Unmixing Remixes: The How and Why of Not Starting Projects from Scratch

Prapti Khawas, Peeratham Techapalokul, and Eli Tilevich
Software Innovations Lab, Department of Computer Science, Virginia Tech
{prappk19, tpeera4, tilevich}@cs.vt.edu

***Abstract*—One of the greatest achievements of Scratch as an educational tool is the eager willingness of programmers to use existing projects as the starting point for their own projects, a practice known as *remixing*. Despite the importance of remixing as a foundation of collaborative and communal learning, the practice remains poorly understood, with the Scratch programming community remaining in the dark about which programming practices encourage and facilitate remixing. Scratch designers lack feedback on how the remixing facility is used in the wild. To gain a deeper insight into remixing, this paper investigates heretofore unexplored dimensions of remixing: (1) the prevailing modifications that remixes perform on existing projects, (2) the impact of the original project's code quality on the granularity, extent, and development time of the modifications in the remixes, and (3) the propensity of the dominant programming practices in the original project to remain so in the remixes. Our findings can encourage remixing and improve its effectiveness, benefiting the educational and end-user programming communities.**

***Index Terms*—Scratch, block-based programming, remixing, program analysis, code quality, learning**

I. Introduction

The success of Scratch as an educational platform is nothing short of amazing. The public repository of Scratch projects contains close to 40 million projects written by computing learners of all ages coming from all over the world.[1] Although it would be hard to pinpoint a single reason for this success, and most likely it is a combination of factors, one of the design features of the Scratch blocks editor—"Remix[this project]"—immediately comes to mind. With a single click of a button, a Scratch programmer can clone any existing project, with the clone becoming the start of a development process. This ability to extend or modify someone else's work must have a tremendous positive influence on the productivity and level of satisfaction of Scratch programmers.

Remixing allows beginner programmers to quickly learn from, experiment with, and add to existing Scratch projects. Remixing has been studied previously. Dasgupta et al. have studied Scratch's effectiveness in building computational knowledge through remixing [1]. However, Hermans and Aivaloglou have established the prevalence of code smells in Scratch projects and the possibility of poor code quality inhibiting remixing [2]. Yet another study explored the relationship between the popularity of remixes and their program size [3]. Hill and Monroy-Hernández measured modification in remixes as edit distances [4], whereas we computed modifications as the script addition metric [5] in our earlier study. Our work builds on these prior studies, to explore the practice of remixing from a three-dimensional perspective that combines the nature of modifications, the impact of code quality, and the rate of adoption of programming practices.

To truly understand how computing learners take advantage of remixing requires a comprehensive exploration. Our goal is not only to increase the likelihood of positive learning outcomes but also to help improve the design of the language and its programming environment. By conducting our study, we seek answers to the following research questions: **RQ1:** How do programmers remix Scratch projects? **RQ2:** How does the code quality of a project impact the granularity, volume, and development time for the modifications made in its remixes? **RQ3:** Do the prevailing coding idioms of the original project influence their use in the remixes?

In this study, we collect a set of 160 representative Scratch projects with 15,010 remixes. We systematically preprocess and analyze this dataset to answer the research questions above. This paper contributes a targeted retrospective inquiry into the practice of remixing in Scratch. Among the novel aspects of our inquiry is a focus on the observed modifications in the remixes as a function of the original project's quality. Another one is an assessment of the influence of the coding practices in the original projects on those in their remixes.

II. Background and Related Work

This section provides the technical background and reviews the most closely related prior work.

Learning via Remixing: Prior work studied the impact of programmers engaging in project remixing on the development of their computational thinking (CT). Their findings indicate that programmers who remix more often then use a larger range of Scratch blocks, thus displaying their superior understanding of CT concepts [1]. Our work focuses on examining the adoption of advanced programming idioms, the issues of software quality, and their impact on remixes.

Advanced Scratch Language Features: Variables and broadcast-receive blocks have been identified as challenging for students [6]. Another study reports that custom blocks ("My Blocks") are the least used ones, based on a study of Scratch coding practices [7].

