How kids code and how we know it: An exploratory study in the Scratch project 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. 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 terms of complexity, used abstractions and programming concepts, and coding style. We found 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 are visual languages, but also use some successful aspects of text-based languages such as limited text-entry and indentation, and as such are closer to 'real', textual programming than other forms of visual programming, like dataflow languages are.

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.

refer to: Research on the effectiveness of those languages for programming education Research on programming habits Our research on smells

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 **link**, we have obtained 250,166 public Scratch programs and performed static analysis on them.

The contributions of this paper are as follows:

- A public data set of 233,491 Scratch programs (Section 3.1)
- An empirical evaluation of the data set for answering the aforementioned research questions

2. BACKGROUND AND MOTIVATION

Block-based languages go back to 1986, when Glinert introduced the BLOX language [7]. BLOX consists of puzzle-like programming statements that can be combined into programs by combining them both vertically and horizontally. After a decade of little activity into block-based languages, they became a research topic again, starting with Alice [4]. More recently, new block-based languages have gained widespread popularity, especially powered by Scratch [22] and Blockly[2]. Over 100 million students have tried Blockly via Code.org, and the Scratch repository currently hosts over 12 million projects. Unlike in BLOX, in these new

[1] `https://developers.google.com/blockly/`
[2] `https://developers.google.com/blockly/`

block-based languages the programming blocks can only be combined vertically, resembling textual code more.

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.

Summarizing the above, we conclude that block-based languages have a clear potential to be a great tool for introductory programming education, in some cases even outperforming text-based languages.

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[3] consists of one sprite, the cat, which is Scratch's default sprite and logo. The code in the sprite will cause the cat to jump up, say "hello", and come back down, when the green flag is clicked, and to make the 'meow' sound when the space bar is pressed.

2.1.2 Signals

uitleggen wat een signal is en evt ook al hoe het een kloon vervangt?

2.1.3 Events

[3] https://scratch.mit.edu/projects/97086781/

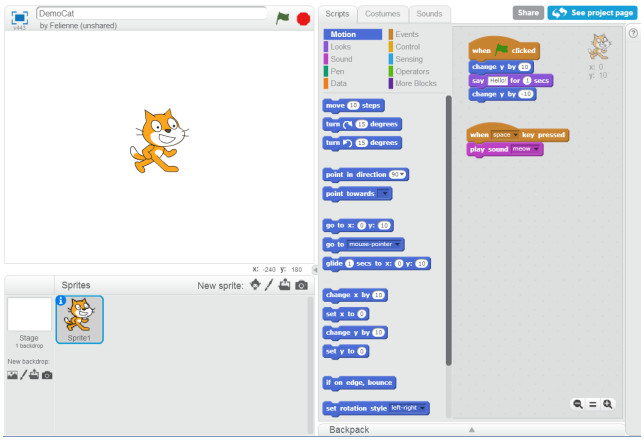

Figure 1: The Scratch user interface consisting of the 'cat' sprite on the left, the toolbox with available blocks in the category 'motion' in the middle and the code associated with the sprite on the right. The upper right corner shows the actual location of the sprite.

Scratch is *event-driven*: all motions, sounds and changes in the looks of sprites are initiated by events. 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. In the Scratch code in Figure 1 there are two events: 'when Green Flag clicked' and 'when space key pressed'.

2.1.4 Scripts

Source code within sprites is organized in scripts: a script always starts with an event, followed by a number of blocks. 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. For example, the code on the left of Figure **??** has five scripts associated with the 'when Green Flag clicked' event.

2.1.5 Remixing

Scratch programs can be shared by their creators in the global Scratch repository[4]. Shared Scratch programs can be 'remixed' by other Scratch users, which means that a copy of this program is placed in the user's own project collection, and can be then further changed. The 'remix tree' of projects is public, so users can track which users remix their programs, a bit similar forking in GitHub. Contrary to forking though, changes upstream cannot be integrated back into the original project.

