. On the other end, 5% of the projects (11,712) have more than 18 sprites and 4.8% (11,214) consist of more than 500 blocks. The analysis also highlighted some surprisingly

		mean	min	Q1	median	Q3	max
Size	Sprites with code per project	5.68	1	1	2	5	525
	Scripts per project	17.35	1	2	5	12	3,038
	Number of blocks per project	154.55	1	12	29	76	34,622
	Blocks in Stage per project	4.80	0	0	0	3	2,613
	Blocks in Sprites per project	115.57	0	10	26	68	34,613
	Blocks in Procedures per project	34.17	0	0	0	0	20,552
Complexity	McCabe Cyclomatic Complexity (CC) per script	1.58	1	1	1	1	246
	McCabe CC per procedure script	3.75	1	1	2	4	183
Procedures	Procedures per project with procedures	11.50	1	1	2	6	847
	Arguments per Procedure	0.95	0	0	0	1	53
	Numerical arguments per procedure with arguments	1.73	0	1	1	2	22
	Text arguments per procedure with arguments	0.28	0	0	0	1	24
	Boolean arguments per procedure with arguments	0.13	0	0	0	0	14
	Calls per procedure	2.14	0	1	1	2	526
	Scripts with calls per procedure	1.13	0	1	1	1	59
Programming concepts	Variables per project	2.06	0	0	0	1	340
	Scripts utilizing variable	4.97	1	1	3	5	1,127
	Lists per project	0.55	0	0	0	0	319
	Conditional statements per project	10.02	0	0	0	3	5,950
	Loop statements per project	7.65	0	1	2	5	2,503
	User input blocks per project	4.77	0	0	1	4	1,889
	Broadcast-receive statements per project	8.57	0	0	0	2	2,460

Table 1: Summary statistics from the dataset of 233,491 non-empty Scratch projects

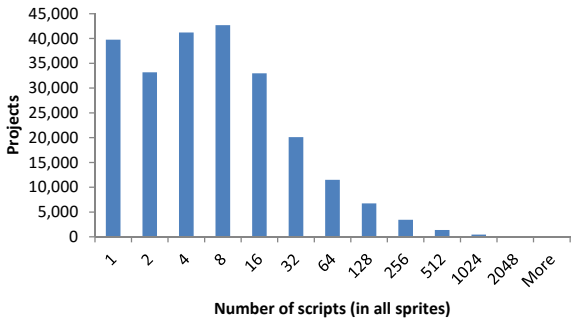

Figure 5: Number of scripts in the analyzed projects

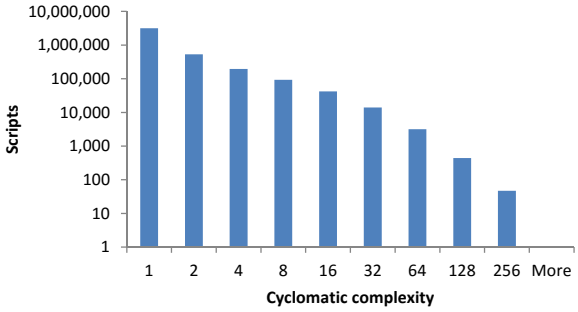

Figure 7: McCabe cyclomatic complexity of the 4,049,356 analyzed scripts

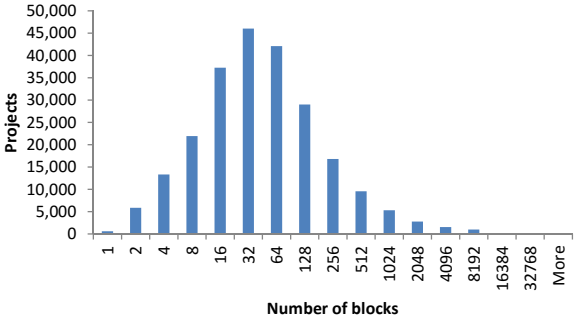

Figure 6: Number of blocks in the analyzed projects

large projects: 135 with more than 300 sprites and even 30 projects with more than 20,000 blocks, whose Scratch identifiers are made available for further inspection.[4]

The number of blocks metric was further analyzed to understand code organization. The majority of Scratch code—74.78% out of 36,085,654 blocks—is written within sprites. An additional 3.1% of the total blocks are found in the stage class. More interestingly, the remaining 22.11% are blocks within defined procedures, which are found in only 7.7% (17,979) of the projects. The projects that contain procedures use them a lot; almost half of their total blocks (48.81%) are within procedures.