Code Quality Metrics: To extend any code, one must be able to understand it. Code quality is known to posi-

[1] https://scratch.mit.edu/statistics/

978-1-7281-0810-0/19/$31.00 ©2019 IEEE

tively correlate with software comprehensibility [8], [9]. Our analyses use the following software quality metrics: *LOC:* Straightforward to calculate, Lines Of Code (LOC) measure program size. Prior research reports the average size of Scratch projects to be around 140-150 LOC [5], [7]. Large projects may overwhelm novice programmers trying to understand them. *Halstead's Volume metric:* As part of the well-known Halstead's complexity metrics [10], *Volume*, calculated based on the number of operators and operands, is reported to negatively impact readability [11], [12]. *Cyclomatic complexity (CC):* CC measures the complexity of control flow [13]. A strong correlation has been reported between CC and the student-perceived code difficulty [14], and a weak correlation between CC and readability [15]. *ABC:* Introduced by Fitzpatrick, *ABC* estimates code complexity based on the number of assignments, branches, and conditional statements [16].

Code Smells in Scratch: Novice programmers find it hard to understand and modify Scratch programs with code smells [2]. The highly prevalent *Long Script* and *Duplicate Code* smells negatively correlate with remix modifications [5].

III. Methodology

We describe our data collection and research methodology.

Project collection: To obtain diverse and popularly remixed projects, we fetch 160 trending projects via the API[2] that is used to display them on the Scratch's "Explore" page. The collected projects in our dataset have between 1 to 700 remixes. We retrieve the remixes for each project via a remix tree API[3]. We collect only the first-level remixes; that is, the remixes that are the immediate children of one of the 160 subject projects. The final study dataset comprises a total of 15,010 Scratch projects.

Data Preprocessing: For each project, we parse its source in JSON format into the AST representation that is more easily amenable for computing quality measures. 3,297 remixes were disregarded as they contained Scratch 3.0's new extension blocks unsupported by our analysis. To detect the differences between the original project and its remixes, we convert the AST to a human-readable string to be able to apply text-similarity techniques inspired by *ranking*, a text-based similarity detection technique [17]. We collect statistics on various types of modification. 8% of the remixes with missing values were filtered out from the experimental dataset. 1% of the remaining remixes ended up being modified more than a 100%, indicating that the remix is a brand new project that shares no similarity with the original project. We conduct our analysis with and without these projects, as they can be associated with side-projects [18], in which the remixes diverge from the original project too much to skew our remixes-specific findings.

Measuring Modifications: Scratch projects can be remixed by adding, deleting, or replacing various program elements as well as by moving scripts around within the workspace, similarly to rearranging code in text-based programs. We refer to *modification size* as the total number of blocks in a remix that are added, deleted, or replaced as compared to the original project's code. Large projects are more likely to have remixes with large modification sizes as compared to smaller projects. Due to this finding, we measure modifications in *percentages* rather than raw numbers: $(\textit{modificationSize} \;/\; \textit{origProjectSize}) \times 100$. Since we study how remixes modify their original projects, we disregard the blocks added to new sprites in the remixes, as the code quality of the original projects has no impact on these modifications.

A. General remixing trends (RQ1)

RQ1.1: *How do programmers modify the project elements of remixed projects?* To understand the practice of remixing, we count the total numbers of insertion, deletion, and replace operations performed on the constituent elements of the remixed Scratch projects: Sprites, Costume, Sound, Variables (Global/Local), and Blocks. In addition, we capture the script rearrangements, that is, when programmers reposition an existing script within the Scratch workspace, and block categories that are most frequently altered.

RQ1.2: *Does the code quality change between the original projects and their remixes?* To assess the change in code quality for metrics discussed in Section II, we conduct a paired t-test (two-tailed) of the remixes. We formulate the null hypothesis that there is no difference in code quality between the original projects and their remixes. If the null hypothesis is rejected, we conduct lower-tailed t-tests to identify the increase/decrease in code quality.