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 programs. To answer our research questions, we

[4] https://scratch.mit.edu/explore/projects/all/

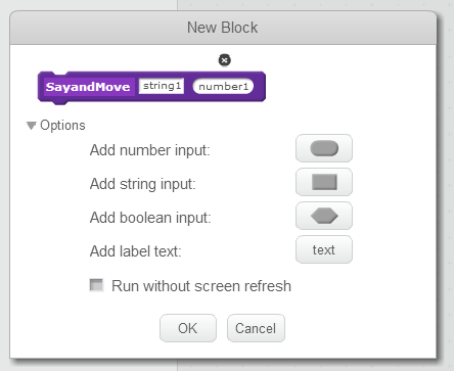

Figure 2:

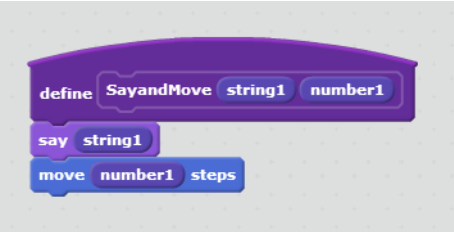

Figure 3:

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. In addition to the programs themselves, Kragle also gathers metadata including the numbers of views, loves, favorites and remixes.

We ran Kragle on March 2nd 2016 and let it running for 24 hours, when it had obtained a little over 250.000 programs.

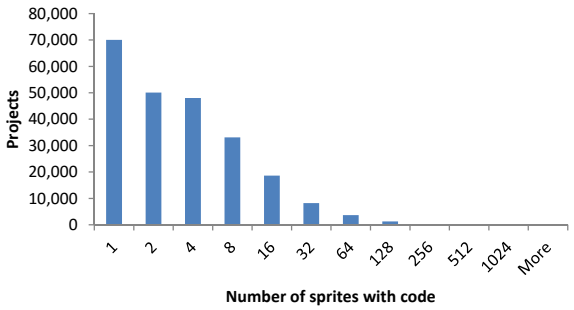

Figure 4: Histogram of the number of sprites in the analyzed projects

Out of the 250,166 , we failed to analyze 2.367 programs due to various technical difficulties **add details, I will check a few to see what went wrong - Felienne**

Kragle, as well as all scraped programs and our analysis files are available via **url I do not think we need to describe the fields in the paper, but we need a good readme with the data (I can do that)**

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[5]. This resulted in a list if used blocks per program, with the sprites and the stage of the program. We also assign types to all the blocks, both their shapes and the tab they originate from. For example, When Green Flag Clicked is a *Hat block* from the *Events category.*

4. RESULTS

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

4.1 Program size and complexity

The dataset is found to contain a relatively small number of projects without any code: 14,307(5.77%). Through random sampling we found that in some cases those projects contained sprites and costumes but no code and in others 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, where each script is a code-block of interconnected Scratch blocks. For example, in Figure 1 the cat sprite contains 2 scripts. We further 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. The maximum code indentation level is used as a code complexity metric. It was calculated using the JSON array depth? **felienne, how did we**

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

	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	4.80	0	0	0	3	2613
LOC in Sprites	115.57	0	10	26	68	34613
LOC in Procedures	34.17	0	0	0	0	20552
McCabe Cyclomatic Complexity per script	1.58	0	0	0	0	246