We further analyzed the utilization frequency of the different block shapes and categories, as defined in the Scratch documentation. Figures 8 and 9 present the results in terms of number of blocks from the total 36,085,654 blocks in the dataset projects. The most commonly used blocks are from the Control and Data categories. The Others category includes the blocks related to procedure calls and arguments.

To understand the complexity of the Scratch projects in

	Number of projects	%
Retrieved	250,166	
Analyzed	247,798	
Non-empty (used for statistics)	**233,491**	
Projects with:		
Procedures	17,979	7.70%
Recursive procedures	1,052	0.45%
Variables	73,577	31.51%
Lists	9,358	4.01%
Conditional statements	92,959	39.81%
User input blocks	131,314	56.24%
Loop statements	180,210	77.18%
`repeat until <condition>`	31,739	13.59%
`broadcast - receive`	69,039	29.57%
Cloned scripts across sprites	59,634	25.54%
Cloned scripts within sprites	23,671	10.14%
Cloned procedures	4,945	2.12%
Cloned blocks across sprites	60,554	25.93%
Exact clones across sprites	27,574	11.81%
Exact clones within sprites	2,043	0.87%
Dead code	65,760	28.16%
Large scripts	69,521	29.77%
Large sprites	31,954	13.68%

Table 2: Elements and characteristics of the projects in the dataset

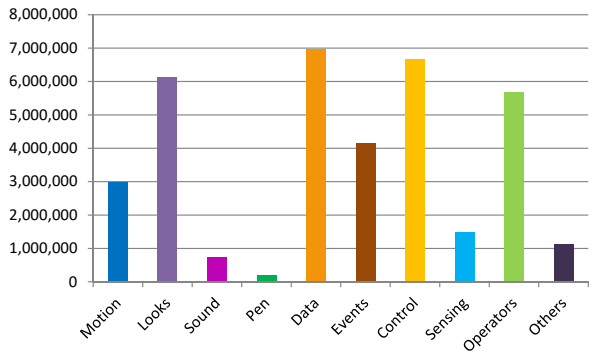

Figure 8: Number of blocks from each category in the analyzed projects

our dataset, we use the McCabe cyclomatic complexity. The results of this metric per script are plotted in Figure 7. The majority (78.33%) of 4,049,356 scripts contain no decision points, while 13.08% have a cyclomatic complexity of 2, containing exactly one decision point. The complexity is higher, over 4, for 3.67% of the scripts. The analysis also highlighted 209 scripts with a cyclomatic complexity over 100 and up to 246.[4] Cyclomatic complexity was greater (mean value of 3.32) in defined procedures, with 56.46% of the procedures having at least one decision point.

RQ1: The majority of Scratch projects are small and simple; 75% of the projects have up to 5 sprites, 12 scripts, 76 blocks and no decision points. Most code is written in sprites. There exist surprisingly large and complex projects.

4.2 Programming Abstractions and Concepts

The first method for abstraction that we investigate are

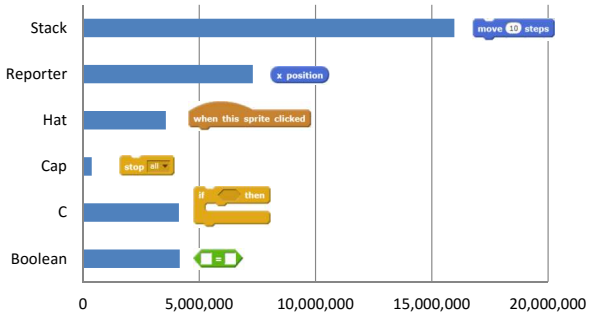

Figure 9: Number of blocks of each shape in the analyzed projects

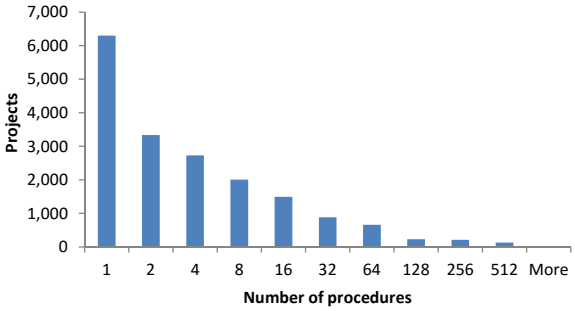

Figure 10: Number of procedures for the 17,979 projects that include at least one

