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 those 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 that end we have scraped the Scratch public repository and retrieved 250.000 projects. We present an analysis of those projects in three different dimensions. Firstly, we look at the types of blocks used and the size of the programs. We then investigate complexity, used abstractions and programming concepts. Finally we look at coding style and detect *code smells and clones*. We find that programming abstraction concepts like functions are not commonly used. We further investigate the presence of code smells and bad programming practices, including code duplication, dead code, long method and large class smells. Our findings indicate that Scratch programs suffer from code smells and especially code duplication.

General Terms

terms

Keywords

Scratch, programming practices, code smells, static analysis

1. INTRODUCTION

Scratch 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 12 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 [24] are block-languages aimed at novice programmers.

Several studies have shown that block-based languages are powerful as a tool for teaching programming [15, 19, 5]. **Fenia, do you have some work on Research on programming habits**

Recent work has focused on *code smells* within Scratch programs, parts of a program that are not designed in the best possible way. **add dr. scratch papers here** Ar recent controlled experiment found that long scripts and duplication decreases a novice programmer's ability to understand and modify Scratch programs **cite**.

The goal of this paper is to obtain a deep understanding of how people program in Scratch and to analyze the characteristics of Scratch programs. Moreover, knowing that bad programming habits and code smells can be harmful, we also want to explore whether they are common. Specifically, the research questions that we are trying to answer are:

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 and bad programming practices in Scratch programs?

Our study is based on data from the Scratch program repository. By scraping the list of recent programs [2], we have obtained 250,166 public Scratch programs and performed analysis on them.

The contributions of this paper are as follows:

- A public data set of 233,491 Scratch programs (Section 3.1)
- An evaluation of the data set in terms of (Section 4)
- A discussion of implications of our findings for educational programming language designers (Section 5)

2. BACKGROUND AND MOTIVATION

Recently new block-based languages have gained widespread popularity, especially powered by Scratch [22] and Blockly[3].

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

Over 100 million students have tried Blockly via Code.org, and the Scratch repository currently hosts over 12 million projects.

Since their introduction, studies have demonstrated the applicability of block-based languages as a tool for education. Scratch, for example, was evaluated with a two-hour introductory programming curriculum for 46 subjects aged 14 [15]. This study indicated that Scratch could be used to teach computer science concepts: analysis of the pre- and post-tests showed a significant improvement after the Scratch course, although some concepts like variables and concurrency remained hard for students.

Moskal *et al.* [19] compared computer science students who studied Alice before or during their first programming course to students that only took the introductory computer science course. Their results show that exposure to Alice significantly improved students' grades in the course, and their retention in computer science in general over a two year period. A follow-up study by Cooper *et al.* [5] obtained similar results, showing that a curriculum in Alice resulted in improved grades and higher retention in computer science.

Most convincingly, Price and Barnes performed a controlled experiment in which students were randomly assigned to either a text-based or a block-based interface in which they had to perform small programming tasks [21]. Their experiment showed that students in the block-based interface were more focused and completed more of the activity's goals in less time.

2.1 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 application and as a web application. Figure 1 shows the Scratch user interface in the Chrome browser.

2.1.1 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 consists two one sprites, the cat, which is Scratch's default sprite and logo 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.

2.1.2 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.

2.1.3 Events

Scratch is *event-driven*: all motions, sounds and changes

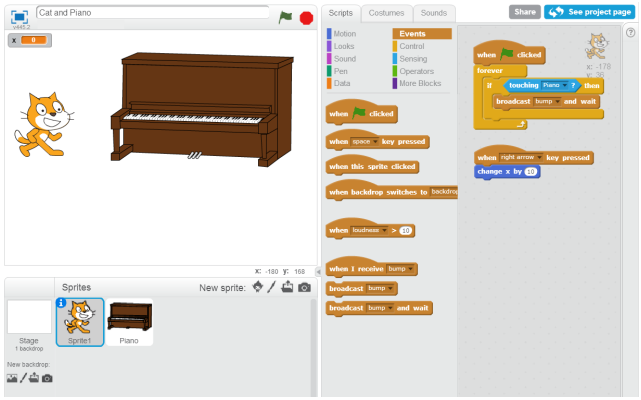

Figure 1: The Scratch user interface consisting of the 'cat' sprite on the left, the toolbox with available blocks in the category 'Events' in the middle and the code associated with the sprite on the right. The upper right corner shows the actual location of the sprite.