RQ1.3: *What is the trend in the code quality of remixes once their original project is shared?* To provide insights into the programming practices of remixing over time, we analyze the time-series of code quality in remixes with low modification.

B. Impact of Code Quality on Remixing (RQ2)

RQ2.1: *Are poor code quality projects more likely to have remixes with smaller-sized modifications?* We investigate the code quality of remixed projects for three categories of remixes based on the modification percentages: None (0%), Low modification (0-50%) and High modification (50-100%).

RQ2.2*: Does code quality affect the time it takes to modify a project in its remixes?* Similar to *RQ2.1*, we examine three categories of remixes based on the time taken to modify: none ($\sim$0 min), low (0 min - 3.6 days) and high (3.6 - 66.8 days). These ranges were chosen based on the minimum, mean, and maximum values of modification time.

To calculate the code quality of Scratch projects, we adapt the code quality metrics, discussed in Section II. To compute *Halstead's volume*, we consider operands to be variable blocks or shadow (default value) blocks, and operators to be all blocks other than variable blocks. For *ABC*, we count the set and change blocks as assignments, the procedure calls and broadcast-receive blocks as branches; and the if-else blocks or blocks containing the operator blocks as conditionals. We apply the scalar value of the ABC metric to express code

[2]https://api.scratch.mit.edu/explore/projects?mode=trending

[3]https://scratch.mit.edu/projects/⟨project_id⟩/remixtree/bare/

Statistic	Min	Max	Mean
Original Projects			
Number of remixes	1	696	85
Program size (# blocks)	13	12632	498.83
Cyclomatic Complexity	2	38	4.888
Program Volume	39.069	3778.843	560.477
ABC	0	10983.706	247.787
Remixes			
Program size (# blocks)	3	9616	516.898
Cyclomatic Complexity	2	38	4.539
Program Volume	49.05	4424.014	804.881
ABC Metric	2	10983.706	320.679
Modification percent	0	890.458	4.88

TABLE I
SUMMARY STATISTICS OF 8,142 PROJECTS

quality. The higher is the ABC value, the lower is the code quality. Additionally, we compute the following two code quality metrics based on the detected code smells: *Long Script Density*, the ratio of long scripts (>11 blocks, as per previous finding [5]) to the total number of scripts, and *Duplicate Groups*, the total number of duplicate groups within a project.

C. Learning from Original Projects (RQ3)

The Scratch community fosters a comfortable environment for novice programmers to learn from each other. Thus, it is likely that existing Scratch projects do influence the programming practices of beginner programmers. To understand how original projects impact their remixes, we conduct an analysis with the goal of answering the following questions:

RQ3.1: *Do the project's sprite cloning constructs impact the number of duplicate sprites in its remixes?* We examine whether the number of "When I start as a clone" block in an original project is associated with the number of duplicate sprites in its remixes.

RQ3.2: *Does the presence of procedures in the original project motivate their increased use in its remixes?* We study projects containing custom blocks and examine whether new custom blocks are added to their remixes.

IV. RESULTS

Table I presents an overview of the projects in our dataset. Out of the initial dataset of 8,142 remixes, 2,758 were left completely unmodified from the original projects, and 371 contained only addition or deletion of sprites without any modification to the remaining sprites' code in original projects. The project sizes of the studied remixes ranged from 3 blocks to 9,616 blocks. We partition these projects into three categories based on size, with 7,920 projects in the small category (between 3 and 3,207 blocks), a size range that seems comfortable for programmers to remix, 151 projects in the medium category (3,207-6,412 blocks), and 71 projects in the large category (6,412-9,616 blocks), suggesting that large projects are hard to understand and modify. We present and discuss the answers to each research question next.

