Clones in Block-Based Languages: A Large Scale Analysis of Scratch Programs

Author1
Institute 1

Author2
Institute 2

Abstract

Recently, block-based programming languages like Alice, Scratch and Blockly have become popular tools for programming education. There is substantial research showing that block-based languages are suitable for early programming education. But do block-based programs suffer from code clones smelly *too? In a recent controlled experiment, we found that code clones in Scratch indeed hamper novice Scratch programmers. In this paper we explore how commonly code clones occur in Scratch. To that end we have scraped Scratch public repository and retrieved 250.000 programs. We have analyzed three types of clones within those programs': clones of entire scripts within sprites, clones of entire between sprites, and cloned conditions. We find that those clones occur in ... of programs.*

terms

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

ACM proceedings, LaTeX, text tagging

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 [1], Blockly[1] and App Inventor [2] are block-languages aimed at novice programmers.

In this paper we explore code clones smells in the context of block-based languages. **hier nog wat background over clones, zie ICSE '13 uiteraard**

Clones so far have mostly been studied in the contact of object-oriented, textual source code, but some other directions have been taken, including spreadsheets and SimuLink programs.

We ourselves have recently performed an experiment where we compared the performance of novice Scratch programmers between three

Knowing that code clones can be harmful, we now turn to the question of whether they are common. As such, the goal of this paper is to **investigate whether code clones are common in public Scratch programs**. To address this goal we have obtained 250.166 public Scratch programs by scraping the list of recent programs **link**. We investigate three types of clones: clones of entire scripts within sprites, clones of entire between sprites, and cloned conditions.

The results show that **todo**

The contributions of this paper are as follows:

- A public data set of 233.514 Scratch programs (Section 4)
- A clone Scratch detection algorithm with an open source implementation (Section 5)
- An empirical evaluation of the occurrence of three types of clones in Scratch programs (Section 6)

2. Background and Motivation

Block-based languages go back to 1986, when Glinert introduced the BLOX language [3]. BLOX consists of puzzle-like programming statements that can be combined into programs by

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

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 [1]. More recently, new block-based languages have gained widespread popularity, especially powered by Scratch [4] 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 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 [5]. 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.* [6] 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.* [7] 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 [8]. 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.

3. Relevant Scratch Concepts

This paper is by no means an introduction into Scratch programming, we refer the reader to [9] 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.

3.1 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]https://developers.google.com/blockly/
[3]https://scratch.mit.edu/projects/97086781/

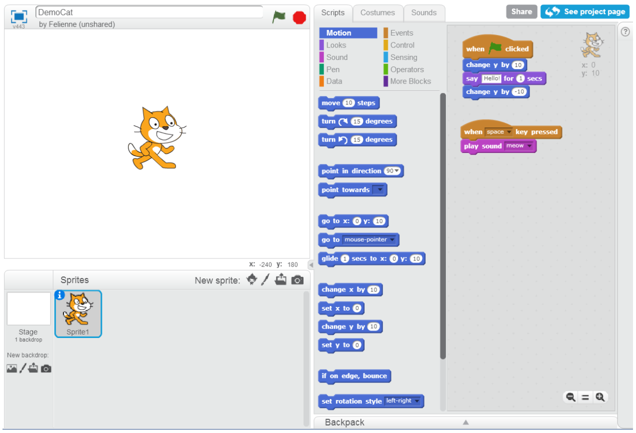

Figure 1. The Scratch user interface consisting of the 'cat' sprite on the left, the toolbox with available blocks in the category 'motion' in the middle and the code associated with the sprite on the right. The upper right corner shows the actual location of the sprite.

3.2 Signals

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

3.3 Events

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'.

3.4 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.

3.5 Remixing

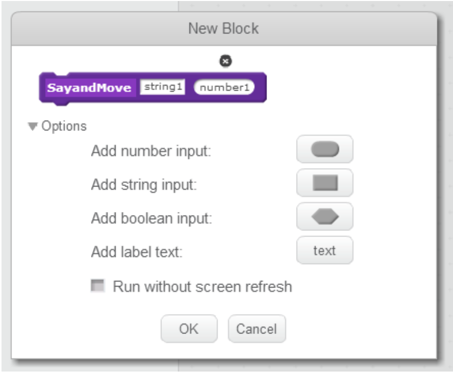

Figure 2.

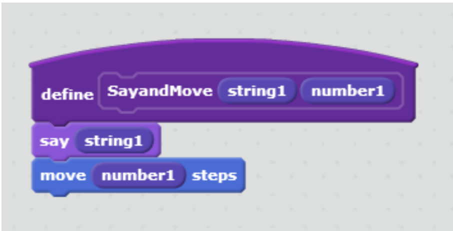

Figure 3.

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.

