Manual Refactoring (by Novice Programmers) Considered Harmful

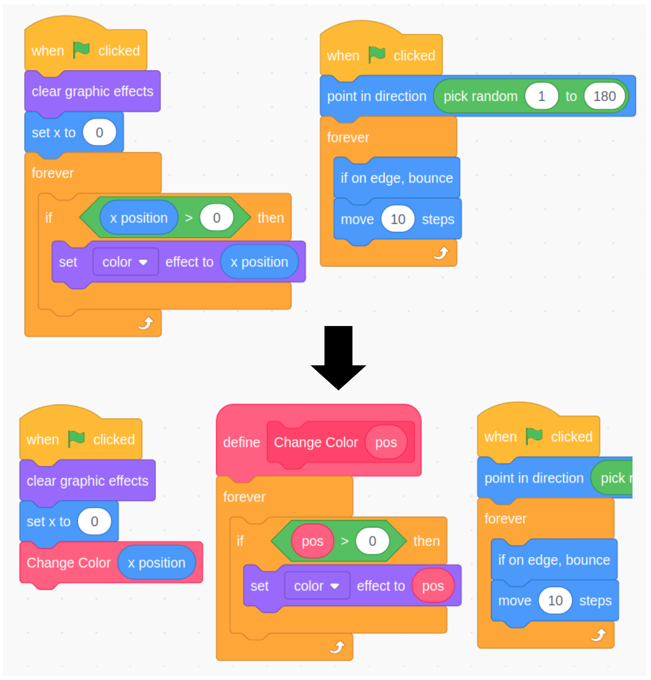
Peeratham Techapalokul and Eli Tilevich
Software Innovations Lab
Dept. of Computer Science, Virginia Tech
{tpeera4, tilevich}@cs.vt.edu

***Abstract*—In recent years, the issue of quality in introductory and end-user programming has become increasingly prominent. In particular, the poor quality of block-based programs has been shown to harm the educational effectiveness in this programming domain. As a result, there has been a growing interest in code quality and its improvement practices among researchers and educators. Refactoring—transforming code to remove quality problems while preserving semantics—is taught as a common and practical technique for improving code quality. To maximize pedagogical effectiveness, some educators advocate teaching students to refactor by hand to elucidate the mechanics behind refactoring instead of relying on tools that transform code automatically. In this position paper, we argue that this pedagogical approach is counterproductive. We advocate teaching refactoring similarly to compilation, a technique that students start applying automatically right away, but learn its inner workings only much later in the curriculum. Even professional developers are advised to avoid carrying out complex refactoring transformations by hand, as this activity leads to hard-to-trace bugs as well as wastes time and effort. To put appropriate tools for improving software quality into the hands of Scratch programmers, we created an automated refactoring infrastructure for Scratch, and we argue that such facility should become a mainstay of programming environments for blocks.**

***Index Terms*—block-based programming, refactoring, code quality, introductory computing, Scratch**

INTRODUCTION

Software quality is important, but requires a concerted effort to sustain. Software refactoring is a common practice for improving the quality of existing code while keeping its functionality intact. That is, refactoring is a semantics-preserving program transformation [1]. For example, applying RENAME refactoring to function *foo* to change its name to *printCustomer* makes the program easier to understand by providing a descriptive name for this function, but it does not change what the program does. Although programmers can apply a refactoring by transforming the source code by hand, in modern software development, refactorings are executed by means of automated program transformation tools, operated under the programmer's direction. The difficult aspect of refactoring is ensuring its semantics-preserving nature. To that end, automated refactoring tools integrate powerful program analysis modules that determine and check a set of preconditions before carrying out any refactorings. Checking refactoring preconditions by hand can be non-trivial, requiring careful attention to detail. Refactoring by hand can introduce unexpected semantic changes or hard-to-trace bugs.



```
when [flag] clicked
clear graphic effects
set x to (0)
forever
    if <(x position) > (0)> then
        set [color] effect to (x position)

when [flag] clicked
point in direction (pick random (1) to (180))
forever
    if on edge, bounce
    move (10) steps
```

```
when [flag] clicked
clear graphic effects
set x to (0)
Change Color (x position)

define Change Color (pos)
forever
    if <(pos) > (0)> then
        set [color] effect to (pos)

when [flag] clicked
point in direction (pick r
forever
    if on edge, bounce
    move (10) steps
```

Fig. 1. Introducing a bug via manual refactoring.

The example program in Figure 1 is a simple animation of a character bouncing around and, when in the stage's right side, changing its color in such a way that the color's value corresponds to that of the character's x coordinate. Consider a novice programmer deciding to extract the color changing code into a separate procedure by carrying out the *Extract Procedure* refactoring by hand. The refactored code shows the procedure's parameter *pos* replaces each *x position* block in the procedure's body. The inadvertently introduced bug in the refactored code causes the character to stop changing its color. By contrast, an automated refactoring tool would reject this refactoring attempt up front. It would detect that *x position* is a non-constant expression block, while any introduced parameter would remain unchanged across loop iterations due to Scratch's pass-by-value parameter passing semantics.

