Enhancing Block-Based Programming Pedagogy to Promote the Culture of Quality From the Ground Up

A Position Paper

Peeratham Techapalokul and Eli Tilevich
Software Innovations Lab
Virginia Tech
Blacksburg VA, USA
{tpeera4, tilevich}@cs.vt.edu

***Abstract*—Block-based programming has proven extraordinarily successful as a pedagogical tool for learning the fundamentals of computing via an exploratory, unconstrained, and hands-on approach. One would think that the issue of software quality is inapplicable in this programming domain. Nevertheless, as recent research shows, block-based programs, written by novice programmers, exhibit recurring quality problems. Although block-based software is not intended for production environments, poor quality can be detrimental to achieving the educational objectives the very use of blocks aims for. Specifically, as we and other researchers have been discovering, introductory programmers, as they gain proficiency, tend to retain poor programming habits, thus continuing to introduce recurring quality problems into their programs. Evidence also indicates that poorly written code is less likely to be reused, thus hindering the potential benefits of this peer-learning mechanism. These findings call for a synergistic effort from educators and tool developers to address the issue of software quality in the context of block-based programming. This effort will require innovating both in the realm of introductory computing curricula and software infrastructure to improve software quality.**

***Index Terms*—Software quality; Block-based programming; Introductory computing curriculum; Novice programmers; Software refactoring**

I. Introduction

Block-based programming has become a highly effective means to introduce novice learners to the computing discipline. Both introductory learners and end-users take advantage of the visual interface and syntax-free program compositions offered by block-based programming environments. Indeed, block-based programming witnesses a growing adoption in both formal and informal settings for learning programming. However, the success of block-based programming is more immediately evident in informal settings, with 19.6 million of Scratch programmers and over 23.7 million projects created and shared[1].

Block-based programming often conjures up the spirit of free exploration unbound by convention or guidelines. As a way to learn, novice programmers are encouraged to tinker with blocks. With this mindset in place, the issue of software quality gets relegated to the margins of the programming process if not disregarded altogether. Nevertheless, as we and other researchers are discovering, software quality problems in block-based programs are an issue whose impact surpasses that of temporary annoyance.

The initial research in this domain focused solely on identifying the presence of software quality problems, while more recent efforts shed light on how prone novice programmers are to introducing quality problems in their code, the negative effect of poor software quality on program comprehensibility and modifiability, as well as how the presence of recurring quality problems negatively affects the eagerness to reuse existing code when engaging in collaborative learning.

Specifically, several works have raised concerns about the poor quality of student-written block-based code. The very nature of Scratch leads students to following bottom-up exploratory programming practices, which go against well-established software engineering principles such as 'design before implementation [1].' Scratch projects written by high school have been shown to contain duplicated code and follow poor naming conventions [2]. The prevalence of quality problems in block-based programs has been substantiated in subsequent works, including an exploratory study of Scratch programs [3], a study of quality problems in block-based languages by Hermans et al. [4], and our own recent large-scale assessment of quality problems in Scratch projects [5]. A popular approach to evaluating software quality is to systematically identify and assess recurring quality problems based on the concept of "code smells," coding patterns indicative of implementation and design shortcomings known to degrade non-functional qualities (e.g., comprehensibility, modifiability, etc.). Hence, we have on our hands strong empirical evidence that confirms the presence of recurring quality problems in programs written by introductory students.

In this position paper, we offer our perspectives on the importance of software quality for introductory CS curriculum, including the negative implication of recurring quality problems on the formation of disciplined programming practices and the learning process in general. We outline our vision and research activities to promote the culture of quality from the ground up, an initiative that can make software quality an integral part of block-based programming pedagogy. By realizing our vision, we hope to enable introductory program-

[1]https://scratch.mit.edu/statistics/ (accessed July 2017)

ming students not only to obtain basic programming literacy, but also to internalize the value of software quality, and some of the proven quality enhancing practices followed by professional software developers.

II. Why should novice programmers care about software quality and quality improvement?

The block-based programming community, including students and CS educators, should treat the issue of software quality seriously for multiple reasons. Next, we outline what we see as the key points:

A. Software quality impact on modern society

Quality is essential to the everyday functioning of modern society. As Juran et al. put it in their Quality Handbook [6] "the importance of quality has continued to grow rapidly. To some extent, that growth is due in part to the continuing growth in complexity of products and systems, society's growing dependence on them, and, thus, society's growing dependence on those 'quality dikes'." Modern society critically depends on software, which is part and parcel of an ever-growing number of goods and services. Despite its wide utilization, software is known to suffer from one of the highest failure rates across all engineering artifacts due to its poor quality [7].