4. Dataset

We scraped all Scratch programs uploaded **add date range?** from the Scratch website with a scraping program. This resulted in he JSON code for 250.166 Scratch programs. In addition to the programs themselves, we also gathered metadata including the numbers of views, loves, favorites and remixes. Out of the 250.166 , we failed to analyze 2.367 programs due to various technical difficulties **add details?**

4.1 Empty programs

Also, interestingly enough, 14.285 were empty (5.7%), at least in terms of scripts. So Scratch users were sharing program already that did not contain code yet. In some cases **analyze too?** they contained scripts and costumes but no code and in others they were entirety empty apart from the Scratch cat added by default.

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

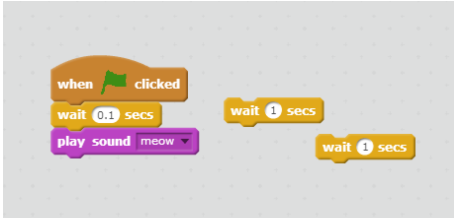

Figure 4. Cloned blocks that are not connected to an event. Program id: 12237615

4.2 Block Usage

4.3 Block Size

4.4 Dead code

4.5 User-defined Blocks

4.6 Remixing

5. Clones Detection for Scratch programs

6. Results

The overarching research question of this work is how common Scratch users clone in their programs. As explained in Section 5 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 233.514 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.

6.1 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 4. 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.

6.2 Exact clones between sprites of one program

6.3 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.

6.4 Conclusion

6.5 Implications

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

6.5.1 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.

6.5.2 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.

6.5.3 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

7. Discussion

8. Related Work

8.1 Code Smells

Efforts related to our research include works on code smells, initiated by the work by Fowler [10]. 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 [11] 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.* [13] 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 [14]. Moha *et al.* [15] designed the 'DECOR' method which automatically generates a smell detection algorithms from specifications. The CCFinder tool [16] finally, aims at detecting clones in source code, which are similar to our *Duplication* smell.

8.2 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 [17] and between worksheets [18].

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 [19].

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 [20].

8.3 Quality of Scratch programs

Finally, there are other works on the quality of Scratch programs. There is for example the Hairball Scratch extension [21], 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 [22].

Most related to our study is the work by Moreno *et al.* [23] 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.

9. References

[1] 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*, ser. CHI '94. New York, NY, USA: ACM, 1994, pp. 295–296. [Online]. Available: http://doi.acm.org/10.1145/259963.260503

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

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

[4] 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*, vol. 52, no. 11, pp. 60–67, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1592761.1592779

[5] 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*, ser. ICER ’10. New York, NY, USA: ACM, 2010, pp. 69–76. [Online]. Available: http://doi.acm.org/10.1145/1839594.1839607

[6] 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.

[7] 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*, ser. SIGCSE ’03. New York, NY, USA: ACM, 2003, pp. 191–195. [Online]. Available: http://doi.acm.org/10.1145/611892.611966

[8] 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*, ser. ICER ’15. New York, NY, USA: ACM, 2015, pp. 91–99. [Online]. Available: http://doi.acm.org/10.1145/2787622.2787712

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

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

[11] R. Marinescu, “Detecting Design Flaws via Metrics in Object-Oriented Systems,” in *Proceedings of TOOLS*. IEEE Computer Society, 2001, pp. 173–182.

[12] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*, 2006. [Online]. Available: http://www.springer.com/alert/urltracking.do?id=5907042

[13] T. L. Alves, C. Ypma, and J. Visser, “Deriving metric thresholds from benchmark data,” in *26th IEEE International Conference on Software Maintenance (ICSM 2010)*. IEEE Computer Society, 2010, pp. 1–10.

[14] 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*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 390–400.

[15] 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*, vol. 36, no. 1, pp. 20–36, Jan. 2010.

[16] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: a multilinguistic token-based code clone detection system for large scale source code,” *TSE*, vol. 28, no. 7, Jul. 2002.

[17] F. Hermans, M. Pinzger, and A. van Deursen, “Detecting and refactoring code smells in spreadsheet formulas,” *Empirical Software Engineering*, vol. 20, no. 2, pp. 549–575, 2014. [Online]. Available: http://link.springer.com/article/10.1007/s10664-013-9296-2

[18] 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)*. ACM Press, 2012, pp. 451–460, to appear.

[19] 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*, Bremen, Germany, 2015, pp. 372–380.

[20] K. T. Stolee and S. Elbaum, “Refactoring Pipe-like Mashups for End-user Programmers,” in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE ’11. New York, NY, USA: ACM, 2011, pp. 81–90. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985805

[21] 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*, ser. SIGCSE ’13. New York, NY, USA: ACM, 2013, pp. 215–220. [Online]. Available: http://doi.acm.org/10.1145/2445196.2445265

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

[23] 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*, no. 46, pp. 1–23, Jan. 2015. [Online]. Available: https://doaj.org