Figure 2: The script for the Piano sprite in Figure 1, causing a sound on receiving the 'bump' signal.

in the looks of sprites are initiated by events called Hat blocks[4]). 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.

2.1.4 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. The Piano reacts on that by making a sound, see Figure 2.

2.1.5 Procedures

Scratch users can define their own blocks, with a pop-up screen displayed in on the left hand side of Figure 3. As shown in this figure, Scratch users can enter the name of the sprite and define input parameters. The block in Figure 3 uses two parameter, one textual (string1) and one numerical (number1). A user-created block is programmed using ordinary Scratch blocks as shown on the right hand side of Figure 3.

3. RESEARCH DESIGN AND DATASET

The main focus of this study is to understand how people program in Scratch by analyzing the characteristics of

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

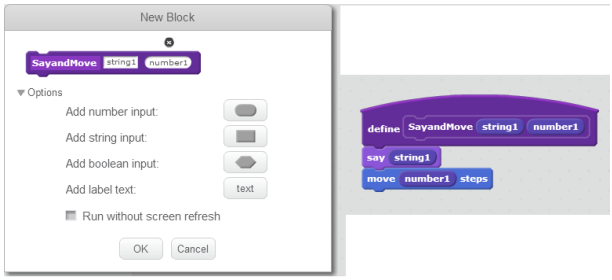

Figure 3: The user interface for creating (left) and defining (right) user-defined blocks.

Scratch programs. To answer our three research questions, we conducted an empirical quantitative evaluation of program data we collected from the Scratch repository. The dataset is described in Section 3.1. **Below, we present how we approached each research question.**

RQ1 What are the size and complexity characteristics of Scratch programs? quantitative analysis, provide and analyze descriptive statistics on the size, complexity metrics, statistical analysis . mentioned taht it was imported to db and done and statistical data from db queries? The outcomes are presented in Section...

RQ2 Which coding abstractions and programming concepts and features are commonly used when programming in the Scratch environment? Using the code in the dataset, we perform a detailed statistical analysis of code characteristics to answer RQ2 (Section ??). We manually assigned types to commands/blocks.

RQ3 Which code smells and bad programming practices are common in Scratch programs? to examine if coding style conventions are used, we qualitatively analyze..

3.1 Dataset

3.1.1 *Obtaining Scratch programs*

We obtained the set of Scratch programs by using a scraping program. Our scraping program called Kragle starts by reading `https://scratch.mit.edu/explore/projects/all/` and thus obtaining program ids of programs that were most recently shared. Subsequently, Kragle obtains the JSON code for each of the listed programs.

We ran Kragle on March 2nd 2016 for 24 hours, when it had obtained a little over 250.000 programs. Out of the 250,166 , we failed to parse and further analyze 2.367 programs due technical difficulties with the provided JSON files. Kragle, as well as all scraped programs and our analysis files are available[5].

3.1.2 *Analyzing the Scratch programs*

Once we obtained the Scratch programs, we parsed the JSON files according to the specification of the format[6]. This resulted in a list of used blocks per program, with the sprites and the stage of the program. We cross referenced also all blocks with the Scratch wiki so we have the shapes and the category of all blocks. For example, When Green Flag Clicked is a *Hat block* from the *Events category*.

[5]`https://github.com/ScratchLover42/ICER-Data-Code`
[6]`http://wiki.scratch.mit.edu/wiki/Scratch_File_Format_(2.0)`