procedures. In the dataset we found 206,799 procedures in 17,979 (7.7%) projects. As summarized in Table 1, the projects that contain procedures have an average of 11.5 procedures, but with 53.59% of these projects having up to 2. Figure 10 shows the distribution of procedures in projects. Regarding procedure arguments, we found that 55.57% have no arguments and 19.48% have only one (shown in Figure 11). The majority of procedure arguments (80.59%) are numeric, and the least used argument type is the boolean one—6.23% of the total procedure arguments, found in 5.32% of the procedures.

The use of procedures in projects was further investigated through the procedure calls, summarized in Figure 12. Most procedures are called exactly once (62.32% of them) or twice (14.30%) and from exactly one script (85.92% of them). Examining the origin of procedure calls, we observed that

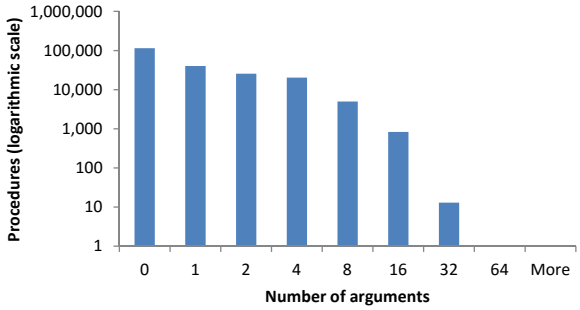

Figure 11: Number of arguments for the procedures in the dataset

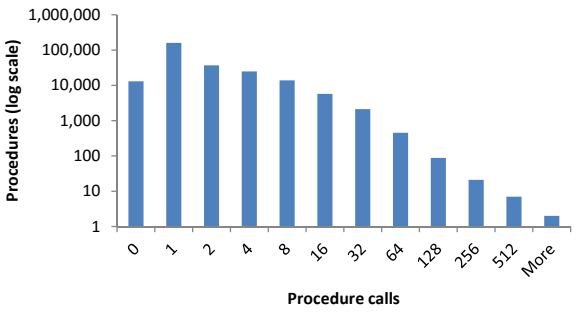

Figure 12: Number of calls of each procedure in the dataset

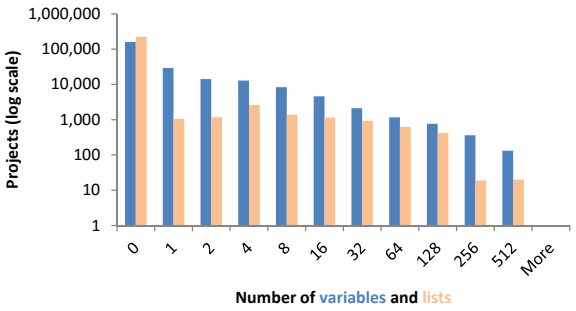

Figure 13: Number of variables and lists used in the projects

most of the calls (56.09%) originate from other procedures, and even 1.06% originate from the same procedure, making them recursive calls. These recursive procedures are found in 1,052 projects, whose identifiers are made available.[4]

As shown in Table 2, almost one-third of the projects use variables and a small number (4.01%) use lists. The number of variables that is being used is also limited, with only 7.48% of the projects having 5 or more variables. The distribution of variable and list utilization is shown in Figure 13. Exceptional cases exist: the analysis highlighted 842 projects with more than 100 variables and with a maximum of 340. Examining the initialization of variables through the `set <variable> to <value>` blocks, we found that for 4.83% of all variables this was missing. While failing to initialize a variable in Scratch will not result in a runtime error as in some other programming languages, correctly setting the initial state of the program is important [2].