The quality of any product is determined by the production processes through which it is developed. To meet high quality standards, people involved in these processes must recognize quality as a critical requirement. In that light, as a society, if we are to drastically improve software quality, we need to accordingly condition the mindset of software developers in regards to this issue. Computing education has a huge role to play in this endeavor. To that end, we advocate a radical notion of *promoting the culture of quality from the ground up*. That is, we propose that software quality be taught alongside the very fundamentals of computing, and block-based programming is at the focal point of this initiative.

B. Code quality affects learning effectiveness

Poor code quality can negatively impact the very foundations of the learning process. Poor software quality hinders the computing learner's ability to read and understand code, the medium of communication in computing. As Martin Fowler put it "Any fool can write code that a computer can understand. Good programmers write code that humans can understand" [8]. The results of a controlled experiment by Hermans et al. [9] show that students find poor code quality programs hard to understand and modify. In our recent work, we also encounter the negative impact that poor software quality can have on the willingness of students to reuse and modify existing projects [5]. In particular, our results suggest that poor code quality can render a project to appear uninviting to other novice programmers to modify, as compared with projects enjoying similar levels of popularity. Thus, poor code quality undermines code sharing, or remixing, an essential learning activity in block-based programming [10].

C. Focusing on software quality can foster the adoption of good programming practices

The habits and discipline required to achieve and maintain good software quality cannot be taught directly, but rather can be promoted and cultivated. Will Durant articulates this principle thusly: "We are what we repeatedly do. Excellence then is not an act but a habit." As it turns out, disciplined programming practices fail to naturally arise in parallel with increases in programming proficiency. As a recent study by Robles et al. shows, students still copy and paste code even knowing how to avoid this harmful practice [11]. Our recent work [12] shows that students continue to introduce quality problems in their programs, even as their levels programming proficiency keep increasing. Thus, it is never too early to start educating students about software quality practices. Even introductory students should be encouraged to write not just working programs, but rather high-quality working programs.

III. Call for Action

Traditional engineering disciplines have long embraced the importance of quality, with engineering education treating this concept as an integral part of the professional curriculum [13]. Inspired by the vast quality improvements experienced by traditional engineering pursuits [14], we next outline our ambitious agenda for promoting the culture of quality in the context of introductory computing curriculum centered around block-based programming. This agenda comprises several interrelated tasks: understanding the problem, building a general knowledge base, and creating innovative pedagogical tools. Once these tasks are successfully carried out, the resulting pedagogical advances are expected to convince CS educators that it is feasible and useful to teach software quality starting from the very fundamentals of computing.

A. Getting a CS education community buy-in

Solving a problem of that magnitude requires a communal effort, and the first step is to get a buy-in from the CS Education community about the importance of promoting the culture of quality from the ground up. The first step in this endeavor is to draw attention of the community to the very presence of the issue of quality. Some research efforts focus on solving this exact problem, with recent studies collecting strong empirical evidence of the presence of recurring quality problems in block-based software, assessing the negative effects of poor code quality, and others as outlined in Section II-B. Other recent findings show that one can effectively teach introductory K-12 students about proper software engineering principles, which include treating software quality as an important consideration [15]. Nevertheless, more research in this area is needed to provide definitive empirical evidence about not only the negative consequences of neglecting the issue of quality in introductory computing curriculum, but also how innovative education interventions in this realm can improve various learning outcomes.

B. Explicit strategies of software quality practices in blocks

Promoting the importance of software quality and its practices will require creating new knowledge and materials to properly support student learning from the conceptual and skills perspectives. A promising direction worth investigating further is Soloway's rigorous instructional framework [16], which observes expert programmers possessing tacit knowledge and strategies, that unconsciously guide their programming practices. By analogy, quality should also be teachable in a form of explicit strategies. In other words, the software quality control and improvement practices of expert practitioners can be captured as explicit strategies to be taught to students.

Conceptual challenges lie in adapting these relevant quality improvement strategies to the context of block-based programming and presenting them at the level that novice programmers can easily internalize.

In terms of the specific strategies, in our work we focus on quality-related coding patterns. Originally developed and used in the architecture field, the concept of patterns has also been widely applied in software design [17]. Patterns codify expert knowledge, and are commonly described along with their motivation, rationale, context, and resulting behavior. For example, code smells are code patterns indicative of design problems [8], and have been a recent research focus for studying quality problems in block-based software. That is, code smells provide explicit strategies for developers to recognize quality problems in the codebase and a shared vocabulary for them to communicate about these problems.

What has been left relatively unexplored so far are common code patterns, which are conducive to producing high quality software. Certainly, the aim of introducing such quality improving patterns to introductory students is not to require that they closely follow these exact patterns, or that they memorize these patterns by rote. In contrast, being introduced to these patterns can help students develop an appreciation of the underlying rationale and motivation of these patterns. This appreciation will promote the culture of quality and influence how students approach the programming process.