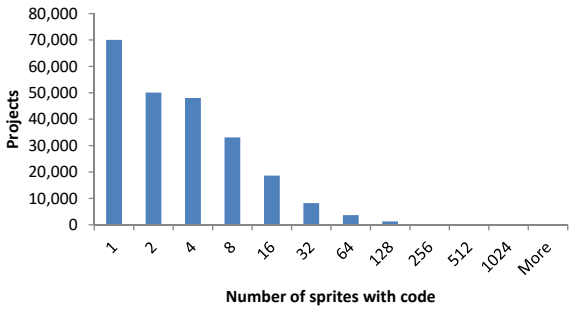

Figure 4: Histogram of the number of sprites in the analyzed projects

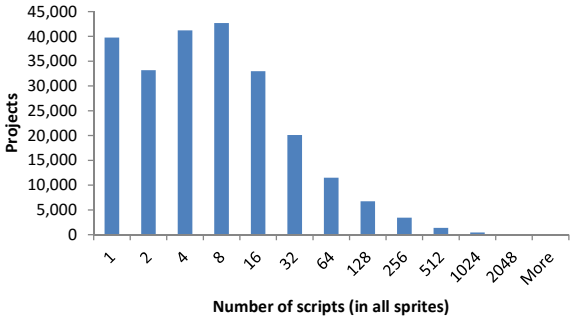

Figure 5: Histogram of the number of scripts in the analyzed projects

4. RESULTS

In the following sections we describe the results obtained for each of the research questions 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 those projects contains only sprites and costumes, but no code, while in other programs they were entirety empty apart from the Scratch cat added by default. Since those 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 statistics for metrics related to size and complexity. 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 per project. We furthermore measure the lines of code per project, considering each Scratch block to represent exactly one line of code, thus not counting the closing constructs of loops or conditional statements as separate lines. Figures 4, 5, 6 and 7 plot the distribution of those 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 lines of code, while one fourth of the projects have up to 12 lines of code. On the other end, 5% of the projects (11,712)

	mean	min	Q1	median	Q3	max
Sprites with code per project	5.68	1	1	2	5	525
Scripts per project	17.35	1	2	5	12	3038
Maximum indentation per project	4.56	1	3	3	5	488
Lines of code (LOC) per project	154.55	1	12	29	76	34622
LOC in Stage per project	4.80	0	0	0	3	2613
LOC in Sprites per project	115.57	0	10	26	68	34613
LOC in Procedures per project	34.17	0	0	0	0	20552
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 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
Variables per project	2.06	0	0	0	1	340
Scripts utilizing variable	4.97	1	1	3	5	1127
Lists per project	0.55	0	0	0	0	319
Conditional statements per project	10.02	0	0	0	3	5950
Recursive loop statements per project	7.65	0	1	2	5	2503
User input blocks per project	4.77	0	0	1	4	1889
Broadcast-receive statements per project	8.57	0	0	0	2	2460

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

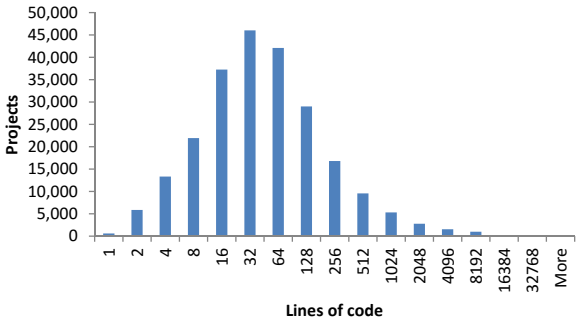

Figure 6: Histogram of the lines of code in the analyzed projects

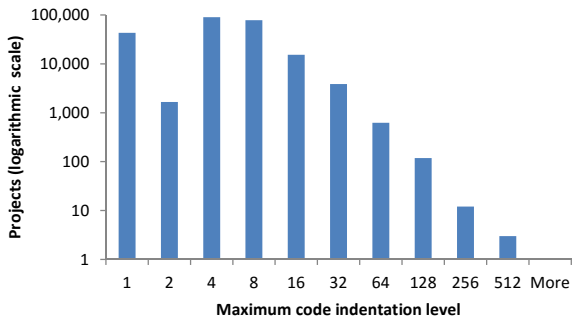

Figure 7: Histogram of the maximum code indentation level in the analyzed projects