Regarding program control features, conditional statements (blocks `if <condition> then` and `if <condition> then else`) are used by 39.81% of the projects. Loops (blocks `repeat <times>`, `forever` and `repeat until <condition>`) are more common, used by 77.18% of the projects. The most common of the three is the `forever` block, accounting for 51.86% of all loops and the least common one is the `repeat until <condition>` block, accounting for 11.57% and used in 13.59% of the total projects.

Investigating user interactivity functionality, we found that 56.24% of the projects in the dataset contain user input blocks—an average of 8.48 blocks per such project. Table 3 lists the frequency of use of user input controls. We do not include the `when Green flag pressed` block here, as this is just used to start a Scratch program and hence cannot really

Block	Projects	Occurrences
`when <> key pressed`	71,096	294,771
`when this sprite clicked`	39,179	198,342
(Sensing) `key <> pressed?`	37,919	291,657
(Sensing) `ask <> and wait`	19,039	66,850
(Sensing) `mouse down?`	9,115	54,079
(Sensing) `<attrib> of <>`	9,068	155,468
(Sensing) `mouse X`	5,977	27,321
(Sensing) `mouse Y`	3,940	22,035
`when <sensor> > <value>`	705	1,570
(Sensing) `video <> on <>`	434	1,397

Table 3: Frequency of use of user input blocks in the 233,491 projects of the dataset

be considered input into the program. The most commonly used user input block is the `when key pressed`, found in 71,096 (30.45% of the total) projects. The most frequently used parameter for the `key` attribute is the space key, followed by the arrows and then the letters and numbers.

Users can define their own events, using the blocks `broadcast`, `broadcast and wait` and `when I receive`.These blocks are used by 29.57% of the projects. `broadcast and wait` is rarely used, in only 3.87% of the projects.

RQ2: A small number of projects (8%) use procedures, but they use them a lot and for more complex code. Most procedures are called once or twice, from a single script which, in more than half of the cases, is another procedure. Recursive procedure calls exist in 1,052 (0.5% of the total) projects. One third of the projects use variables, sometimes without initializing them. 40% of the projects contain conditional statements and 77% contain loops, but conditional loops are rarely used. More than half of the projects are interactive. 30% of the projects use broadcast and receive blocks.

4.3 Code smells

The duplicated code smell is the first smell that we examine. As explained in Section 3.2, we use 5 as the minimum number of blocks for the identified clones. In total, in the dataset we found 170,532 scripts cloned across sprites in 59,634 (25.54% of the total non-empty) projects. 726,316 copies of these scripts were found, making each clone being copied an average of 4.26 times. Figure 14 plots the distribution of clones across projects. The majority of projects contain up to two cloned scripts; 7.24% of the projects contain three or more. Figure 15 plots the number of copies of the identified clones. It is of interest that 79,378 (46.55%) of the identified clones are copied three or more times, and even in 585 cases from 411 projects they are copied more than 50 times and up to 974.[4]

We further inspected which of the identified clones were duplicated only within the same sprite: 63,682 (37.34% of the total) clones, in 10.14% of all projects. Procedure clones were measured to 12,878 (7.55% of the total) clones, in 2.12% of the projects.

Exact clones were found in 11.81% of the total projects. Their total number was 66,750 (39.14% of the total) clones. Exact clones in the same sprite are cases of clearly redundant code. These were rare, found in only 0.87% of the projects.

Apart from whole scripts we also examined cases where scripts differed only in the first (Hat) block. This way we