A. General remixing trends (RQ1)

RQ1.1: Table II indicates that sprites are more modified and deleted than added anew in the remixes. However, based on the sprites present in both the original projects and their remixes (i.e., disregarding the deleted and newly inserted sprites), we find that programmers more frequently insert blocks than delete them. Out of 8,142 projects, only 124 remixes either added or deleted variables. Surprisingly, the majority of the added variables are local, indicating an awareness of the scoping issues and preventing the introduction of the *Broad Variable Scope* smell, identified previously in [5].

Scratch Elements	Insertion	Deletion	Alteration
Sprite	0.253	0.484	1.365
Blocks	8.305	7.409	6.56
Costume	0.411	0.978	-
Sound	0	0	-
Global variable	0.003	0	-
Local variable	0.167	0.004	-

TABLE II
AVG. MODIFICATIONS IN 5,384 REMIXES (EXCLUDING UNMODIFIED REMIXES)

The remixes alter the block types in the original projects with the following distribution: Control (30.9%), Looks (25.2%), Event (14.5%), Data (14.1%), Motion (10.1%), Sound (4.1%) and Others (1.1%). We observe that programmers often alter control blocks in the remixes. In addition, we observe that around 35% of the altered blocks change some sprite attributes (i.e., Motion and Looks blocks) and add sound related functionality, thus warranting a comprehensive palette of animation and media blocks. In 36 projects, scripts are rearranged without modifications. The "Clean Up" has been applied only to 3 of these shared projects, thus implying that programmers may not care about how blocks are arranged in the project workspace.

RQ1.2: Based on the paired t-test (two-tailed) from Table III, we reject the null hypothesis of no difference in the three code quality metrics between the original projects and their remixes ($p < 0.05$). On conducting the lower-tailed t-tests for the three code quality metrics, we observe a decrease in all of the code quality metrics ($p < 0.05$) in remixes. This change is explained by the prevalence of deleting sprites as a part of remixing, thus lowering program complexity. The observed high number of unmodified remixes motivates the need for warning programmers about sharing duplicated projects, thus encouraging originality and creativity.

	ABC Metric	Prog. Difficulty	Complexity
t Stat	3.742	9.654	5.522
P (T≤t) one-tail	$9.193\text{x}10^{-5}$	$3.013\text{x}10^{-22}$	$1.718\text{x}10^{-8}$
t Critical one-tail	1.645	1.645	1.645
P (T≤t) two-tail	$1.838\text{x}10^{-4}$	$6.026\text{x}10^{-22}$	$3.438\text{x}10^{-8}$
t Critical two-tail	1.960	1.960	1.960

TABLE III
PAIRED T-TEST OF CODE QUALITY BETWEEN ORIGINAL AND ITS REMIXES

RQ1.3: We find that most remixes maintain the code quality levels of the original projects over time. In addition, for about a third of the original projects (31.88%), the code complexity of their remixes gradually increases and then plateaus out, as illustrated in Figure 1 with three such representative projects. One explanation is that the complexity of the original projects keeps increasing over time as they evolve, a trend reflected

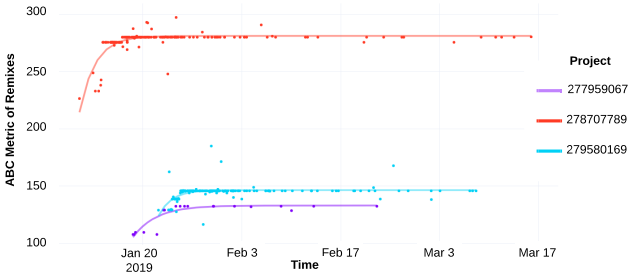

Fig. 1. A gradual increase of ABC in three sample projects

Metrics	Modification Size		
	None (∼0%)	Low (0-50%)	High (50-100%)
Code Quality			
Program Size	551.295	520.679	256.855
ABC	206.434	182.548	98.614
Program Volume	779.544	729.561	466.666
CC	4.251	3.765	3.764
Code Smell			
Long Script Density	0.204	0.203	0.169
Duplicate Groups	12.909	12.197	5.547

TABLE IV
COMPARING AVERAGE CODE QUALITY BASED ON MODIFICATION %

in their remixes. These findings are consistent with the established body of knowledge in software maintenance and evolution, as codified by the "Increasing Complexity" law of Software Evolution [19]: software complexity increases unless a concerted effort is put to maintain or reduce it. It is interesting to see this law manifesting itself in block-based software as well.