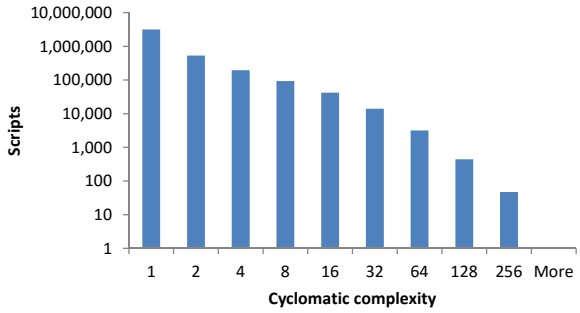

Figure 8: Histogram of the McCabe cyclomatic complexity of the 4,049,356 analyzed scripts

have more than 18 sprites and 4.8% (11,214) consist of more than 500 lines of code. The analysis also highlighted some surprisingly large projects: 135 with more than 300 sprites and even 30 projects with more than 20,000 lines of code[7].

The lines of code metric was further analyzed to understand code organization. As shown in Table 1, the majority of Scratch code—74.78% out of 36,085,654 lines of code—is written within sprites. An additional 3.1% of the total lines are found in the stage class. More interestingly, the remaining 22.11% are lines 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 lines of code (48.81%) are lines within procedures.

To understand the complexity of the Scratch programs in our dataset, we use the McCabe cyclomatic complexity [14], a quantitative measure of the number of linearly in-

[7]The Scratch identifiers of those projects can be found in **link**

	Number of projects	Percentage
Retrieved	250,166	
Analyzed	247,798	
Non-empty (used for statistics)	**233,491**	
Procedures	17,979	7.70%
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>` statements	31,739	13.59%
`broadcast - receive` statements	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 functionality 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	56,890	24.36%
Large scripts	69,521	29.77%
Large sprites	31,954	13.68%

Table 2: Characteristics in the projects in the dataset

dependent 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. The results of the cyclomatic complexity metric per script are plotted in Figure 8. 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[8]. 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 lines of code and no decision points. Most code is written in sprites. A small number of projects (7.7%) use procedures, but they use them a lot and for more complex code. There exist surprisingly large and complex projects.

4.2 Programming Abstractions and Concepts

The first method for abstraction that we investigate is the use of 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 those projects having up to 2. Figure 9 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 10). 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 investi-

[8]The Scratch identifiers of those projects can be found in **link**

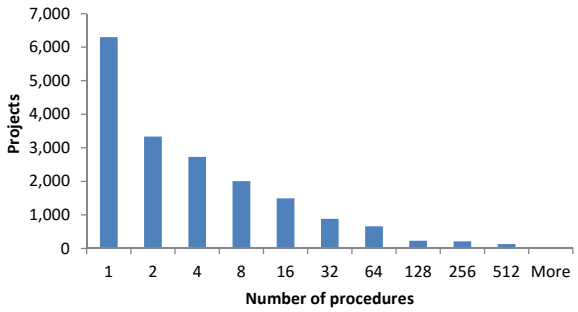

Figure 9: Histogram of the number of procedures for the 17,979 projects that include at least one

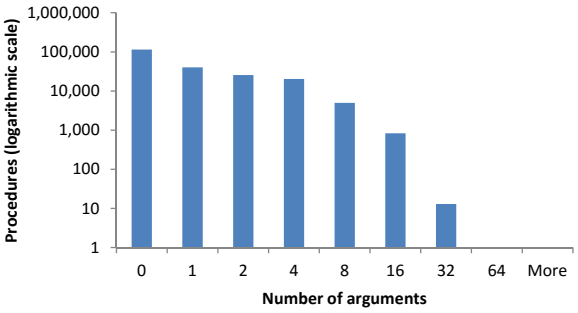

Figure 10: Histogram of the number of arguments for the procedures in the dataset