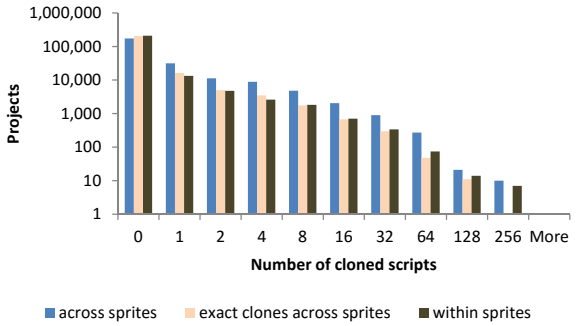

Figure 14: Cloned scripts in the dataset projects

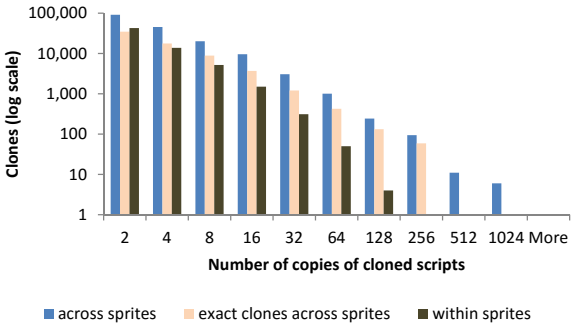

Figure 15: Number of copies of the identified clones

examine if Scratch programmers assign the same functionality to handle different types of events. Cloned functionality blocks are found to be rare: without considering the first block, only 2,243 additional clones were found in 920 projects.

The second smell that we examine is the dead code smell. We identify four types of dead code: (1) procedures that are not invoked, (2) unmatched broadcast-receive messages, (3) code that is not invoked and (4) empty event scripts. Investigating the first type, we find that a significant number of the defined procedures (13,036 or 5.06%) are not called in the projects. This is also shown in Figure 12 and it occurs in 2,079 projects. For the second type, we examined the broadcast-receive messages and found that they are not always synchronized: 3,33% of the `when I receive` blocks were found to wait for a message that is never being broadcasted, while 4,4% of the `broadcast` blocks broadcast a message that is not being received. This lack of syncronization occurs in 18,669 (7.99% of the total) projects.

The third and fourth cases are incomplete scripts. They are either never invoked due to the lack of a starting `when <trigger>` block from the Scratch Events or Control category, or are comprised of only a `when <trigger>` block without any functionality. A total of 322,475 scripts like that were found in 56,890 (24.36% of the total) projects. The majority of these scripts (86.6%) are scripts missing the starting block. Examining the size of these dead scripts, 72.34% are composed of a single block. As shown in Figure 16, however, considerably large dead scripts exist; 2,358 of these scripts originating from 1,553 different projects have more than 30 blocks and up to 2,610.[4]

The number of projects exhibiting the dead code smell, considering all four types of dead code combined, is 65,760 (28.16% of the total projects).

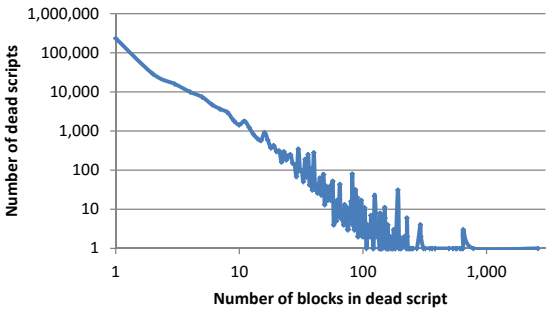

Figure 16: Size of scripts identified as dead code

Finally, we examine the large script and the large sprite smells. The thresholds we use for the identification of large scripts and large sprites are 18 blocks and 59 blocks respectively, as explained in Section 3.2. The number of projects exhibiting the large script smell, i.e., containing at least one script with 18 or more blocks, is 69,521 (29.77% of the total projects) and the number of projects with the large sprite smell is 31,954 (13.68%).

RQ3: Code clones are found in 26% of the projects, with almost half of the clones copied three or more times, in the same or across sprites. 28% of the projects contain code that is never invoked, and thus exhibit the dead code smell. In some cases these scripts are large. The large script smell is found in 30% of the projects and the large sprite one in 14%.

5. DISCUSSION

5.1 Implications

We believe a large scale study of programs like ours can help language designers to tailor their language. In this section we highlight directions in which our study could support language design. There are many other implications to be considered, which is why we have made our dataset public.

5.1.1 Popularity of different block types

Our analysis shows that some categories of blocks are rarely used, like the 'Pen' blocks, of which only 194,885 occur within 19,090 (8.17% of the total) programs. Hence, in future changes to the language, 'Pen' blocks might be less important to users to support or maintain.

5.1.2 Dead code

In our analysis, we find that more than one quarter of the Scratch projects contain dead 'scripts': scripts that are never invoked due to the lack of a starting `when <trigger>` block from the Scratch Events or Control category, scripts that are comprised of only a `when <trigger>` block without any functionality, procedures that are not invoked, or unmatched broadcast-receive messages.