C. Software infrastructure for quality improvement

Popular block-based programming environments can be enhanced to allow novice programmers to engage in software quality improvement practices as a natural extension of other learning activities. Specifically, a refactoring infrastructure that automates behavioral preserving transformations, can be very effective as a means to promote continuous quality improvement practices. There are several challenges that stand on the way of realizing this vision. For one, thus far, refactoring has been applied almost exclusively in the domain of text-based languages. It remains an open question how refactoring should be provided in block-based programming environments.

One important requirement is that refactoring support be made intuitive and friendly for novice programmers. One concern is that refactoring may seem magical to students, who may not have yet developed a complete picture of what is happening behind the scenes. Therefore, special care must be taken to make the refactoring process transparent to students. We posit that drag-and-drop interfaces, which have been previously applied to improve refactoring usability in text-based development environments [18], to be also well-applicable for block-based environments. Since drag-and-drop mechanisms are innate to blocks, one can leverage this feature to build refactoring gestures in this computing domain. One potential educational benefit is that refactoring support will render software as a malleable artifact to novice programmers, something that can be intuitively transformed at will, improving structure and design in the process.

Figure 1 shows one possible way to make *Extract Custom Block* refactoring intuitive. A programmer can invoke this refactoring by selecting a region of blocks and dragging it outside of the original script's area.

Additionally, to increase the educational value of the refactoring tool, following a learner-centered design [19] (i.e., intelligent tutoring system or scaffolding), quality improvement practices can be made more accessible to novice programmers. Augmenting the programming environment to support quality improvement practices will allow individualized, feedback and guidance required for effective learning in both formal and informal settings.

Nevertheless, the success of quality improvement support will require further study to evaluate the effectiveness of the provided tools in how effectively they promote software quality practices among novice programmers. This infrastructure development would be best introduced as part of a broader pedagogical intervention that includes in-class presentations and homework assignments that highlight the importance and benefits of refactoring. In addition to students learners, end users will also benefit by getting the ability to systematically improve the quality of their block-based software.

IV. Conclusion

Software quality can no longer be ignored as an irrelevant issue in the context of block-based programming and introductory computing curricula. While proven as a viable avenue for introductory computing, block-based programming and its pedagogy can be effectively enhanced with software quality concepts and practices. If as a society we are to improve the overall quality of our software infrastructure, we have to change the software development culture, and this change can be initiated from the introductory computing curriculum via block-based programming pedagogy.

Acknowledgements

This research is supported in part by the National Science Foundation through the Grant DUE-1712131.

References

[1] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari, "Habits of programming in scratch," in *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*. ACM, 2011, pp. 168–172.

[2] J. Moreno and G. Robles, "Automatic detection of bad programming habits in Scratch: A preliminary study," in *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*. IEEE, 2014, pp. 1–4.