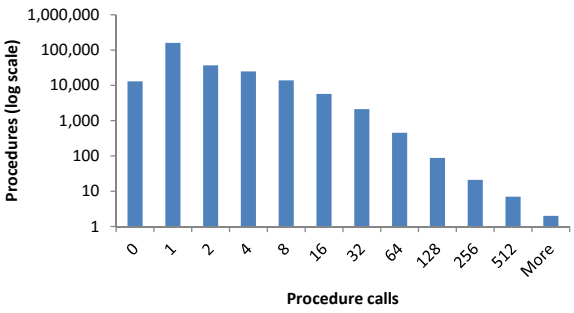

Figure 11: Histogram of the number of calls of each procedure in the dataset

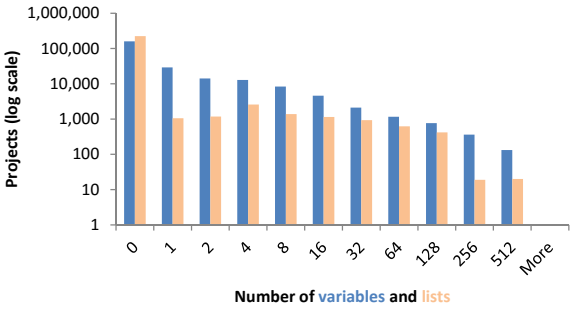

Figure 12: Histogram of the number of variables and lists used in the projects

gated through the procedure calls, summarized in Figure 11. A significant number of the defined procedures (13,036 or 5.06%) are not called in the projects. 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 most of the calls (56.09%) originate from other procedures, and even 1.06% originate from the same procedure, making them recursive calls. Those recursive procedures are found in 1,052 projects[9].

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 12. Exceptional cases exist: the analysis highlighted 842 projects with more than 100 variables and up to 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, it is considered bad style **citation needed**.

Regarding program control features, conditional statements (blocks `if <condition> then` and `if <condition> then else`) are used by 39.81% of the projects. Recursive 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 recursive 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.

[9]list of recursive procedures published **link**

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 233,491projects of the dataset

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 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`. Those blocks are used by 29.57% of the projects. `broadcast and wait` is rarely used, in only 3.87% of the projects. The events 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 syncronizations occurs in 18,669 projects (7.99% of the total non-empty ones).

RQ2: One third of the projects use variables, sometimes without initializing them. 39.81% of the projects contain conditional statements and 77.18% recursive loops, but conditional recursion is rarely used. More than half of the projects are interactive. 29.57% of the projects use broadcast and receive blocks, but in 7.99% of the projects they are not synchronized. 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 1052 (0.45% of the total) projects.

4.3 Code smells and bad programming practices

The duplicated code smell is the first smell that we examine. The first step for our analysis is 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 program. For the identification of clones we do 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

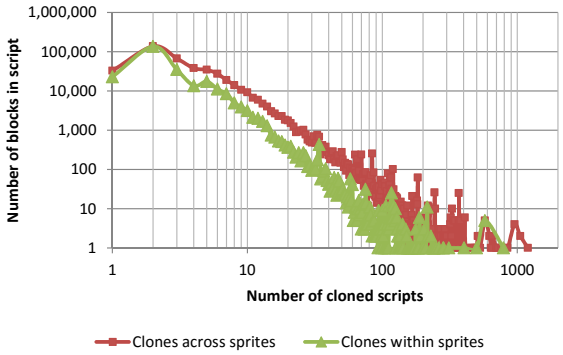

Figure 13: Number of cloned scripts of different block sizes across and within sprites

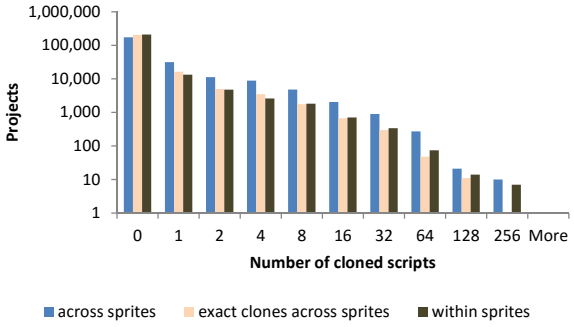

Figure 14: Histogram of the number of cloned scripts

parameter values, and we refer to them as *exact clones*. The next step in the analysis is to determine the minimum size of the scripts that are considered clones instead of incidentally similar. For this, we examine the number of detected clones for different script sizes and present the results in Figure 13. Based on this distribution, we opt to adopt the number also used by the authors in [17], which is the minimum size of 5 blocks per script.

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 those 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 [10]. Manual inspection of the mostly copied clones revealed that they were scripts used multiple times to cater for different parameter values, sometimes in multiple sprites.

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 the projects. These are duplication smells that could be avoided by using procedures, which in Scratch can only be defined for specific sprites.