In a sense, the dead scripts are harmless, as they are not executed. However, they do cause 'visual clutter' and might be distracting to novice programmers, as it might be hard to see which scripts are dead. In contrast to other visual educational languages, Scratch does not indicate scripts that are dead. LEGO Mindstorms, for example, does give the user such feedback by making unconnected blocks gray.

Looking at the number of unconnected blocks, we hypothesize that Scratch programmers have a need for a separate workspace to store unconnected blocks temporarily. We envision that would be like the 'backpack' meant to move scripts across sprites. In order to help novice programmers keep their code clean, the programming interface could actively encourage users to move unconnected blocks to that workspace when they exit the environment.

5.1.3 Exact clones between sprites

With occurrences in 11% of the Scratch projects in our dataset, the use of exactly identical clones between sprites is relatively common. In a sense, the Scratch users are not to blame here, as Scratch does not support procedure calls between sprites, only within them. So in many cases there is no way to share the functionality other than by making a copy. We are not aware of the underlying rationale of the Scratch team that lead to this decision, however it seems that a large part of the Scratch users would use the functionality to call procedures between sprites.

5.1.4 Sharing of scripts and procedures

Investigating the use of clones between projects, we observe that there are 1,700 scripts that are used in multiple projects, sometimes as often as in 1,600 different projects. This seems to indicate that there are common patterns in Scratch projects, which means it might be very beneficial to Scratch programmers if they could not only share their projects, but also share some of their functionality, for others to use, like a library. An example of such a library could be: functions for platforming games, including the movement of a player, collision detection and the implementation of 'lives'. This might empower new Scratch users to get started faster.

5.2 Threats to validity

A threat to the validity of this study is the fact that we did not scrape a random sample, but the most recent 250,000 projects. It could be the case that the programming habits of Scratch users are changing over time. However, we counterbalanced that by using a large dataset which comprises of around 2% of all 14 million shared Scratch projects.[6].

Furthermore we use the number of blocks in the Scratch projects as a measure for the length of a program, while this does not exactly correspond to the 'length' of a program in lines, and there can be multiple Scratch blocks on one line. For example, in Figure 1 the 'if' and 'touching' blocks are present on the same 'line'. We however believe that the number of blocks is a good proxy for size, and we plan a future experiment in which we will compare 'lines of Scratch code' to 'number of blocks'.

6. RELATED WORK

The evaluation of block-based languages in general, and Scratch in particular, as tools for programming education has received significant research attention during the past years. A number of studies have been carried out on the understanding of programming concepts and the programming practices of novice programmers in block-based environments, on the programming skills they develop, and on the quality of Scratch programs.

[6]`https://scratch.mit.edu/statistics/`

For example, a study on the internalization of programming concepts with Scratch with 46 students was presented in [12]. Concepts like loops, conditional loops, message passing, initialization, variables and concurency were examined, and it was found that students had problems with the last three. In a later study with an equal set of subjects [11] the same authors identified two bad programming habits in Scratch, namely bottom-up development and extremely fine-grained programming. They connected the later to the reduced use of if-blocks and finite loops and the increased use of infinite loops, a finding that is verified by our study. In [18] 29 Scratch projects created from 60 students working in groups were evaluated based on a list of criteria related to programming concepts, code organization and usability design.

Most related to our study for the second research question of programming abstractions and concepts is the work by Maloney *et al.* [9], who analyzed 536 Scratch projects for blocks that relate to programming concepts including loops, conditional statements, variables, user interaction, synchronization, and random numbers. Compared to their findings, our investigation reveals increased use of the first three concepts, and especially variables.

The Scratch automated quality analysis tools Hairball [2] and Dr. Scratch [14] are also related to our work on smell detection. The Hairball Scratch extension is a lint-like static analysis tool for Scratch that can detect initialization problems and unmatched broadcast and receive blocks. In their work [13], Moreno and Robles extended Hairball to detect two bad programming habits in Scratch: not changing the default object names and duplicating scripts, and apply them for evaluating 100 projects from the Scratch repository. The results on script duplication are substantially different than ours—we find projects with script clones to appear half as frequently. The Dr. Scratch tool [14] includes bad naming, code duplication and dead code identification functionality, and also evaluates Scratch projects in terms of abstraction, parallelism, logical thinking, synchronization, flow control, user interactivity and data representation.