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

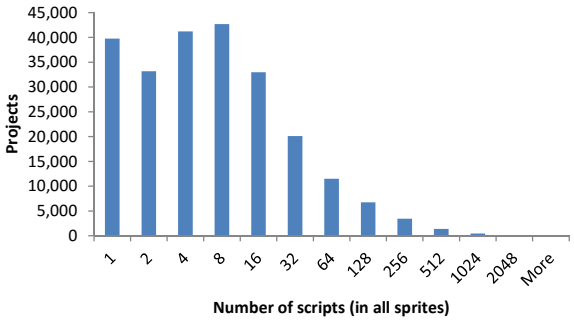

Figure 5: Histogram of the number of scripts in the analyzed projects

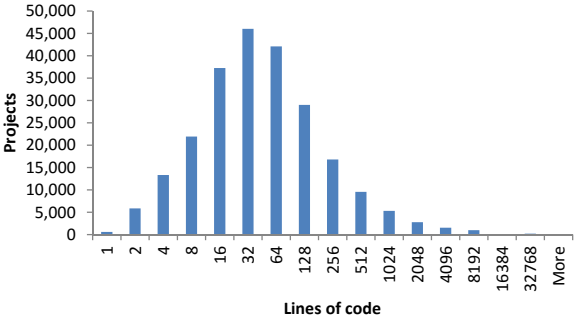

Figure 6: Histogram of the lines of code in the analyzed projects

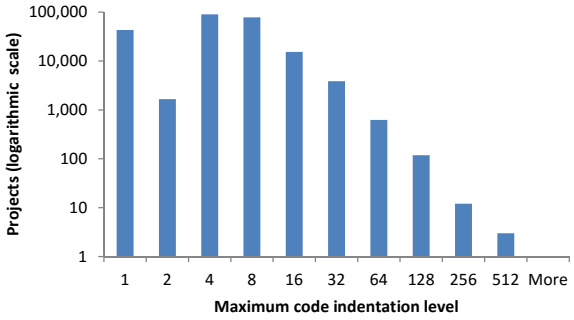

Figure 7: Histogram of the maximum code indentation level in the analyzed projects

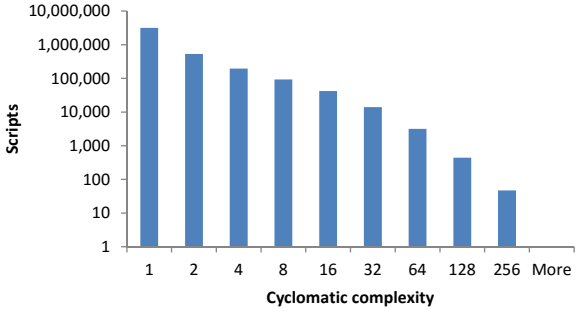

Figure 8: Histogram of the McCabe cyclomatic complexity of the 4,049,356analyzed scripts

measure this?. This corresponds to code indentation level returning 1 for all first-level blocks and increasing by 2 for block enclosures in loops or conditional statements. 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) are found to 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[6].

The lines of code metric was further analyzed to understand code placement. As shown in Table 1, the majority of Scratch code—74.78% out of 36,085,654lines 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.

As a complexity metric we also used the McCabe cyclomatic complexity [14], a quantitative measure of the number of linearly 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. The results of the cyclomatic complexity metric per script are plotted in Figure 8. The majority (78.33%) of 4,049,356scripts 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

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

Figure 9: Cloned blocks that are not connected to an event. Program id: 12237615

highlighted 209 scripts with cyclomatic complexity over 100 and up to 246.

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. Few projects (7.7%) use procedures, but they use them a lot. There exist surprisingly large and complex projects.

4.2 Programming abstractions and concepts

loops logic operations variables lists user interativity functions - arguments parallelism - wait blocks, briadcast, receive

4.3 Code smells

dead code duplicates/clones -graph, number of overlapping positioning, number of dead code duplicates long method large class conditional complexity Dublicated wait block-same trigger caught by more than 1

coding style -naming -functions, sprites, variables code block positioning - overlapping /visibility

4.3.1 Clones Detection for Scratch programs

We distinguish three different forms of clones: Exact clones within one sprite, exact clones between sprites of one program, clones of 'wait' blocks. This section presents the results of our clone detection algorithm on the 233,491 with scripts.