[10] **link**

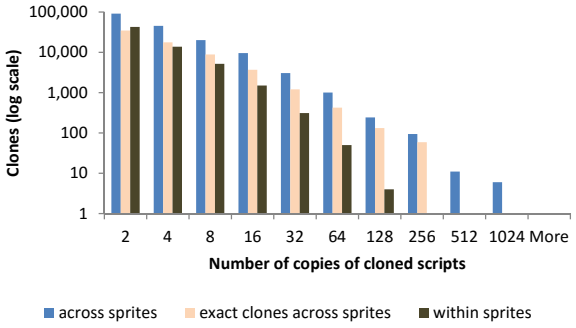

Figure 15: Histogram of the number of copies of the identified clones

The lack of support for project-wide procedures could be the cause for procedure clones. These 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 with only the first block being different. In first block is commonly the event block and, by separating this one from the subsequent functionality blocks, we could examine if 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. Already in the previous section, Scratch projects were found to have two types of dead code: defined procedures that are not called (5.06% of the defined procedures in 2,079 projects) and unmatched broadcast-receive messages (in 7.99% of the projects). In this section we examine an additional case: scripts that are never invoked due to the lack of a starting `when <trigger>` block from the Scratch Events or Control category, like in the example in Figure 17, or scripts that 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 those scripts (86.6%) are scripts missing the starting block. Examining the size of those dead scripts, 72.34% are composed of a single block. As shown in diagram 16, however, considerably large dead scripts exist; 2,358 of those scripts originating from 1,553 different projects have more than 30 blocks and up to 2,610[11].

Finally, we examine the long method and the large class smell, considering them in the context of Scratch as large script and large sprite smells respectively. For those two smells we use the number of blocks as the size metric. Figure 18 presents the number of blocks in the scripts and the sprites of our dataset. We used those numbers to split the dataset and retrieve the top 10% **felienne, can we justify why 10 percent?** largest scripts and sprites, and in this way we found the thresholds to be used for the calculation of the large script and large sprite: it is 18 blocks and 59 blocks respectively. Using these thresholds, the number of projects exhibiting the large script smell, i.e., containing at

[11] **link**

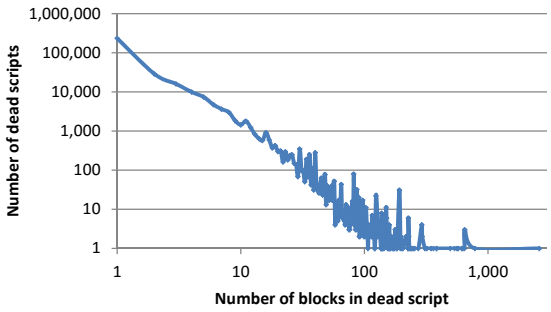

Figure 16: Size of scripts that are identified as dead code

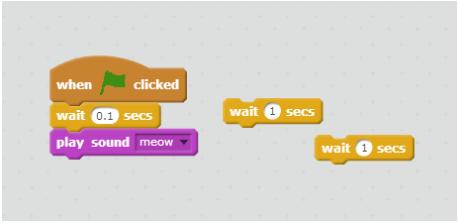

Figure 17: Cloned blocks that are not connected to an event. Program id: 12237615

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 25.54% of the projects, with 46.55% of the clones copied three or more times, in the same or across sprites. 24.36% of the projects contain scripts that are never invoked, and thus exhibit the dead code smell. In some cases those scripts are large. The large script smell is found in 29.77% of the projects and the large sprite in 13.68%.

5. DISCUSSION

5.1 Implications

Our results have some interesting implications for designers of educational programming languages.