7. CONCLUSIONS

In this paper we presented a large-scale study on 247,798 projects we scraped from the Scratch repository. We analyze these projects in terms of size, complexity, application of programming abstractions and utilization of programming concepts including procedures, variables, conditional statements, loops, and broadcast-receive functionality. We find that procedures and conditional loops are not commonly used. We further investigate the presence of code smells, including code duplication, dead code, long method and large class smells. Our findings indicate that Scratch programs suffer from code smells and especially from dead code and code duplication.

In addition to the findings presented in this paper, we provide as contributions the dataset that we used for our study, as well as information on the edge cases that we found in the dataset in terms of size and number of procedures, variables, cyclomatic complexity, clones and dead code.[4]

8. REFERENCES

[1] T. L. Alves, C. Ypma, and J. Visser. Deriving metric thresholds from benchmark data. In *26th IEEE*

International Conference on Software Maintenance (ICSM 2010), pages 1–10. IEEE Computer Society, 2010.

[2] B. Boe, C. Hill, M. Len, G. Dreschler, P. Conrad, and D. Franklin. Hairball: Lint-inspired Static Analysis of Scratch Projects. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 215–220, New York, NY, USA, 2013. ACM.

[3] K. Brennan, C. Balch, and M. Chung. *CREATIVE COMPUTING*. Harvard Graduate School of Education, 2014.

[4] M. Conway, R. Pausch, R. Gossweiler, and T. Burnette. Alice: A Rapid Prototyping System for Building Virtual Environments. In *Conference Companion on Human Factors in Computing Systems*, CHI '94, pages 295–296, New York, NY, USA, 1994. ACM.

[5] S. Cooper, W. Dann, and R. Pausch. Teaching Objects-first in Introductory Computer Science. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '03, pages 191–195, New York, NY, USA, 2003. ACM.

[6] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[7] F. Hermans and E. Aivaloglou. Do code smells hamper novice programming? In *Proceedings of the International Conference on Program Comprehension*, 2016. to appear.

[8] F. Hermans, M. Pinzger, and A. van Deursen. Detecting and refactoring code smells in spreadsheet formulas. *Empirical Software Engineering*, 20(2):549–575, 2015.

[9] J. H. Maloney, K. Peppler, Y. Kafai, M. Resnick, and N. Rusk. Programming by choice: Urban youth learning programming with scratch. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, pages 367–371, New York, NY, USA, 2008. ACM.

[10] T. J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[11] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari. Habits of programming in scratch. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, pages 168–172, New York, NY, USA, 2011. ACM.

[12] O. Meerbaum-Salant, M. Armoni, and M. M. Ben-Ari. Learning Computer Science Concepts with Scratch. In *Proceedings of the Sixth International Workshop on Computing Education Research*, ICER '10, pages 69–76, New York, NY, USA, 2010. ACM.

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

[14] J. Moreno-LeÃşn, G. Robles, and M. RomÃąn-GonzÃąlez. Dr. Scratch: Automatic Analysis of Scratch Projects to Assess and Foster Computational Thinking. *RED : Revista de EducaciÃşn a Distancia*, (46):1–23, Jan. 2015.

[15] B. Moskal, S. Cooper, and D. Lurie. Evaluating the Effectiveness of a New Instructional Approach. In *Proceedings of the SIGCSE technical symposium on Computer science education*, 2005.

[16] T. W. Price and T. Barnes. Comparing Textual and Block Interfaces in a Novice Programming Environment. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, pages 91–99, New York, NY, USA, 2015. ACM.

[17] M. Resnick, J. Maloney, A. Monroy-HernÃąndez, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: Programming for All. *Commun. ACM*, 52(11):60–67, Nov. 2009.

[18] A. Wilson, T. Hainey, and T. Connolly. Evaluation of computer games developed by primary school children to gauge understanding of programming concepts. In *European Conference on Games Based Learning*, page 549. Academic Conferences International Limited, 2012.

[19] D. Wolber, H. Abelson, E. Spertus, and L. Looney. *App Inventor: Create Your Own Android Apps*. O'Reilly Media, Sebastopol, Calif, 1 edition edition, May 2011.