In total 67,177 (26.9% of programs with scripts) exhibit one of the three types of cloning.The most common type of cloning is copying entire scripts between sprites. This occurs in 62.939 programs (25.2%). In 8.220 programs (3.3%) wait conditions are cloned and in 2.463 files there is a script cloned within a sprite.

Exact clones within one sprite.

In total, we found **x** cases of exact script clones within one sprite. Upon further inspection, we found that some of them consisted of unconnected blocks, as depicted in Figure 9. While this is technically a clone, as this is *dead code* and will never be executed, this could be considered unharmful sketching. Looking only at clones starting with an event block, **...** clones remain over **?** files.

Exact clones between sprites of one program.

Clones of 'wait' blocks.

Previous work has established that the cloned 'wait' blocks can be harmful when modifying Scratch code, which is the reason why we investigated this particular type of cloning. As outlined in section **?** signals can replaced repeated wait blocks, by catching the condition in one place and broadcasting a signal to all other sprites. This is why we found it interesting to see in how many of the cases where a wait block was cloned, the creator of the program also used signals in their program. This indicates that the concept of signals is known, and hence the duplicate condition is not used because the user is not aware of the alternative. In **...** of the cases the program with the duplicated wait block also contained one or more signals.

5. DISCUSSION

5.1 Implications

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

5.1.1 Unconnected Blocks

In the whole dataset, not just in the programs with clones, we observe unconnected scripts, scripts without an event block and events without associated code. This is a form of *dead code*, code that will never be executed. Because this dead code caused visual clutter, it would be better to have a separate workspace (much like the 'backpack in Scratch' where users can store blocks temporarily and use them later. A programming interface could ask actively encourage users to move unconnected block when they exit the environment to encourage a 'clean' working space.

5.1.2 Code Clones Between Sprites

With occurrences in one quarter of the 250 thousand programs, the use of exactly identical clones between sprites is extremely common. In a sense, the Scratch users are not to blame here, Scratch does not support the sharing of **right word? procedures** between scripts, only within them. So in many cases there is no other way to share the functionality than 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 needs this functionality.

5.1.3 Refactoring support for duplicate wait blocks

In todo... of the cases, the programs with duplicate conditions, also signals were present. This means that it is feas

6. RELATED WORK

6.1 Code Smells

Efforts related to our research include 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. Lanza and Marinescu [12] explain this methodology in more detail. Alves *et al.* [1] focus on a strategy to obtain thresholds for metrics from a benchmark. Olbrich *et al.* furthermore investigates the changes in smells over time, and discusses their impact [20]. 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 *Duplication* smell.

6.2 Smells beyond the OO paradigm

In recent times, code smells have been applied to programs outside of the regular programming domain. In our past work, we have, for example, studied code smells within spreadsheets, both at the formula level [9] and between worksheets [8].

More recently, we compared two datasets: one containing spreadsheets which users found unmaintainable, and a version of the same spreadsheets rebuilt by professional spreadsheet developers. The results show that the improved versions suffered from smells to a lesser degree, increasing our confidence that presence of smells indeed coincides with users finding spreadsheets hard to maintain [10].

In addition to spreadsheets, code smells have also been studied in the context of Yahoo! Pipes a web mashup tool. An experiment demonstrated that users preferred the non-smelly versions of Yahoo Pipes programs [23].

6.3 Quality of Scratch programs

Finally, there are other works on the quality of Scratch programs. There is for example the Hairball Scratch extension [2], which is a lint-like static analysis tool for Scratch that can detect, for example, unmatched broadcast and receive blocks, infinite loops and duplication. An evaluation of 100 Scratch programs showed that Scratch programs indeed suffer from duplication and bad naming [17].

Most related to our study is the work by Moreno *et al.* [18] who gave automated feedback on Scratch programs to 100 children aged 10 to 14. Their results demonstrated that feedback on code quality helped improve students' programming skills.

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. GuÃľhÃľneuc, 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.