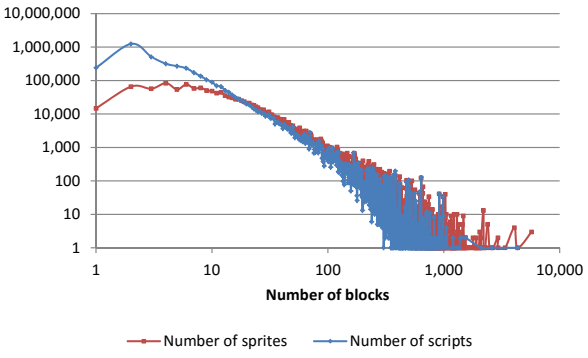

Figure 18: Size of sprites and scripts in number of blocks

5.1.1 *Dead Code*

In our analysis, we find that almost one quarter of the Scratch programs 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, like in the example in Figure 17, or scripts that are comprised of only a `when <trigger>` block without any functionality. **Fenia, this is correct, right, the 23% is just these types and not also uncalled procs and unmatched broadcasts?**.

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 with other visual educational languages, Scratch does on indicate scripts are dead. LEGO Mindstorms, for example, does give the user 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 to keep their code clean, the programming interface could actively encourage users to move unconnected blocks to that workspace when they exit the environment.

5.1.2 *Exact Clones between Sprites*

With occurrences in 11% of the Scratch programs 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 other way to share the functionality 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.

5.1.3 *Sharing of Scripts and Procedures*

Investigating the use of clones between programs, we observe that there are 1700 scripts that are used in multiple programs, sometimes as often as in 1600 different programs. This seems to indicate that there are common patterns in Scratch programs. This again leads us to the idea that it might be very beneficial to Scratch programmers if they could not only share their programs, 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 quicker.

5.2 Threads to validity

A thread to validity of this study is the fact that we did not scrap a random sample, but the most recent 250,000 programs. It could be the case that programming habits of Scratch users are changing over time. However, we counterbalanced that by using a very large dataset. Furthermore, for some of the analyses we manually inspected programs **Fenia, which ones?**

Furthermore we use the number of blocks in the Scratch programs 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.

A study on the internalization of programming concepts with Scratch with 46 students was presented in [15]. 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 [**?**] 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 [**?**] 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.* [**?**], 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 [18] 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 [17], 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 [18] 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.

Also related to our research are works on code smells, initiated by the work by Fowler [6]. His book gives an overview of code smells and corresponding refactorings. Fowler's work was followed by efforts focused on the automatic identification of code smells by means of metrics. Marinescu [13] for instance, uses metrics to identify *suspect* classes: classes which could have design flaws. Alves *et al.* [1] focus on a strategy to obtain thresholds for metrics from a benchmark Moha *et al.* [16] designed the 'DECOR' method which automatically generates a smell detection algorithms from specifications. The CCFinder tool [11] finally, aims at detecting clones in source code, which are similar to our code duplication smell.

7. 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] E. Glinert. âĂIJTowards "Second Generation" Interactive, Graphical Programming Environments,âĂİ. In *Proceedings of the IEEE Workshop on Visual Languages*, 1986.

[8] F. Hermans, M. Pinzger, and A. v. Deursen. Detecting and Visualizing Inter-Worksheet Smells. In *Proceeding of the 34rd international conference on Software engineering (ICSE 2012)*, pages 451–460. ACM Press, 2012. to appear.

[9] F. Hermans, M. Pinzger, and A. van Deursen. Detecting and refactoring code smells in spreadsheet formulas. *Empirical Software Engineering*, 20(2):549–575, 2014.

[10] B. Jansen and F. Hermans. CODE SMELLS IN SPREADSHEET FORMULAS REVISITED ON AN INDUSTRIAL DATASET. In *Proceedings of the International Conference on Software Maintenance and Evolution*, pages 372–380, Bremen, Germany, 2015.

[11] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *TSE*, 28(7), July 2002.

[12] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. 2006.

[13] R. Marinescu. Detecting Design Flaws via Metrics in Object-Oriented Systems. In *Proceedings of TOOLS*,

pages 173–182. IEEE Computer Society, 2001.

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

[15] 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.

[16] N. Moha, Y. GuÃl'hÃl'neuc, L. Duchien, and A. Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, Jan. 2010.

[17] 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.

[18] 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.

[19] 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.

[20] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *Proceedings of International Symposium on Empirical Software Engineering and Measurement*, pages 390–400, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[21] 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.

[22] 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.

[23] K. T. Stolee and S. Elbaum. Refactoring Pipe-like Mashups for End-user Programmers. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 81–90, New York, NY, USA, 2011. ACM.

[24] 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.