B. Impact of Code Quality on Remixing Modifications (RQ2)

RQ2.1: From Figure 2, we observe that the program size and the ABC are lower for bigger modification percentages and higher for unmodified projects. This trend continues for other quality metrics, as per Table IV. The ANOVA test reveals that these groups differ significantly in each of the cases ($p < 0.05$). This finding implies that high quality projects (i.e., projects with low metric values) are modified to a greater degree in their remixes. Or conversely, the remixes of lower quality projects tend to be modified to a lesser extent.

RQ2.2: From Figure 2, we observe that the program volume values are higher when the programmers have taken either no time or too long to modify. We observe this trend for other quality metrics as well, as described in Table V, except for CC. Based on the ANOVA test of these groups, we validate that they differ significantly in each of the cases ($p < 0.05$). This finding implies that programmers are more comfortable working with high quality projects. Or conversely, when remixing lower quality projects, programmers tend to take longer or avoid making changes altogether.

C. Learning from Original Projects (RQ3)

RQ3.1: In 5,187 remixes (modified $> 0\%$ in small category), we find that 75 remixes introduce duplicate sprites. Out of them, the original projects of 52 (69.3%) remixes contain

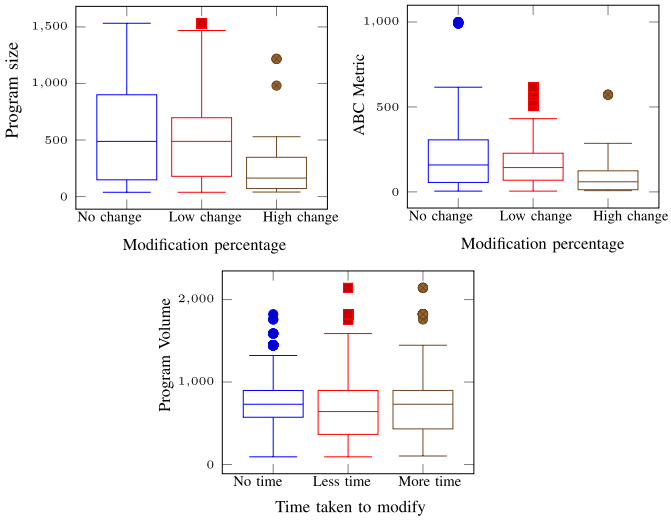

Fig. 2. Impact of code quality on remixing and time taken to remix

Metrics	Modification Time		
	None(∼0 min)	Short(0 min – 3.6 days)	Long(3.6 – 66.8 days)
Code Quality			
Program Size	583.806	495.239	523.242
ABC	200.934	173.932	186.339
Program Volume	760.964	706.015	752.394
CC	3.681	3.748	3.886
Code Smell			
Long Script Dens.	0.204	0.202	0.211
Duplicate Groups	13.861	11.769	12.982

TABLE V
COMPARING AVERAGE CODE QUALITY BASED ON MODIFICATION TIME

"When I start as a clone" block. Surprisingly, the presence of this block in a project seems to have *no* impact on preventing duplicate sprites in the project's remixes. Perhaps programmers find it hard to understand the concept of cloneable sprites or find it impossible to avoid duplicating sprites.

RQ3.2: From 5,187 remixes, 2,529 are derived from the original projects that contain procedures. Out of the 2,529 remixes, 36 projects (1.4%) add new procedures as part of their modifications of the original projects. Whereas, out of the remaining 2,658 remixes derived from the original projects with no procedures, only 7 (0.2%) of them add new procedures. This finding indicates that to organize code as procedures, programmers tend to follow the procedure usage trends established in an original project in its remixes.