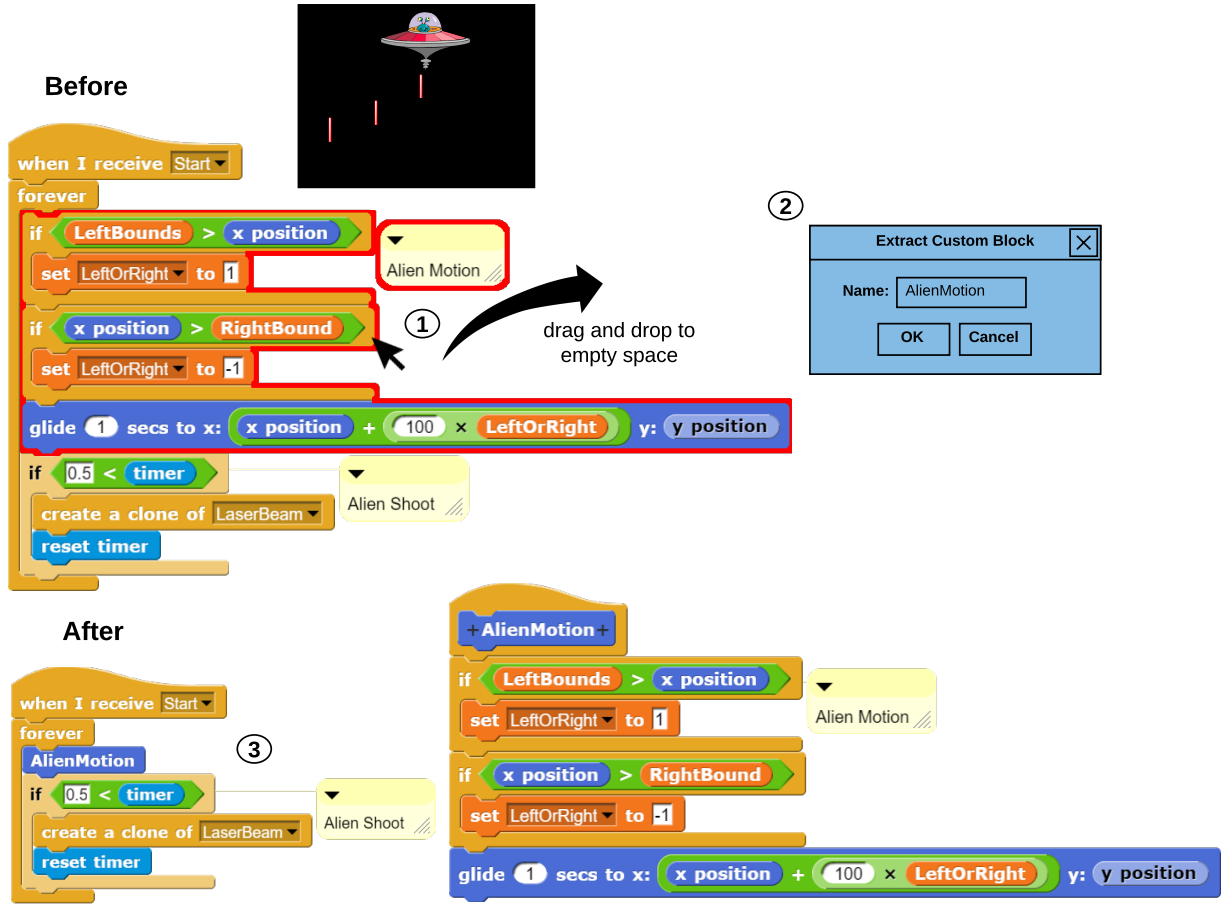

Figure 1. A refactoring interface mockup for a block-based programming environment

[3] 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.

[4] F. Hermans, K. T. Stolee, and D. Hoepelman, "Smells in block-based programming languages," in *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Sept 2016, pp. 68–72.

[5] P. Techapalokul and E. Tilevich, "Understanding recurring quality problems and their impact on code sharing in block-based software," in *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2017.

[6] J. Juran and A. B. Godfrey, "Quality Handbook," *Republished McGraw-Hill*, pp. 173–178, 1999.

[7] C. Jones and O. Bonsignour, *The economics of software quality*. Addison-Wesley Professional, 2011.

[8] M. Fowler and K. Beck, *Refactoring: Improving the design of existing code*. Addison-Wesley Professional, 1999.

[9] F. Hermans and E. Aivaloglou, "Do code smells hamper novice programming? A controlled experiment on Scratch programs," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–10.

[10] S. Dasgupta, W. Hale, A. Monroy-Hernández, and B. M. Hill, "Remixing as a pathway to computational thinking," in *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*, ser. CSCW '16. New York, NY, USA: ACM, 2016, pp. 1438–1449.

[11] G. Robles, J. Moreno-León, E. Aivaloglou, and F. Hermans, "Software clones in Scratch projects: On the presence of copy-and-paste in computational thinking learning," in *Software Clones (IWSC), 2017 IEEE 11th International Workshop on*. IEEE, 2017, pp. 1–7.

[12] P. Techapalokul and E. Tilevich, "Understanding recurring software quality problems of novice programmers," Virginia Tech, Article, 2017. [Online]. Available: http://hdl.handle.net/10919/78337

[13] M. Zairi, *Total quality management for engineers*. Elsevier, 1991.

[14] S. B. Knouse, P. P. Carson, K. D. Carson, and R. B. Heady, "Improve constantly and forever: The influence of w. edwards deming into the twentyfirst century," *The TQM Journal*, vol. 21, no. 5, pp. 449–461, 2009.

[15] F. Hermans and E. Aivaloglou, "Teaching software engineering principles to K-12 students: A MOOC on Scratch," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering and Education Track*, ser. ICSE-SEET '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 13–22.

[16] E. Soloway, "Learning to program = learning to construct mechanisms and explanations," *Communications of the ACM*, vol. 29, no. 9, pp. 850–858, 1986.

[17] S. Berczuk, "Finding solutions through pattern languages," *Computer*, vol. 27, no. 12, pp. 75–76, 1994.

[18] Y. Y. Lee, N. Chen, and R. E. Johnson, "Drag-and-drop refactoring: intuitive and efficient program transformation," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 23–32.

[19] C. Quintana, J. Krajcik, and E. Soloway, "Issues and approaches for developing learner-centered technology," *Advances in computers*, vol. 57, pp. 271–321, 2003.