Programming Environments for Blocks Need First-Class Software Refactoring Support

A Position Paper

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

***Abstract*—Block-based programming languages and their development environments have become a widely used educational platform for novices to learn how to program. In addition, these languages and environments have been increasingly embraced by domain experts to develop end-user software. Though popular for having a "low floor" (easy to get started), programs written in block-based languages often become unwieldy as projects grow progressively more complex. Software refactoring—improving the design quality of a codebase while preserving its external functionality—has been shown highly effective as a means of improving the quality of software written in text-based languages. Unfortunately, programming environments for blocks lack systematic software refactoring support. In this position paper, we argue that first-class software refactoring support must become an essential feature in programming environments for blocks ; we present our research vision and concrete research directions, including program analysis to detect "code smells," automated transformations for block-based programs to support common refactoring techniques, and integration of refactoring into introductory computing curricula.**

***Index Terms*—refactoring, metrics, code smells, block-based programming languages, end-user software engineering, computer science curriculum, introductory programming.**

I. Introduction

As a software project grows, so does the complexity of its source code. Without targeted and timely programming efforts to counter this complexity, it can quickly overwhelm the average programmer. *Software refactoring*—changing a program's structure to improve the program's design while preserving its external functionality—has become an indispensable part of the modern software development process. Modern text-based programming environments support refactoring via *refactoring browsers*, sophisticated program analysis and automated transformation engines; they enable the programmer to *refactor with confidence*, with assurance of refactoring transformations not changing the program in unexpected ways. This first-class refactoring support, which has become an inextricable part of modern integrated development environments (IDEs), serves as one of the pillars of the iterative programming process, in which development is intermingled with a continuous stream of improvement and enhancement tasks[1], [2].

Block-based programming languages, including Scratch and Blockly, have become highly popular by providing both a "low floor" for novices to get started and a "high ceiling" for more experienced programmers to create increasingly complex projects over time. Despite their proven educational utility, programming environments for blocks are still in infancy compared with text-based languages when it comes to supporting the software development process with state-of-the-art software development tools, based on program analysis and automated transformation.

Nevertheless, software engineered in block-based languages suffers from the same problems of design rot and code quality deterioration, as programs become increasingly complex. The problem is real. Scratch projects authored by novice programmers have been found to contain a large quantities of message passing and scripts scattering around[3]. Other prevalent design problems included duplicate code, uncommunicative name, excessive use of concurrent scripts, etc. These manifestations of unnecessary design complexity and code quality degradation make software written in block-based programs hard to comprehend, maintain, and evolve.

We argue that the best practices of systematic refactoring support for text-based languages can and should be applied to block-based languages and environments, so that block-based software can reap similar software engineering benefits. To that end, programming environments for blocks should be enhanced to provide support for program analysis and transformation. State-of-the-art support is required to enable block-based language programmers to become aware of design problems as they arise by ascertaining standard software metrics used to measure code quality, to discover "code smells" to uncover the symptom of poor design, and finally, to transform programs automatically with a refactoring tool to safely and efficiently improve their design.

The expected benefits will be immediately tangible: block-based software that is easier to understand, modify, and evolve. By systematically growing the refactoring support for block-based languages, one can effectively raise their "high ceiling" property; first-class refactoring support can equip the average programmer with powerful automated tools for managing

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

Authorized licensed use limited to: University of Southern Denmark. Downloaded on October 23,2025 at 12:04:02 UTC from IEEE Xplore. Restrictions apply.

program complexity. Systematic refactoring support can also advance the role of block-based programming for end-user software development. In the following discussion, we present the research ideas and directions that will form the basis of this effort as well as discuss some possible paths for integrating refactoring into introductory computing curricula.

II. Capturing Design Quality

Without a way to measure the design quality of a program, one cannot be certain about when and where the program needs improvement. Thus, essential to modern software development process are the tools to evaluate the design quality and to detect bad design. These can be achieved by using software metrics, and identifying "code smells", respectively.

A. Software Metrics

Software metrics aids programmers in understanding applications, getting an overview of a large system, and identifying potential design problems [4]. Most of the metrics proposed in the literature focus exclusively on text-based languages [4], [5], [6]. For block-based languages, one promising research direction would be to focus on the set of metrics to capture both structural quality and visual organization.

Since the visual aspect is prominent for block-based languages, research questions should seek to answer how metrics can be formulated to capture the software quality concerning with the visual organization of the code. As suggested by Conversy [7], the representation of code should rely on the capability of the human visual system. "Programs must be written for people to read, and only incidentally for machines to execute[8]." Aside from the spatial layout enforced by languages' syntax, it would be interesting to explore how one can measure the design quality (e.g., how the readability of block-based programs is affected by visual grouping of scripts.)