V. CONCLUSIONS

We have conducted an empirical study of 8,142 remixes to understand and improve the remixing culture in Scratch. We explored the remixing behavior in terms of its general remixing trends, the impacts of code quality on the nature of remixes, and the influence of a project's programming practices on that in its remixes. A project's code quality and coding practices do affect how programmers modify code in its remixes. Our findings can be applied to encourage and expand remixing, as a highly effective communal learning technique that can also be better supported by programming environments.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable feedback that helped improve this manuscript. This research is supported by the National Science Foundation through the Grant DUE-1712131.

REFERENCES

[1] S. Dasgupta, W. Hale, A. Monroy-Hernndez, and B. M. Hill, "Remixing as a pathway to computational thinking," *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing - CSCW 16*, 2016.

[2] F. Hermans and E. Aivaloglou, "Do code smells hamper novice programming? a controlled experiment on scratch programs," *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 2016.

[3] B. M. Hill and A. Monroy-Hernández, "The cost of collaboration for code and art," *Proceedings of the 2013 conference on Computer supported cooperative work - CSCW 13*, 2013.

[4] ——, "The remixing dilemma: The trade-off between generativity and originality," *CoRR*, vol. abs/1507.01295, 2015. [Online]. Available: http://arxiv.org/abs/1507.01295

[5] P. Techapalokul and E. Tilevich, "Understanding recurring quality problems and their impact on code sharing in block-based software," *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2017.

[6] O. Meerbaum-Salant, M. Armoni, and M. M. Ben-Ari, "Learning computer science concepts with scratch," *Computer Science Education*, vol. 23, no. 3, pp. 239–264, 2013.

[7] E. Aivaloglou and F. Hermans, "How kids code and how we know: An exploratory study on the scratch repository," in *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ser. ICER '16. New York, NY, USA: ACM, 2016, pp. 53–61. [Online]. Available: http://doi.acm.org/10.1145/2960310.2960325

[8] J.-C. Lin and K.-C. Wu, "A model for measuring software understandability," in *The Sixth IEEE International Conference on Computer and Information Technology (CIT'06)*, Sep. 2006, pp. 192–192.

[9] ——, "Evaluation of software understandability based on fuzzy matrix," in *2008 IEEE International Conference on Fuzzy Systems (IEEE World Congress on Computational Intelligence)*, June 2008, pp. 887–892.

[10] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977.

[11] D. Alawad, M. Panta, and M. F. Zibran, "An empirical study of the relationships between code readability and software complexity," 2018.

[12] D. Posnett, A. Hindle, and P. Devanbu, "A simpler model of software readability," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985454

[13] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, Jul. 1976. [Online]. Available: https://doi.org/10.1109/TSE.1976.233837

[14] N. Kasto and J. Whalley, "Measuring the difficulty of code comprehension tasks using software metrics," in *Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136*, ser. ACE '13. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2013, pp. 59–65. [Online]. Available: http://dl.acm.org/citation.cfm?id=2667199.2667206

[15] R. P. L. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, July 2010.

[16] M. A. Kuznetsov and V. O. Surkov, "Analysis of complexity metrics of a software code for obfuscating transformations of an executable code," *IOP Conference Series: Materials Science and Engineering*, vol. 155, p. 012008, nov 2016. [Online]. Available: https://doi.org/10.1088/1757-899X/155/1/012008

[17] T. C. Hoad and J. Zobel, "Methods for identifying versioned and plagiarized documents," *Journal of the American Society for Information Science and Technology*, vol. 54, no. 3, pp. 203–215, 2003. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/asi.10170

[18] Y. Dong, S. Marwan, V. Catete, T. Price, and T. Barnes, "Defining tinkering behavior in open-ended block-based programming assignments," *Proceedings of the 50th ACM Technical Symposium on Computer Science Education - SIGCSE 19*, 2019.

[19] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, Sep. 1980.