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

Refactoring is closely related to compilation. Both processes transform programs: refactoring transformations are within the same language, while compiler transformations typically convert programs in a higher-level language to a lower-level one. Both transformations are semantics-preserving. If introductory learners are introduced to computing by using a compiled language, they are taught how to compile programs as an intrinsic part of the curriculum. The instructor would mention that compilation translates a program to executable code, but without explicating the compilation process.

We argue that we should teach introductory students to improve the code quality of their programs by applying automated refactoring tools. If convincing budding programmers that code quality is important, automated refactoring tools should become a standard part of their programming arsenal. Novice programming environments should be able to highlight code quality problems, removable via automated refactorings put at the programmer's fingertips. Similarly to how we teach students to compile their code to create an executable version, we should also teach them to apply such automated refactoring tools to improve the code quality of their programs.

ARGUMENTS AGAINST MANUAL REFACTORING

For pedagogical reasons, some educators favor the idea of making refactoring transformations more transparent by teaching students how to refactor by hand first, before teaching them how to use any automated refactoring tools. Although many programming tasks lend themselves to such a bottom-up teaching strategy, refactoring is not one of them. For example, some IDEs can automatically generate much of boilerplate code; however, before using such code generators, students should learn how to write the generated code by hand. A similar argument applies to using an IDE's code completion facilities. Nevertheless, there are several dangers associated with learning refactoring in such a bottom-up way.

a) Performing tedious, error-prone transformations by hand kills the joy of coding and creativity flow: When designing for beginners, one must choose what black boxes are (i.e., what should remain hidden from the user's purview) [2]. If the main goal is for novice programmers to learn and engage their creativity with coding while picking up valuable code quality improvement practices, we argue that automated refactoring is the right black box for this educational design. Prior research shows that professional software developers refactor frequently [3]. With the low-level, error-prone, and cognitively demanding task of manual refactoring, it is hard to imagine that refactoring would become an integral part of the programming process of budding programmers instead of discouraging them from ever refactoring their code.

b) Manual refactoring goes against the bottom-up programming style of beginners: Without the RENAME refactoring, one can only imagine how cognitively taxing it would be for novice programmers to consistently rename variables scattered around numerous places in the code. This refactoring is necessitated by the complexity for novice programmers to be able to always come up with descriptive names for program identifiers. If this refactoring is forgone, any non-trivial codebase would quickly become incomprehensible. Another unintended consequence is that students may be compelled to think unnecessarily hard about this low-level implementation consideration during the design phase, a behavior incongruent with the bottom-up programming process that encourages student programmers to freely experiment with their code. Hence, automated refactoring is an important ingredient in bringing programming with blocks closer to the style of the iterative refinement process of modern programming practices.

c) Manual refactoring discourages novices from forming desirable programming habits: Novice programmers are yet to develop an appreciation for sustaining and improving the quality of their code. As they involuntarily engage in cost-benefit analysis, they could quickly get discouraged and even have negative experiences with the prescribed manual refactoring practices. Finally, if students persist with the practice of performing refactoring manually, they risk developing undesirable programming habits when transferring their skills to other software development activities. The complexity and requirements of modern software development call for powerful automated tools for improving software quality with automated refactoring support being part and parcel of all major IDEs. Modern development practices encourage the adoption of automated tools, including refactoring to both prevent errors and improve productivity.

AUTOMATED REFACTORING FOR BLOCKS

In our research, we have been enhancing Scratch with automated refactoring. Our refactoring infrastructure, QIS (pronounced as /chēz/) [4], is designed for novice programmers, providing on-the-fly contextualized coding hints that inform programmers about code quality problems (presented as improvement opportunities). Programmers can then decide to apply the suggested refactoring transformations associated with the hints. The refactorings in QIS are highly applicable to remove some of the highly recurring quality problems in Scratch codebase [5]. Our preliminary findings are promising, having shown to motivate Scratch novices to improve code quality. Nevertheless, automated refactoring for block-based programming remains at an early stage of integration, with great potential for introductory and end-user programming pursuits as well as exciting future research opportunities.

REFERENCES

[1] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[2] M. Resnick and B. Silverman, "Some reflections on designing construction kits for kids," in *Proceedings of the 2005 Conference on Interaction Design and Children*, 2005.

[3] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2011.

[4] P. Techapalokul and E. Tilevich, "QIS: Automated refactoring for Scratch," in *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct 2019.

[5] ——, "Code quality improvement for all: Automated refactoring for Scratch," in *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct 2019.