B. Code Smells

The concept of "code smells" is commonly used to identify various structural characteristics of software indicative of design problems. The most notable work in this area is the refactoring book by Fowler [9], which documents code smells and the corresponding refactoring steps to remove them. Evidence from both educators and researchers [10], [3], [11] points out both structural (e.g., spaghetti code, duplicate code) and visual organization shortcomings (e.g., the scattering of small scenario-based scripts that cause "a confusing visual effect"), which are prevalent in the Scratch language family.

For block-based languages, language-independent code smells (e.g., *Long Script, Duplicate Code, etc.*) can be adapted from the existing ones and more specific smells can be formulated to capture the design shortcomings unique to block-based language features for both structural and visual organization aspects. A set of software metrics described previously can be composed to capture high-level smells similar to [4]. To automate the task of discovering code smells or refactoring opportunities, a targeted research effort would need to focus on formalizing the code smells and analyzing statistically the thresholds of the metrics used. Realizing these research ideas as a general framework for building automated code smell detection tools for programming environments for blocks can provide a practical tool for a wide programming audience.

III. Software Refactoring Support for Block-Based Languages

As discussed above, software refactoring[12] has become an intrinsic component of any non-trivial software development effort. The quality of all software tend to degrade over time, unless special efforts are put in place to actively counteract the process. Software engineered in block-based languages is subject to the same quality pressures [3], [11]. Practical refactoring hinges on automated program transformation tools, as refactoring by hand is often tedious and error-prone.

Figure 1 shows an example of a manual refactoring that extracts a custom block (a.k.a. *Extract Method*), one of the most commonly used refactoring transformations in text-based development environments[1]. The simple Snap! program segment in Figure 1 makes a character jump differently depending on which keyboard (space bar or up arrow) is pressed. Though seemingly simple, the code in the *before* version is not easy to read—a fragment of code represents jumping, yet it may not be so obvious to the programmer trying to understand it. Adding comments to the source code to clarify its meaning is sometimes used as a simple fix, often a good indication of poor code readability. What we have here is a unit of potentially reusable functionality that can be used in multiple places in the program. Unfortunately, when not encapsulated within a method, a code fragment can only be reused by means of copy-and-paste, thus replicating a code fragment in several places within a project. This repeated sequence of code is a common "code smell" known as *Duplicate Code*. As a result, not only the code is hard to read, but it is also hard to change, as the changes need be made everywhere in all duplicates (e.g., consider the changes one would need to make to add a gravity effect to the jump). These issues would be further exacerbated in the context of a large, real-world program.

In contrast, the code in the *after* version is improved with respect to readability and reusability. In particular, when a custom block is given a descriptive name, the improved code is self-documenting, with a custom block name explaining the high-level intent of the code fragment. The resulting program is easier to read, with custom block names serving as code comments. Extracting a code fragment into a custom block increases the chance of it getting reused elsewhere in the project.

One can see how simple refactoring transformations can improve program comprehension and reusability, reducing both code duplication and complexity. Unfortunately, modern programming environments for blocks have limited support for automated program transformation, required to support refactoring. As a result, programmers can only refactor their block-based programs by hand.

Authorized licensed use limited to: University of Southern Denmark. Downloaded on October 23,2025 at 12:04:02 UTC from IEEE Xplore. Restrictions apply.

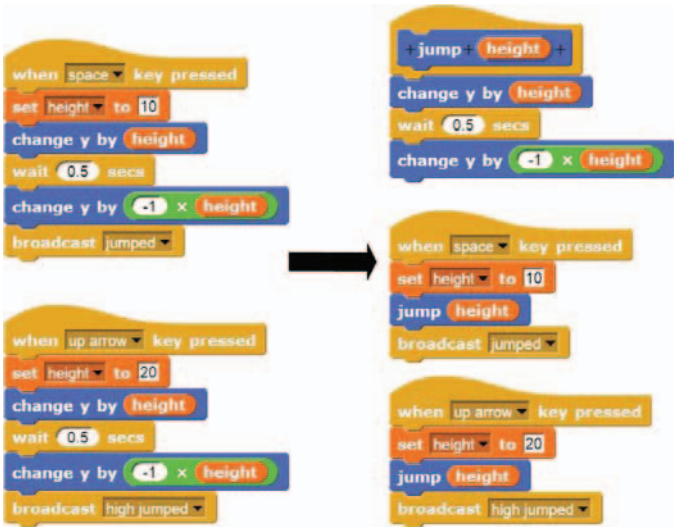

Fig. 1. An example of the code *before* and *after* performing (by hand) the *Extract Custom Block* refactoring in Snap!

What constitutes a conceptual challenge is formulating a set of systematic guidelines to determine which refactoring techniques are necessary for block-based programming and should be exposed as automated refactoring transformations within the development environments. In other words, which design problems plague block-based software most commonly, and which of these problems are amenable to a refactoring treatment. Because the refactoring research has almost exclusively focused on text-based languages, the designers of programming environments for blocks find themselves in need of guidelines tailored specifically for them.

Another important research effort in that realm should focus on the creation of internal program representations and analysis tools in support of refactoring. Although some block-based languages can be represented as text, analyzing the textual representation may not be sufficient for the objectives at hand. What is needed are internal representations that reflect not only the semantic, but also the visual organization of a block-based program. In other words, the refactorings should be mindful of how blocks are displayed on the screen, so as to increase program readability. A promising direction is to use the refactoring techniques developed for text-based languages as a starting point, and then modify and enhance them as necessary for the needs of block-based programming.

IV. Refinement-Centric Approach to Block-Based Software Development

The pedagogical effectiveness of block-based languages and their development environments is evident. These languages, including Scratch, Snap!, and Blockly, have witnessed increasing adoption as a means of teaching introductory programming. Unfortunately, with the current lack of powerful refactoring support as discussed above, programming environments for blocks can inadvertently condition novice programmers to view programming as a single-step linear process, unaligned with the iterative process of enhancement and improvement, an intrinsic part of modern software development.

Although programmers can always refactor their block-based programs by hand, the process can be tedious, error-prone, and in some cases equivalent to rewriting from scratch in complexity. As programs grow in complexity, programmers are likely to forgo even simple refinements, thus adversely impacting software quality with respect to modularity, cohesion, coupling, etc. Forgoing refinements is an ill-conceived programming habit that gives rise to potentially insurmountable difficulties in comprehending, maintaining, and reusing software.

Making refactoring tools available for block-based languages will highlight the importance of continuous code improvement in the minds of novice programmers, thereby improving the pedagogical effectiveness of block-based programming. Thus, the outlined research directions are concerned with instantiating the theoretical bases of refactoring in block-based languages and systematically evaluating the developed technologies with diverse student audiences.

References

[1] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE 31st International Conference on Software Engineering*, pp. 287–297, 2009. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5070529

[2] K. Beck, *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

[3] O. Meerbaum-salant, R. Israel, and M. Ben-ari, "Habits of Programming in Scratch," 2011.

[4] M. Lanza, R. Marinescu, and S. Ducasse, *Object-Oriented Metrics in Practice*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[5] K. K. Chahal and H. Singh, "Metrics to study symptoms of bad software designs," *SIGSOFT Softw. Eng. Notes*, vol. 34, no. 1, pp. 1–4, Jan. 2009. [Online]. Available: http://doi.acm.org/10.1145/1457516.1457522

[6] K. A. M. Ferreira, M. A. S. Bigonha, R. S. Bigonha, L. F. O. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics," *J. Syst. Softw.*, vol. 85, no. 2, pp. 244–257, Feb. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2011.05.044

[7] S. Conversy, "Unifying textual and visual: A theoretical account of the visual perception of programming languages," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2014. New York, NY, USA: ACM, 2014, pp. 201–212. [Online]. Available: http://doi.acm.org/10.1145/2661136.2661138

[8] G. Sussman, H. Abelson, and J. Sussman, "Structure and interpretation of computer programs," 1983.

[9] M. Fowler, K. Beck, J. Brant, and W. Opdyke, "Refactoring: Improving the design of existing code."

[10] "Scratch project forum discussion," http://scratched.gse.harvard.edu/discussions/computer-science-education/what-are-worst-scratch-programming-practices, accessed:2015-07-20.

[11] M. Gordon, A. Marron, and O. Meerbaum-Salant, "Spaghetti for the main course?: Observations on the naturalness of scenario-based programming," in *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '12. New York, NY, USA: ACM, 2012, pp. 198–203. [Online]. Available: http://doi.acm.org/10.1145/2325296.2325346

[12] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, Feb. 2004. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1265817

Authorized licensed use limited to: University of Southern Denmark. Downloaded on October 23,2025 at 12:04:02 UTC from IEEE Xplore. Restrictions apply.